



From Technologies to Solutions

Programming Microsoft Dynamics NAV 2009

Develop and maintain high performance NAV applications to meet changing business needs with improved agility and enhanced flexibility

David Studebaker

PACKT
PUBLISHING

www.allitebooks.com

Programming Microsoft® Dynamics™ NAV 2009

Develop and maintain high performance NAV applications to meet changing business needs with improved agility and enhanced flexibility

David Studebaker



BIRMINGHAM - MUMBAI

Programming Microsoft® Dynamics™ NAV 2009

Copyright © 2009 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2009

Production Reference: 1271009

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847196-52-1

www.packtpub.com

Cover Image by Faiz Fattohi (faizfattohi@gmail.com)

Credits

Author

David Studebaker

Reviewers

Mark J. Brummel

Steven Renders

Acquisition Editor

Douglas Paterson

Development Editor

Ved Prakash Jha

Technical Editors

Aanchal Kumar

Charumathi Sankaran

Copy Editor

Sanchari Mukherjee

Indexer

Rekha Nair

Editorial Team Leader

Gagandeep Singh

Project Team Leader

Lata Basantani

Project Coordinator

Joel Goveya

Proofreaders

Claire Cresswell-Lane

Erica Mukherjee

Graphics

Nilesh Mohite

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

David Studebaker is Chief Technical Officer and the owner of Liberty Grove Software, Inc., with his partner Karen Studebaker. Liberty Grove Software, a Microsoft Partner, provides development, consulting, training, and upgrade services for Microsoft Dynamics NAV resellers and firms using Dynamics NAV internally.

David has been recognized by Microsoft as a Certified Professional for NAV in all areas – Development, Applications, and Installation & Configuration. He has been honored as a Lead Certified Microsoft Trainer for NAV. He has been programming since 1962 and developing in C/AL since 1996. David has been an active participant in each step of computing technology – from the early mainframes to today's technology, from binary assembly language coding to today's C/AL and C#.

David's special achievements include the development of the very first production SPOOLing system in 1967. Application areas in which David has worked include manufacturing, distribution, retail, engineering, general accounting, association management, professional services billing, distribution/inventory management, freight carriage, data collection, and production management among others.

David has had a wide range of development, consulting, sales, and management roles throughout his career. He has been partner or owner and manager of several software development businesses, while always maintaining a significant role as a business applications developer.

David has a BS in Mechanical Engineering from Purdue University and an MBA from the University of Chicago. He has been writing for publication since his undergraduate college days. David has been a member of the Association for Computing Machinery since 1963 and was a founding officer of two local chapters of the ACM.

Acknowledgement

I would like to especially thank my partner in life and at work, Karen Studebaker, for her unflagging support and encouragement in all ways since those early days at Purdue. No one could have a more wonderful partner or spouse. I would like to acknowledge the guidance and love that I received from my parents as well as the enthusiastic support and love of my wonderful children and other family members. Finally, though there are far too many individuals to list, whatever I have been able to accomplish would not have been possible without the help of many, many friends, mentors, and associates along the way. Life would be very poor without all the kind and generous folks I have met. I also wish to thank the great people at Microsoft and Packt who assisted me with their contributions and advice throughout the creation of this book. May you enjoy this book and find it useful.

A special thanks to these helpful people at Microsoft:

Microsoft Technical Reviewers:

Overall Coordination – Michael Nielsen, Director of Engineering, Microsoft Dynamics NAV.

Chapter 1: *A Short Tour through NAV 2009* - Esben Nyhuus Kristoffersen

Chapter 2: *Tables* - Thomas Hejlsberg

Chapter 3: *Data Types and Fields for Data Storage and Processing* - Thomas Hejlsberg

Chapter 4: *Pages – Tools for Data Display* - Esben Nyhuus Kristoffersen

Chapter 5: *Reports* - Yuri Belenky

Chapter 6: *Introduction to C/SIDE and C/AL* - Lars Hammer

Chapter 7: *Intermediate C/AL* - Lars Hammer

Chapter 8: *Advanced NAV Development Tools* - Lars Hammer, Hans Kierulff

Chapter 9: *Extend, Integrate, and Design – into the Future* - Christian Abeln, Bardur Knudsen

Dynamics NAV Help documentation: Paul Chapman, Dynamics NAV 2009 Documentation Manager, and his team, including Jill Frank, Søren Groes-Petersen, John Swymer, and Bob Papsdorf

Dynamics NAV UX Guide: Hans Roed Mark, UX Manager, Microsoft Dynamics User Experience Team

About the Reviewers

Mark J. Brummel is an all-round Microsoft Dynamics NAV specialist. He started in 1997 as an end user but quickly moved to the other side of the table. During ten years, he has worked for resellers where designing and maintaining add-on systems was his specialization. Some of these add-on systems exceed the standard product where it comes to size and complexity. In addition, coaching colleagues and troubleshooting 'impossible' problems is his passion and part of day to day work. Mark has trained most of the experienced NAV developers for the NAV 2009 product in The Netherlands and Belgium. Today he is working freelance, is hired by almost every NAV reseller in the Benelux area, and is also frequently asked to help out in escalated implementations by end users. Mark is an associate in the Liberty Grove Software network and a business partner of SQL Perform Benelux. Mark was the first to use the NAV 2009 (CTP3) product in a production system feeding back valuable information to Microsoft.

A special project and passion is performance tuning of the Dynamics NAV product on SQL Server. Since 2007, he is involved in the development of the 'SQL Perform Tools'. A specialist's toolset which allows both trend and escalation analysis of key elements for systems speed. As a unique specialist, he has done break-through research in improving the performance of Dynamics NAV on SQL Server.

In his free time, Mark maintains his blog on www.brummels.com. This blog contains a wide range of articles about both the Microsoft Dynamics NAV and SQL Server product. He is also a frequent speaker at Microsoft events. In 2006, Mark was rewarded by Microsoft with the *Most Valuable Professional* award for his contribution to the online and offline communities. In 2007, he also reviewed *Programming Microsoft® Dynamics™ NAV*.

Steven Renders is a Microsoft Certified Trainer in Microsoft Dynamics NAV. He has more than 12 years of business and technical experience. He joined Plataan in 2006, where he provides training and consultancy focused on Microsoft Dynamics NAV development, Microsoft SQL Server, Business Intelligence solutions, Microsoft SQL Server Reporting Services, and Database Performance Tuning. He is also an expert on Dynamics NAV 2009, on which he has already delivered many training sessions. Steven has also developed content for Microsoft Learning.

Foreword

Since the first version of Dynamics NAV, simplicity has always been the biggest asset of the product, and the goal has always been that it should be easy to learn, easy to use, and easy to develop. For the NAV developers, this has been accomplished by limiting the number of concepts they have to learn.

The first is to use the concepts which are well known from real life, for example, Form, Page, Table, and Report. The next is to introduce a programming language C/AL, which is targeted at writing business logic and not device drivers. The third is to add an integrated development environment, which removes the need for "plumbing" logic, which typically pollutes the code in a normal development environment. The fourth is to add automatic transaction and error handling, which saves the developers a lot of time since they don't have to write "clean up" code. The fifth and probably most important, is to reuse code constructs across the application, so that once you have learned one subsystem, it is easy to understand and master the rest. The latter is the secret sauce of NAV and what makes it possible to master doing customization across the whole NAV application.

This is, in very few words, what David's book is all about and what Dynamics NAV is all about.

Michael Nielsen,
Director of Engineering,
Microsoft Dynamics NAV

Table of Contents

Preface	1
Chapter 1: A Short Tour through NAV 2009	13
NAV 2009: An ERP system	14
Financial Management	16
Manufacturing	16
Supply Chain Management (SCM)	17
Business intelligence and reporting	18
Relationship Management (RM)	18
Human Resource management	19
Project management	19
Significant changes in NAV 2009	19
Two-tier versus three-tier	20
Role Tailored Client	21
SSRS-compatible report viewer	21
Web services	21
NAV 2009: A set of building blocks and development tools	21
NAV object types	22
The C/SIDE Integrated Development Environment	22
Object Designer tool icons	24
NAV object and system elements	25
NAV functional terminology	28
User interfaces	29
An introduction to development	31
Our scenario for development exercises	31
Getting started with application design	32
Application tables	32
Designing a simple table	32
Creating a simple table	33
Field numbering	34

Pages/Forms	36
Keyboard shortcuts	49
Run a table	50
Reports	50
Creating a List format report	52
Codeunits	58
MenuSuites	59
Dataports	59
XMLports	60
Integration tools	61
Backups and documentation	62
Summary	63
Review questions	64
Chapter 2: Tables	67
Overview of tables	67
Components of a table	68
Table naming	69
Table numbering	69
Table properties	70
Table triggers	72
Keys	74
SumIndexFields	77
Field Groups	78
Expanding our sample application	82
Creating and modifying tables	82
Assigning a TableRelation property	87
Creating Forms for testing	87
Adding Secondary keys	95
Adding some activity-tracking tables	96
New tables	98
Keys and SumIndexFields in our examples	103
Types of tables	107
Wholly modifiable tables	107
Master	107
Journal	108
Template	109
Ledger	110
Reference	112
Register	114
Posted Document	114
Setup	116
Temporary	117

Content-modifiable tables	117
System	117
Read-Only tables	119
Virtual	119
Summary	121
Review questions	122
Chapter 3: Data Types and Fields for Data Storage and Processing	125
Basic definitions	125
Fields	126
Field properties	126
Field numbering	132
Changing the data type of a field	134
Field triggers	135
Data structure examples	136
Variable naming	137
Data types	138
Fundamental data types	138
Numeric data	138
String data	139
Date/Time data	139
Complex data types	141
Data structure	141
Objects	142
Automation	142
Input/Output	142
DateFormula	143
References and other	149
Data type usage	150
FieldClass property options	152
Filtering	158
Defining filter syntax and values	158
Filtering on equality and inequality	159
Filtering by ranges	160
Filtering with Boolean operators	161
Filtering with wildcards	161
Filtering with combinations	162
Experimenting with filters	163
Accessing filter controls	172
Summary	175
Review questions	176

Chapter 4: Pages—Tools for Data Display	179
What is a page?	180
Controls	180
Bound and unbound	180
Pages—a stroll through the gallery	181
A sample RoleTailored Client page	182
Types of pages	187
List page	187
Card page	188
Document page	188
FastTab	189
List+ page	191
Journal/Worksheet page	191
Confirmation (Dialog) page	192
Request page	192
Navigate page	193
Departments page	194
Role Center page	196
Page parts	197
FactBoxes	198
Page names	199
Accessing the Page Designer	200
What makes up a page?	201
Page properties	202
Types of page controls	205
Inheritance	211
Page control details	211
Container controls	211
Group controls	212
Field controls	214
Using page controls in a Card page	218
Page Part controls	224
Creating a Card Part FactBox	226
Page Control triggers	228
Adding more List pages to our ICAN application	229
Creating a simple list page	230
Creating related List and Card pages	231
Learning more about pages	234
UX (User Experience) Guidelines	234
Creative plagiarism	235

Experimenting with page controls and control properties	235
Help searching	236
Experimentation	236
Testing	238
Design	238
Summary	239
Review questions	240
Chapter 5: Reports	243
What is a report?	244
Two NAV report designers	244
A hybrid report designer	247
NAV report—look and feel	247
NAV report types	248
Report types summarized	253
Report naming	253
Report components overview	254
The components of a report description	255
Report Data Flow	256
The elements of a report	259
Report properties	260
Report triggers	263
Data Items	264
Data item properties	264
Data item triggers	269
Data item Sections	270
Creating RTC reports via the Classic Report Wizard	271
Learn by experimentation	279
Runtime formatting	280
Inheritance	281
Other ways to create RTC reports	281
Modify an existing RTC report	282
The Visual Studio Report Designer layout screen	284
Report Items	286
Make the report changes	288
Request Page	300
Processing-Only reports	304
Creating a report from scratch	305
Creative report plagiarism	306
Summary	307
Review questions	308

Chapter 6: Introduction to C/SIDE and C/AL	311
Essential navigation	312
Object Designer	312
Starting a new object	313
Some designer navigation pointers	319
Exporting objects	320
Importing objects	322
Text objects	324
Object number licensing	324
Some useful practices	325
Changing data definitions	326
Saving and compiling	326
Some C/AL naming conventions	328
Variables	329
Global identifiers	329
Local identifiers	330
Special working storage variables	331
A definition of programming in C/SIDE	334
Functions	335
Basic C/AL syntax	343
Assignment and punctuation	343
Wildcards	344
Expressions	345
Operators	346
Some basic C/AL	350
MESSAGE, ERROR, CONFIRM, and STRMENU functions	350
MESSAGE function	350
ERROR function	351
CONFIRM function	352
STRMENU function	354
SETCURRENTKEY function	355
SETRANGE function	356
GET function	356
FIND	357
FIND ([Which]) options and the SQL Server alternates	358
BEGIN-END compound statement	360
IF-THEN-ELSE statement	361
Indenting code	362
Some simple coding modifications	363
Adding a validation to a table	363
Adding code to enhance a report	368
Summary	375
Review questions	376

Chapter 7: Intermediate C/AL	379
Some C/AL development tools	379
C/AL Symbol Menu	380
Internal documentation	381
Computation and Validation utility functions	384
TESTFIELD	384
FIELDERROR	385
VALIDATE	386
ROUND	387
TODAY, TIME, and CURRENTDATETIME functions	388
WORKDATE function	389
Data conversion functions	390
FORMAT function	390
EVALUATE function	391
DATE functions	391
DATE2DMY function	391
DATE2DWY function	392
DMY2DATE and DWY2DATE functions	392
CALCDATE function	393
FlowField-SumIndexField functions	394
CALCFIELDS function	395
CALCSUMS function	396
CALCFIELDS and CALCSUMS comparison	396
Flow control	397
REPEAT–UNTIL control structure	397
WHILE–DO control structure	397
CASE–ELSE statement	398
WITH–DO statement	400
QUIT, BREAK, EXIT, SKIP, and SHOWOUTPUT functions	401
QUIT function	401
BREAK function	402
EXIT function	402
SKIP function	402
SHOWOUTPUT function	403
Input and Output functions	403
NEXT function with FIND or FINDSET	403
INSERT function	404
MODIFY function	405
Rec and xRec	405
DELETE function	405
MODIFYALL function	406
DELETEALL function	406

Filtering	407
SETRANGE function	408
SETFILTER function	408
COPYFILTER and COPYFILTERS functions	409
GETFILTER and GETFILTERS functions	409
MARK function	409
CLEARMARKS function	410
MARKEDONLY function	410
RESET function	410
Filter Groups	410
InterObject communication	411
Communication via data	411
Communication through function parameters	411
Communication via object calls	412
Using the new knowledge	412
A development challenge for you	413
Creating more ICAN test data	413
Developing the Donor Recognition Status report	419
Summary	433
Review questions	434
Chapter 8: Advanced NAV Development Tools	437
NAV process flow	438
Data preparation	439
Transactions entry	439
Testing and Posting the Journal batch	440
Accessing the data	440
Ongoing maintenance	441
Role Center pages	441
Role Center structure	442
Role Center activities page	444
Cue Groups and Cues	445
Cue source table	446
Cue Group Actions	449
System Part	451
Page Part	452
Navigation Pane and Action Menus	455
Departments	461
MenuSuite levels	462
MenuSuite structure	462
MenuSuite development	463
MenuSuite transformation	466
Configuration and personalization	467

Creating new C/AL routines	468
Callable functions	469
Codeunit 358 – Date Filter-Calc	470
Codeunit 359 – Period Form Management	471
Codeunit 365 – Format Address	473
Codeunit 396 – NoSeriesManagement	474
Codeunit 397 – Mail	475
Codeunit 408 – Dimension Management	475
Codeunit 412 – Common Dialog Management	476
Sampling of function models to review	477
Codeunit 228 – Test Report-Print	478
Codeunit 229 – print documents	478
Other objects to review	479
Management codeunits	479
Documenting modifications	480
Multi-language system	481
Multi-currency system	482
Code analysis and debugging tools	483
Developer's Toolkit	483
Relations to Tables	484
Relations from Objects	486
Source Access	486
Where Used	486
Trying it out	488
Working in exported text code	491
Using Navigate	493
Testing with Navigate	493
The C/SIDE Debugger	497
The C/SIDE Code Coverage tool	498
Client Monitor	498
Debugging NAV in Visual Studio	499
Dialog function debugging techniques	500
Debugging with MESSAGE	500
Debugging with CONFIRM	500
Debugging with DIALOG	500
Debugging with text output	501
Debugging with ERROR	501
C/SIDE test driven development	502
Summary	504
Review questions	505

Chapter 9: Extend, Integrate, and Design—into the Future	507
Interfaces	507
XMLports	508
XMLport components	510
XMLport properties	510
XMLport triggers	514
XMLport data lines	514
XMLport line properties	520
Element or attribute	523
XMLport line triggers	524
XMLport Request Page	526
Advanced interface tools	527
Automation Controller	528
NAV Communication Component	528
Linked Server Data Sources	529
C/OCX	529
C/FRONT	529
NAV Application Server (NAS)	529
Client Add-ins	530
Client Add-in definition	530
Client Add-in construction	531
Client Add-in comments	534
Web services	534
Exposing a web service	536
Publishing a web service	537
Determining what was published	538
Customizing Help	542
NAV development projects	544
Knowledge is key	545
Different approaches for different scopes	545
Advantages of designing new functionality	545
Modifying an existing functional area	546
NAV development time planning	547
Data-focused design	547
Determining the data needs	548
Defining the needed data views	548
Designing the data tables	548
Designing the user data access interface	548
Designing the data validation	549
Data design review and revision	549

Designing the Posting processes	550
Designing the supporting processes	550
Double-check everything	550
Design for efficiency	551
Disk I/O	551
Locking	552
Design for updating	553
Customization project recommendations	554
One change at a time	554
Testing thoroughly	555
Plan for upgrading	559
Benefits of upgrading	560
Coding considerations	560
Careful naming	561
Good documentation	561
Low-impact coding	562
The upgrade process	563
Upgrade executables only	563
Full upgrade	564
Supporting material	565
Sure Step	565
RIM	565
Other reference material	566
Into the future...	568
Summary	569
Review questions	570
Answers	573
Chapter 1	573
Chapter 2	574
Chapter 3	574
Chapter 4	575
Chapter 5	575
Chapter 6	576
Chapter 7	576
Chapter 8	577
Chapter 9	577
Index	579

Preface

To exist is to change, to change is to mature, to mature is to go on creating oneself endlessly – Henri Bergson

By choosing to study C/AL and C/SIDE for NAV 2009, you are once again choosing to embrace change. The knowledge you gain here about these tools can be applied for your and others' benefit. The information in this book will shorten your learning curve on how to program for the NAV 2009 ERP system using the C/AL language, the C/SIDE integrated development environment, and all the new capabilities therein.

By embarking on the study of NAV and C/AL, you are joining a high-quality, worldwide group of experienced developers. There is a collegial community of C/AL developers on the Web who readily and frequently share their knowledge. There are formal and informal organizations of NAV-focused users, developers, and vendor firms both on the Web and in various geographic locations. The NAV product is one of the best on the market and it continues to grow and prosper. Welcome aboard and enjoy the journey.

A business history timeline

The current version of Microsoft Dynamics NAV is the result of much inspiration and hard work along with some good fortune and excellent management decision making over the last quarter century or so.

The beginning

Three college friends, Jesper Balser, Torben Wind, and Peter Bang, from Denmark Technical University (DTU) founded their computer software business in 1984 when they were in their early twenties. That business was Personal Computing & Consulting (PC & C) and its first product was called PC Plus.

Single user PC Plus

PC Plus was released in 1985 with a primary goal of ease of use. An early employee said its functional design was inspired by the combination of a manual ledger journal, an Epson FX 80 printer, and a Canon calculator. Incidentally, Peter Bang is the grandson of one of the founders of Bang & Olufsen, the manufacturer of home entertainment systems par excellence.

PC Plus was PC DOS-based, a single user system. PC Plus' design features included the following:

- An interface resembling the use of documents and calculators
- Online help
- Good exception handling
- Minimal computer resources required

The PC Plus product was marketed through dealers in Denmark and Norway.

Multi-user Navigator

In 1987, PC & C released a new product, the multi-user Navigator and a new corporate name, Navision. Navigator was quite a technological leap forward. It included:

- Client/Server technology
- Relational database
- Transaction-based processing
- Version management
- High-speed OLAP capabilities (SIFT technology)
- A screen painter tool
- A programmable report writer

In 1990, Navision was expanding its marketing and dealer recruitment efforts into Germany, Spain, and the United Kingdom. Moreover, in 1990, V3 of Navigator was released. Navigator V3 was still a character-based system, albeit a very sophisticated one. If you had an opportunity to study Navigator V3.x, you would instantly recognize the roots of today's NAV product. By this time, the product included:

- A design based on object-oriented concepts
- Integrated 4GL Table, Form, and Report Design tools (the IDE)
- Structured exception handling
- Built-in resource management
- The original programming language that became C/AL
- Function libraries
- The concept of regional or country-based localization

When Navigator V3.5 was released, it also included support for multiple platforms and multiple databases. Navigator V3.5 would run on both Unix and Windows NT networks. It supported Oracle and Informix databases as well as the one that was developed in-house.

At about this time, several major strategic efforts were initiated. On the technical side, the decision was made to develop a GUI-based product. The first prototype of Navision Financials (for Windows) was shown in 1992. At about the same time, a relationship was established that would take Navision into distribution in the United States. The initial release in the US in 1995 was V3.5 of the character-based product, rechristened Avista for US distribution.

Navision Financials for Windows

In 1995, Navision Financials V1.0 for Microsoft Windows was released. This product had many (but not all) of the features of Navigator V3.5. It was designed for complete look and feel compatibility with Windows 95. There was an effort to provide the ease of use and flexibility of development of Microsoft Access. The new Navision Financials was very compatible with Microsoft Office and was thus sold as "being familiar to any Office user". Like any V1.0 product, it was fairly quickly followed by a V1.1 that worked much better.

In the next few years, Navision continued to be improved and enhanced. Major new functionalities were added:

- Contact Relation Management (CRM)
- Manufacturing (ERP)
- Advanced Distribution (including Warehouse Management)

Various Microsoft certifications were obtained, providing muscle to the marketing efforts. Geographic and dealer base expansion continued apace. By 2000, according to the Navision Annual Report of that year, the product was represented by nearly 1,000 dealers (Navision Solution Centers) in 24 countries and used by 41,000 customers located in 108 countries.

Growth and mergers

In 2000, Navision Software A/S and its primary Danish competitor, Damgaard A/S, merged. Product development and new releases continued for the primary products of both original firms (Navision and Axapta). In 2002, the now much larger Navision Software, with all its products (Navision, Axapta, and the smaller, older C5 and XAL) was purchased by Microsoft, becoming part of the Microsoft Business Systems division along with the previously purchased Great Plains Software business and its several product lines. Since that time, one of the major challenges for Microsoft has been to meld these previously competitive businesses into a coherent whole. One aspect of that effort was to rename all the products as Dynamics software, with Navision being renamed to Dynamics NAV.

Fortunately for those who have been working with Navision, Microsoft has not only continued to invest in the product, but has increased the investment. This promises to be the case for the foreseeable future.

Continuous enhancement

As early as 2003, research began with the Dynamics NAV development team planning moves to further enhance NAV, taking advantage of various parts of the Microsoft product line. Goals were defined to increase integration with products such as Microsoft Office and Microsoft Outlook. Goals were also set to leverage the functional capabilities of Visual Studio and SQL Server, among others. All the while, there has been a determination not to lose the strengths and flexibility of the base product.

This was a massive change that required almost a complete rewrite of the underlying code, the foundation that's normally not visible to the outside world. To accomplish that while not destroying the basic user interface, the business application model, or the development environment, was a major effort. The first public views of this new version of the system, a year or two later, were not greeted with universal enthusiasm from the NAV technical community. But the Dynamics NAV development persevered and Microsoft continued supporting the investment, until NAV 2009 was released in late 2008. With the addition of Service Pack 1 in mid-2009, the biggest hurdles to the new technologies have been cleared. More new capabilities and features are yet to come, taking advantage of all these efforts.

The new product will take ever-increasing advantage of SQL Server technologies. Development will become more and more integrated with Visual Studio and be more and more .NET compliant. The product is becoming more open and, at the same time, more sophisticated, supporting features like Web Services access, integration of third-party controls, RDL reporting, and so on. In our industry, it would be appropriate to say *To survive is to change*. Change and survive are part of what Dynamics NAV does very well.

C/AL's roots

One of the first questions often asked by developers and development managers new to C/AL is *What other language is it like?* The proper response is "Pascal". If the questioner is not familiar with Pascal, the next best response would be "C" or "C#".

At the time the three founders of Navision were attending classes at Denmark Technical University (DTU), Pascal was in wide use as a preferred language not only in computer courses, but in other courses where computers were tools and software had to be written for data analysis. Some of the strengths of Pascal as a tool in an educational environment also served to make it a good model for Navision's business applications development.

Perhaps coincidentally (perhaps not) at DTU in this same time period, a Pascal compiler called Blue Label Pascal was developed by Anders Hejlsberg. That compiler became the basis for what was Borland's Turbo Pascal, which was the "every man's compiler" of the 1980s because of its low price. Anders went with his Pascal compiler to Borland. While he was there, Turbo Pascal morphed into the Delphi language and IDE tool set under his guidance.

Anders later left Borland and joined Microsoft, where he led the C# design team. Much of the NAV-related development at Microsoft is now being done in C#. So the Pascal-C/AL-DTU connection has come full circle, only now it appears to be C#-C/AL. Keeping it in the family, Anders' brother, Thomas Hejlsberg also works at Microsoft on NAV and AX at the campus in Copenhagen. Each in their own way, Anders and Thomas continue to make significant contributions to Dynamics NAV.

In a discussion about C/AL and C/SIDE, Michael Nielsen of Navision and Microsoft, who developed the original C/AL compiler, runtime, and IDE, said that the design criteria were to provide an environment that could be used without:

- Dealing with memory and other resource handling
- Thinking about exception handling and state
- Thinking about database transactions and rollbacks
- Knowing about set operations (SQL)
- Knowing about OLAP (SIFT)

Paraphrasing some of Michael's additional comments, the language and IDE design was to:

- Allow the developer to focus on design, not coding, but still allow flexibility
- Provide a syntax based on Pascal stripped of complexities, especially relating to memory management
- Provide a limited set of predefined object types, reducing the complexity and learning curve
- Implement database versioning for a consistent and reliable view of the database
- Make the developer and the end user more at home by borrowing a large number of concepts from Office, Windows, Access, and other Microsoft products

Michael is still working as part of the Microsoft team in Denmark on new capabilities for NAV; this is another example of how, once part of the NAV community, most of us want to stay part of that community.

What you should know

This book will not teach you programming from scratch, nor will it tutor you in business principles. To get the maximum out of this book, you should come prepared with some significant experience and knowledge. You will benefit most if you already have the following attributes:

- Experienced developer
- Know more than one programming language
- IDE experience
- Knowledgeable about business applications
- Good at self-directed study

If you have those attributes, then by careful reading and performance of the suggested exercises in this book, you should significantly reduce the time it will take you to become productive with C/AL and NAV. Those who don't have all these attributes, but want to learn about the development technology of Dynamics NAV, can still gain a great deal by studying Chapter 1 in detail and other chapters as the topics appear to apply to their situation.

This book's illustrations are from the W1 Cronus database V2009 SP1.

Hopefully this book will smooth the path to change and shine a little light on some of the challenges and the opportunities alike. Your task is to take advantage of this opportunity to learn, to change, and then use your new skills productively.

What this book covers

Chapter 1, *A Short Tour through NAV 2009*, covers basic definitions as they pertain to NAV and C/SIDE. In addition, an introduction to eight types of NAV objects, Page and Report Creation Wizards, and tools that we use to integrate NAV with external entities is provided. There is a brief discussion of how backups and documentation are handled in C/SIDE.

Chapter 2, *Tables*, focuses on the top level of NAV data structure: tables and their structures. You will work your way through hands-on creation of a number of tables in support of an example application. We will review most types of tables found in the out of the box NAV application.

In Chapter 3, *Data Types and Fields for Data Storage and Processing*, you will learn about the basic building blocks of NAV data structure, fields and their attributes, data fields that are available, and field structure elements (properties, triggers) for each type of field. This chapter covers the broad range of Data Type options as well as Field Classes. We will also discuss the concept of filtering and how it can be considered as you design your database structure.

In Chapter 4, *Pages – Tools for Data Display*, we will review different types of pages, work with some of these, and review all the controls that can be used in pages. You will learn to use the Page Wizard and have a good introduction to the Page Designer. You will expand your example system, creating a number of forms for data maintenance and inquiry.

In Chapter 5, *Reports*, we will learn about the structural and layout aspects of NAV Report objects using both the Classic Report Designer and the Visual Studio Report Designer. In addition, you will be experimenting with some of the tools and continue to expand your example application.

Chapter 6, *Introduction to C/SIDE and C/AL*, will help you learn about the general Object Designer Navigation as well as more specific Navision individual (Table, Form/Page, Report) Designers. This chapter also covers variables of various types created and controlled by the developer or by the system, basic C/AL syntax and some essential C/AL functions.

Chapter 7, *Intermediate C/AL*, covers a number of practical tools and topics regarding C/AL coding and development. You will learn about the C/AL Symbol Menu and how it assists in development. This chapter also discusses various Computation, Validation and Data Conversion functions, Dates, FlowFields and SIFT, Processing Flow Control, Input-Output, and Filtering functions.

In Chapter 8, *Advanced NAV Development Tools*, we will review some of most important elements of the Role Tailored User Experience, in particular Role Center Page construction. In addition, we will cover a number of tools and techniques aimed at making the life of a NAV developer easier and more efficient.

Chapter 9, *Extend, Integrate, and Design – into the Future*, covers a variety of interfaces, with special emphasis on XMLports and Web Services. It also discusses designing NAV modifications, creating a new functional area, or enhancing an existing functional area. Finally, this chapter provides tips for design efficiency, updating and upgrading the system, all with the goal of helping you to be a more productive NAV developer.

What you need for this book

You will need some basic tools, including at least the following:

1. A license and database that you can use for development experimentation. An ideal license is a full Developer's license. If the license only contains the Form/Page, Report, and Table Designer capabilities, you will still be able to do many of the exercises, but you will not have access to the inner workings of Form/Pages and Tables.
2. The best database for your development testing and study will be a copy of the NAV Cronus demo/test database, but you may want to have a copy of a production database at hand for examination as well. This book's illustrations are from the W1 Cronus database for V2009 SP1.

If you have access to other NAV manuals, training materials, websites, and experienced associates, those will obviously be of benefit as well. But they are not required for your time with this book to be a worthwhile investment.

Who this book is for

- The business applications software designer/developer who:
 - Wants to become productive in NAV C/SIDE-C/AL development as quickly as possible
 - Understands business applications and the associated software
 - Has significant programming experience
 - Has access to NAV including at least the Designer granules and a standard Cronus demo database
 - Is willing to do the exercises to get hands-on experience
- The Reseller manager or executive who wants a concise, in-depth view of NAV's development environment and tool set
- The technically knowledgeable manager or executive of a firm using NAV who is about to embark on a significant NAV enhancement project
- The technically knowledgeable manager or executive of a firm considering purchase of NAV as a highly customizable business applications platform
- The reader of this book:
 - Does not need to be expert in object-oriented programming
 - Does not need to have previous experience with NAV

Conventions


In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.


Code words in text are shown as follows: "Transactions are entered into a Journal table; data is preliminarily validated as it is entered, master and auxiliary data tables are referenced as appropriate."

A block of code is set as follows:

```
http://localhost:7047/DynamicsNAV/WS/Services
http://Isaac:7047/DynamicsNAV/WS/CRONUS_International_Ltd/Services
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking on the **Next** button moves you to the next screen".

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.


To send us general feedback, simply send an email to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or email suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

 **Downloading the example code for the book**
Visit http://www.packtpub.com/files/code/6521_Code.zip to directly download the example code.
The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or web site name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

A Short Tour through NAV 2009

The toughest thing about being a success is that you've got to keep on being a success – Irving Berlin

Microsoft Dynamics NAV (including the earlier Navision generation) has been a successful product line for over two decades. During the 2008-2009 fiscal year, Microsoft Dynamics NAV crossed the milestone of more than 1,250,000 installed users, a major achievement for any application software.

At the end of calendar 2008, Microsoft Dynamics NAV 2009 was released – a major new version of the product. While this new version contains the same business application functionality as the previous release (V5 Service Pack 1), it is based on a completely new infrastructure and presents a dramatically different face, the **Role Tailored Client**, to users. Our focus in this book is the NAV 2009 system, including the new three tier Role Tailored Client.

In this chapter, we will take a short tour through NAV 2009. Our path will be along the following trail:

- NAV 2009 from a functional point of view as an ERP system
- What's new in NAV 2009
- Definitions of terms as used in NAV
- The C/SIDE development environment and tools
- A development introduction to the various NAV object types
- Other useful NAV development information

Your goal in this chapter is to gain a reasonably complete, "big picture" understanding of NAV. When you complete this chapter, you should be able to communicate to a business manager or owner about the capabilities NAV can provide to help them manage their firm. This will also give you a context for what follows in this book.

A product as complex and flexible as NAV can be considered from several points of view. One can study the NAV application software package as a set of application functions designed to help a business manage information about operations and finances. One can also look at NAV as a stack of building blocks from which to extend or build applications – and the tools with which to do the construction.

In NAV 2009, which has two quite different user interface options available, one must consider how the user interface affects both the application design and the presentation to the user. This requirement overlaps both the application viewpoint and the construction viewpoint.

You should know the different object types that make up a NAV system and the purposes of each. You should also have at least a basic idea of the tools that are available to you, in order to enhance (small changes) or extend (big changes) an NAV system. In the case of NAV, the **Integrated Development Environment (IDE)** includes essentially all of the tools needed for NAV application development. Later in this book, we will discuss where the IDE can be supplemented.

Prior versions of NAV were two-tier systems. One of the tiers was the database server, the other tier was the client. As the traditional two-tier NAV Client (now referred to as the Classic Client) is still an integral part of the system, we will cover the aspects of where it must be used for development and support. All development and much of the system administration uses the Classic Client. So, even though our focus is on developing for the Role Tailored Client (aka "the RTC"), many of the images scattered throughout this book will be of Classic Client displays. In brief, the RTC is for users, and as a developer, you will generally use the Classic Client for your work.

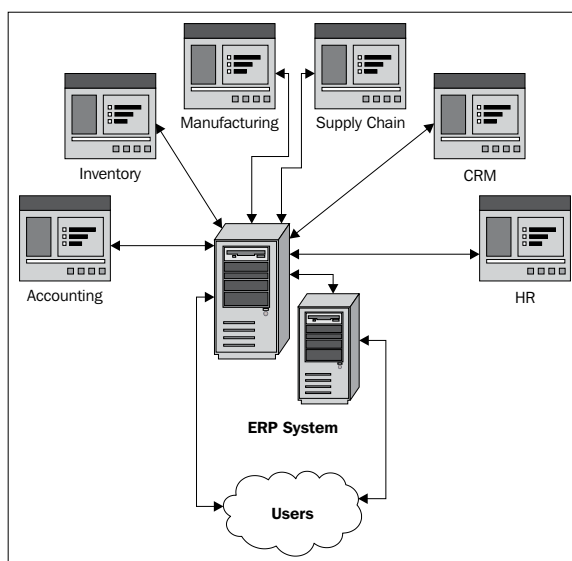
NAV 2009: An ERP system

If you look at NAV 2009 from the point of view of a firm using NAV to help run its business, you will see it as an integrated set of business applications software.

Microsoft Dynamics NAV is generally characterized as an ERP System. **ERP** stands for **Enterprise Resource Planning**. An ERP system is a set of integrated application software components designed to track and coordinate a wide variety of business activities, especially those involving products, orders, production and finances. An ERP system will typically include the following:

- Basic accounting functions (for example, general ledger, accounts payable, accounts receivable)
- Order processing and inventory (for example, sales orders, purchase orders, shipping, inventory, receiving)
- Relationship management (for example, vendors, customers, prospects, employees, contractors, and so on)
- Planning (for example MRP, sales forecasting, production forecasting)
- Other critical business areas (for example, manufacturing, warehouse management, fixed assets)

The integration of an ERP system is supported by a common database, by an "enter once, use everywhere" data philosophy, by a modular software design, and with data extraction and analysis tools. The following image is a view of an ERP system from the highest level:



The design of Microsoft Dynamics NAV addresses all the points in the above description and more. The NAV ERP system includes integrated modules covering the breadth of business functions for a wide range of business types. These modules share a common database and, where appropriate, share common data.

In the NAV system, there is a considerable overlap of components across application areas, with individual functions showing up in multiple different "modules". For example, in NAV, Inventory is identified as part of Financial management, but it is also, obviously, an integral part of Manufacturing, Supply Chain, and others.

The particular grouping of individual functions into modules that follows is based on Microsoft marketing materials. Some of these assignments are a bit arbitrary. What's important is for you to obtain a reasonable understanding of the overall set of application components that make up the NAV ERP system. In several of the following groupings, menu screenshots are included as examples. These are from the Role Tailored Client Departments menu screen.

Financial Management

Financial Management is the foundation of any ERP system. No matter what the business is, the money must be kept flowing, and the flow of money must be tracked. The tools which help to manage the capital resources of the business are included in NAV's Financial Management module. These include all or part of the following application functions:

- General Ledger – managing the overall finances of the firm
- Accounts receivable – tracking the incoming revenue
- Accounts payable – tracking the outgoing funds
- Analytical accounting – analyzing the various flows of funds
- Cash management and banking – managing the inventory of money
- Inventory and fixed assets – managing the inventories of goods and equipment
- Multi-Currency and Multi-Language – supporting international business activities



Manufacturing

NAV Manufacturing is general purpose enough to be appropriate for **Make to Stock (MTS)**, **Make to Order (MTO)**, and variations such as **Assemble to Order**, and so on. While off-the-shelf NAV is not particularly suitable for most process manufacturing and high-volume assembly line operations, there are third party add-on and add-in enhancements available for these. As with most of the NAV application functions, manufacturing can be installed in parts or as a whole, and can be used in a simplistic fashion or in a more sophisticated manner. NAV Manufacturing includes the following functions:

- Product design (BOMs and Routings) – managing the structure of product components and the flow of manufacturing processes
- Capacity and supply requirements planning – tracking the intangible and tangible manufacturing resources
- Production scheduling (infinite and finite), execution, and tracking – tracking the planned use manufacturing resources, both on an unconstrained and constrained basis



Supply Chain Management (SCM)

Obviously, some of the functions categorized as part of NAV Supply Chain Management (for example sales, purchasing, and so on) are actively used in almost every NAV implementation. As a whole, these constitute the base components of a system appropriate for a distribution operation. The Supply Chain applications in NAV include parts of the following applications:

- Sales order processing and pricing – supporting the heart of every business – entering, pricing, and processing sales orders
- Purchasing (including Requisitions) – planning, entering, pricing, and processing purchase orders
- Inventory management – managing inventories of goods and materials
- Warehouse management including receiving and shipping – managing the receipt, storage, retrieval, and shipment of material and goods in warehouses



Business intelligence and reporting

Although Microsoft marketing materials identify **Business Intelligence (BI)** and reporting as though it were a separate module within NAV, it's difficult to physically identify it as such. Most of the components that are used for BI and reporting purposes are (appropriately) scattered throughout various application areas. In the words of one Microsoft document, "Business Intelligence is a strategy, not a product." Functions within NAV that support a Business Intelligence strategy include the following:

- Standard Reports – distributed ready-to-use by end users
- Report wizards – tools to create simple reports or foundations for complex reports
- Account schedules and analysis reports – a very specialized report writer for General Ledger data
- Analysis by dimensions – a capability embedded in many of the other tools
- Interfaces into Microsoft Office including Excel – communications of data either into NAV or out of NAV
- SQL server reporting services compatible report viewer – provides the ability to present NAV data in a variety of textual and graphic formats, includes user interactive capabilities
- Interface capabilities such as Automation Controllers and web services – technologies to support interfaces between NAV 2009 and external software products
- NAV Business Analytics – an OLAP cube based data analysis tool option

Relationship Management (RM)

NAV's **Relationship Management (RM)** functionality is definitely the "little brother" (or, if you prefer, "little sister") to the fully featured standalone Microsoft CRM system. The big advantage of RM is its tight integration with NAV customer and sales data.

Also falling under the heading of Customer Relationship module is the NAV Service Management (SM) functionality. While the RM component shows up in the menu as part of sales and marketing, the SM component is identified as an independent function in the menu structure.

- (RM) Marketing campaigns – plan and manage promotions
- (RM) Customer activity tracking – analyze Customer orders
- (RM) To do lists – manage what's to be done and track what's been done

- (SM) Service contracts – support service business operations
- (SM) Labor and part consumption tracking – track the resources consumed by the service business
- (SM) Planning and dispatching – managing service calls

Human Resource management

NAV Human Resources (HR) is a small module, but relates to a critical component of the business, the people. Basic employee data can be stored and reported via the master table (in fact, one could use HR to manage data about individual contractors in addition to employees). A wide variety of individual employee attributes can be tracked by use of dimensions fields.

- Employee tracking – maintain basic employee description data
- Skills inventory – inventory of the capabilities of employees
- Absence tracking – maintain basic attendance information
- EEOC statistics – tracking government required employee attribute data

Project management

The NAV Project management module consists of the jobs functionality supported by the resources functionality. Projects can be short or long term. They can be external (that is billable) or internal. This module is often used by third parties as the base for vertical market add-ons (for example, for construction or job oriented manufacturing). This application area includes parts or all of the following functions:

- Budgeting and cost tracking – managing project finances
- Scheduling – planning project activities
- Resource requirements and usage tracking – managing people and equipment
- Project accounting – tracking the results

Significant changes in NAV 2009

Even though the NAV 2009 release doesn't contain any significant changes in the business application functionality, the changes in the infrastructure are major. We will discuss those briefly now and in detail in the later chapters.

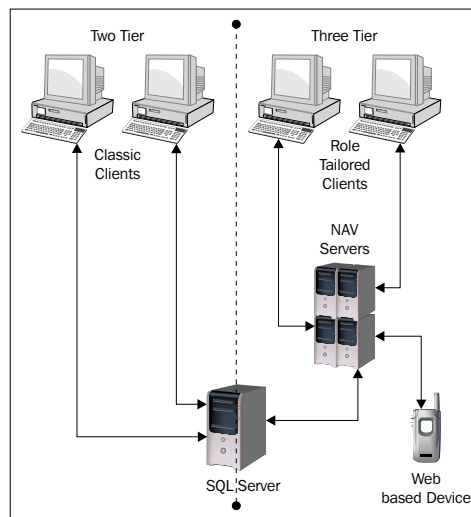
Two-tier versus three-tier

The two-tier (Classic Client) system has all the business logic in the client (first tier) and the database management in the database (second tier). The database can either be the Classic C/SIDE database or a SQL Server database. This is the mode of all NAV releases until Version 2009. The three-tier system, new with NAV 2009, has the Role Tailored Client as the first tier (minimal logic), the NAV Service Tier as the second tier (where all the business logic now resides), and the SQL Server database as the third tier.

Both the two-tier and the three-tier configurations are shown in the following image. In both cases, the database is SQL Server (a stand-alone two-tier configuration could have a Classic NAV database server). The server tier performs the same role in both instances. For the two-tier configuration, the Classic Clients handle authentication, object management, presentation/rendering, and the processing of all code, triggers, and validation.

For the three-tier configuration, the Role Tailored Clients are limited to some state tracking, simple data type validation and, of course, all the presentation and rendering for the new user interface. The NAV Service tier (NAV Servers) handles authentication, object management, the processing of all code, triggers and validation, plus offers web services support.

The following image illustrates a possible configuration of the NAV 2009 system. It has both the two-tier and three-tier options running in parallel. The three-tier option has multiple NAV Servers installed for capacity reasons (one NAV Server is projected to handle 40 to 60 users). Finally, the NAV Service tier is providing web services to support Internet-based access to the system from mobile devices.



Role Tailored Client

The new **Role Tailored Client (RTC)** is quite different in look and feel from the Classic Client. The RTC brings with it a completely new approach to designing and coding for the user interface. As the name implies, the orientation of the new client is to present a user interface that focuses on the specific role of the individual user. We will spend quite a bit of time studying the design and development for the new client.

SSRS-compatible report viewer

The new RTC supports reporting through the use of Visual Studio report designer and a Report Renderer/Viewer that replicates much of the functionality of SQL Server Reporting Services. RTC reports provide a host of new capabilities. RTC reports can be dynamically sorted in review (on-screen) mode, include graphics and expand (detail)/collapse (summary) displays, have drill-down and drill-through capabilities, and generate PDF or .xls (that is, Excel) files. We will spend quite a bit of time in Chapter 5, *Reports*, studying the Reporting Services for NAV 2009.

Web services

Web services is a new major feature of the NAV 2009 three-tier implementation. There isn't much to learn within NAV about web services because the implementation design is incredibly simple. Publishing a page or codeunit as a web service only requires a single record entry in the appropriate table.

Web services is very powerful, because it allows us to take advantage of the NAV Service Tier and expose NAV business logic, user authentication, and data access/validation to external processes. It is the new API for any application that has the ability to consume web services provided by another application.

NAV 2009: A set of building blocks and development tools

If you look at NAV 2009 from the point of view of a developer, you may see it as a set of customizable off-the-shelf program objects (the building blocks) plus the IDE which allows you to modify those objects and create new ones (the C/SIDE development tools).

The NAV 2009 system is an object-based system, consisting of several thousand application objects, the building blocks, made up of the eight different object types available in NAV. NAV does not have all of the features of an object-oriented system. A full-featured object-oriented system would allow the definition and creation of new object types, while NAV only allows for the creation and modification of the predefined object types.

NAV object types

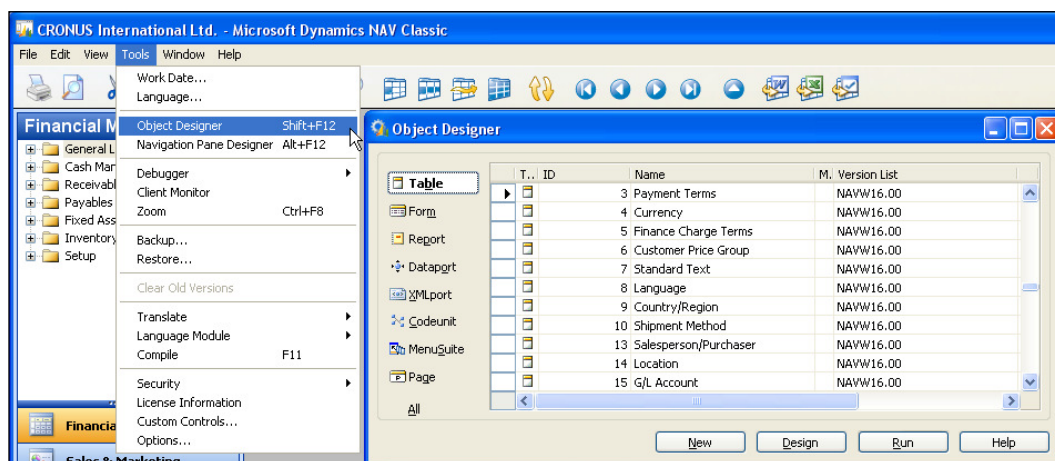
Let's start with some basic definitions of the object types that are part of NAV:

- **Table:** Tables are the definers and containers of data.
- **Form:** Forms are the screen display constructs for the Classic Client user interface.
- **Page:** Pages are the screen display constructs for the Role Tailored Client user interface. Pages are designed and rendered (displayed) using technology that is new to NAV 2009.
- **Report:** Reports allow the display of data to the user in "hardcopy" format, either onscreen (preview mode) or via a printer device. Report objects can also update data in processes with or without accompanying data display output.
- **Dataport:** Dataports allow the importing and exporting of data from/to external files in the Classic Client.
- **XMLport:** XMLports are similar to Dataports. In the Classic Client, XMLports are specific only to XML files and XML formatted data. In the Role Tailored Client, XMLports handle the tasks of both XMLports and Dataports.
- **Codeunit:** Codeunits are containers for code, always structured in code segments called functions.
- **MenuSuite:** MenuSuites contain menus which refer in turn to other types of objects. MenuSuites are structured differently from other objects, especially since they cannot contain any code or logic. In the Role Tailored Client, MenuSuites are translated into Navigation Pane menu entries.

The C/SIDE Integrated Development Environment

NAV includes a full set of software development tools. All NAV development tools are accessed through the C/SIDE Integrated Development Environment. This environment and its full complement of tools are generally just referred to as C/SIDE. C/SIDE includes the C/AL compiler. All NAV programming uses C/AL. No NAV development can be done without using C/SIDE.

The C/SIDE Integrated Development Environment is referred to as the **Object Designer** within NAV. It is accessed through the **Tools | Object Designer** menu option as shown in the following screenshot:



The language in which NAV is coded is **C/AL**. A sample of some C/AL code is shown as follows. C/AL syntax has considerable similarity to Pascal. As with any programming language, readability is enhanced by careful programmer attention to structure, logical variable naming, process flow consistent with that of the code in the base product and good documentation both inside and outside of the code.

```
IF "No." = '' THEN BEGIN
    SalesSetup.GET;
    SalesSetup.TESTFIELD("Customer Nos.");
    NoSeriesMgt.InitSeries(SalesSetup."Customer Nos.",xRec."No.
Series",0D,"No.", "No. Series");
END;
IF "Invoice Disc. Code" = '' THEN
    "Invoice Disc. Code" := "No.";










IF NOT InsertFromContact THEN
    UpdateContFromCust.OnInsert(Rec);

DimMgt.UpdateDefaultDim(
    DATABASE::Customer,"No.",
    "Global Dimension 1 Code","Global Dimension 2 Code");
```


A large portion of the NAV system is defined by tabular entries, properties, and references. This doesn't reduce the requirement for the developer to understand the code, but it does allow some very significant applications development work to be done on a more of a point-and-choose or fill-in-the-blank approach than the traditional "grind it out" approach to coding. As we may mention more than once in this book, NAV gives developers the ability to focus on design rather than code. Much of the time, the effort required to create a new NAV application or modification will be heavily weighted on the side of design time, rather than technical development time. This long term goal for systems development tools has always been true for NAV. As the tools mature, NAV development continues to be more and more heavily weighted to design time rather than coding time.

Object Designer tool icons

The following screenshot shows a list of Object Designer tool icons. These Object Designer **icons** are shown isolated in the screenshot and then described briefly in the following table. Some of these icons apply to actions that are only for objects which run in the Classic Client, some are for objects for both clients. Where appropriate, the terminology in these descriptions will be explained later in this book. Additional information is available in the C/SIDE **Help** files and the Microsoft NAV documentation.

			
Icon	Name	Shortcut	Description
	Find		Search for data in a field matching a search string
	Toolbox		Access Tools for Controls use for Classic Client objects
	Properties	Shift+F4	Show Properties list for highlighted object or control
	Color		Access the color property
	Font		Access the font property
	Field Menu		Access the menu of all the fields for the table associated with the object
	C/AL Symbol Menu	F5	Display the C/AL Symbol Menu
	C/AL Code	F9	Open the C/AL Code in edit mode

The following table lists the specific development tools used for each object type. Also, as shown in this table, some objects are limited to being used in the Classic Client, some are limited to being used in the RTC environment, some are used in both, and some are interpreted differently depending on the environment in which they are invoked.

Object Type	Design Tool	User Interface	Comments
Table	Table Designer	Classic and RTC	
Form	Form Designer	Classic	
Page	Page Designer	RTC	
Report	Report Designer	Classic (and RTC)	If a report is run from the RTC but doesn't have an RTC compatible layout, NAV will run it under a temporary instance of the Classic Client
	Report Designer (data definition) + Visual Studio (user interface)	RTC	
Dataport	Dataport Designer	Classic	
XMLPort	XMLport Designer	Classic and RTC	For the RTC, Dataport functionality is handled through appropriately defined XMLports
Codeunit	IDE code editor for the C/AL language	Classic and RTC	
MenuSuite	Navigation Pane Designer	Classic and RTC	Much of the navigation in the RTC is done via Role Center pages rather than menus

NAV object and system elements

Let's take a look at the following:

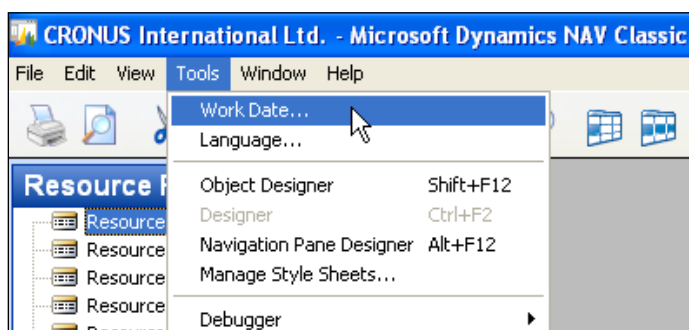
- Database:** NAV has two physical database options. One is the C/SIDE Database Server; the other is the Microsoft SQL Server. The C/SIDE Database Server, formerly known as the "Native" database, only supports the two-tier client. The NAV 2009 three-tier functionality is only compatible with a SQL Server implementation. You won't be surprised to know that one or two product versions in the future, the only database option will be SQL Server. At the basic application or code design levels, you don't care which database is being used. For sophisticated or high data volume applications you care a great deal about the underlying database strengths, weaknesses and features.

- **Properties:** These are the attributes of the element (for example object, data field, or control) that define some aspect of its behavior or use. For example, attributes such as display length, font type or size, and if the elements are either editable or viewable.
- **Fields:** These are the individual data items, defined either in a table or in the working storage of an object.
- **Records:** These are groups of fields (data items) that are handled as a unit in most Input/Output operations. The table data consists of rows of records with columns consisting of fields.
- **Controls:** These are containers for constants and data. A control corresponds to a User Interface element on a form/page or a report. The visible displays in reports and forms consist primarily of controls.
- **Triggers:** The generic definition is a mechanism that initiates (fires) an action when an event occurs and is communicated to the application object. A trigger is either empty or contains code that is executed when the associated event fires the trigger. Each object type has its own set of predefined triggers. The event trigger name begins with the word "On" such as OnInsert, OnOpenPage, and OnNextRecord. NAV triggers have some similarities to those in SQL, but they are not the same. NAV triggers are locations within the various objects where a developer can place comments or C/AL code.
- When you look at the C/AL code of an object in its Designer, the following have a physical resemblance to the NAV event based triggers:
 - **Documentation** which can contain comments only, no executable code. Every object type except MenuSuite has a single Documentation section at the beginning.
 - **Functions** which can be defined by the developer. They represent callable routines that can be accessed from other C/AL code either inside or outside the object where the called function resides. Many functions are provided as part of the standard product. As a developer, you may add your own custom functions as needed.
- **Object numbers and field numbers:** The object numbers from 1 (one) to 50,000 and in the 99,000,000 (99 million) range are reserved for use by NAV as part of the base product. Objects in this number range can be modified or deleted, but not created with a developer's license. Field numbers are often assigned in ranges matching the related object numbers (that is starting with 1 for fields relating to objects numbered 1 to 50,000, starting with 99,000,000 for fields in objects in the 99,000,000 and up number range). Object and field numbers from 50,001 to 99,999 are generally available to the rest of us for assignment as part of customizations developed in the field using

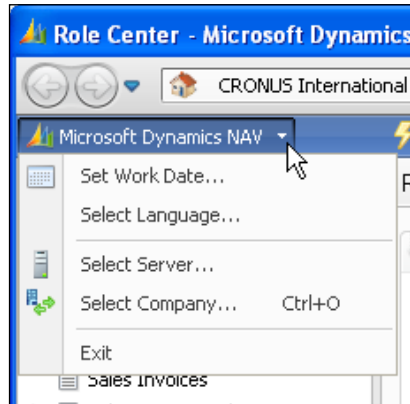
a normal development license. But object numbers from 90,000 to 99,999 should not be used for permanent objects as those numbers are often used in training materials. Microsoft allocates other ranges of object and field numbers to **Independent Software Vendor (ISV)** developers for their add-on enhancements. Some of these (in the 14,000,000 range in North America, other ranges for other geographic regions) can be accessed, modified, or deleted, but not created using a normal development license. Others (such as in the 37,000,000 range) can be executed, but not viewed or modified with a typical development license. The following table summarizes the content as:

Object Number range	Usage
1 – 9,999	Base-application objects
10,000 – 49,999	Country-specific objects
50,000 – 99,999	Customer-specific objects
100,000 – 98,999,999	Partner-created objects
Above 98,999,999	Microsoft territory

- **Work Date:** This is a date controlled by the operator that is used as the default date for many transaction entries. The System Date is the date recognized by Windows. The Work Date that can be adjusted at any time by the user, is specific to the workstation, and can be set to any point in the future or the past. This is very convenient for procedures such as ending Sales Order entry for one calendar day at the end of the first shift, and then entering Sales Orders by the second shift dated to the next calendar day. You set the Work Date in the Classic Client by selecting **Tools | Work Date** (see following screenshot).



In the Role Tailored Client, you can set the Work Date by selecting **Microsoft Dynamics NAV | Set Work Date**, and then entering a date (see following screenshot).



- **License:** A data file supplied by Microsoft that allows a specific level of access to specific object number ranges. NAV licenses are very clever constructs which allow distribution of a complete system, all objects, modules, and features (including development) while constraining exactly what is accessible and how it can be accessed. Each license feature has its price. Microsoft Partners have access to "full development" licenses to provide support and customization services for their clients. End-user firms can also purchase licenses allowing them developer access to NAV.

NAV functional terminology

For various application functions, NAV uses terminology that is more akin to accounting terms than to traditional data processing terminology. Some examples are:

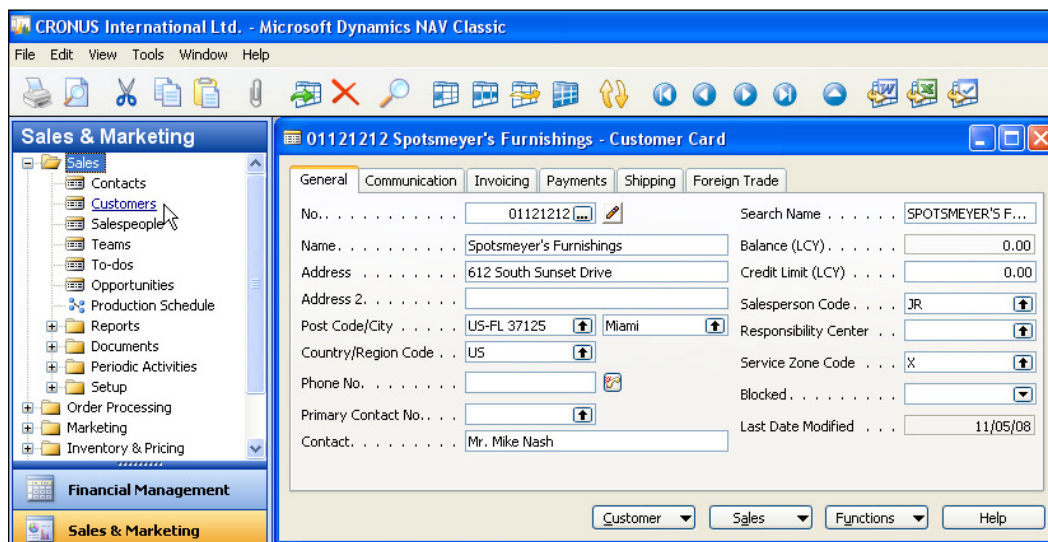
- **Journal:** A table of transaction entries, each of which represents an event, an entity, or an action to be processed. There are General Journals for general accounting entries, Item Journals for changes in inventory, and so on.
- **Ledger:** A detailed history of transaction entries that have been processed. For example, General Ledger, a Customer Ledger, a Vendor Ledger, an Item Ledger, and so on. Some Ledgers have subordinate detail ledgers, typically providing a greater level of date plus quantity and/or value detail.

- **Posting:** The process by which entries in a Journal are validated, and then entered into one or more Ledgers.
- **Batch:** A group of one or more Journal entries that were posted in one group.
- **Register:** An audit trail showing a history, by Entry No. ranges, of the Journal Batches that have been posted.
- **Document:** A formatted report such as an Invoice, a Purchase Order, or a Check, typically one page for each primary transaction.

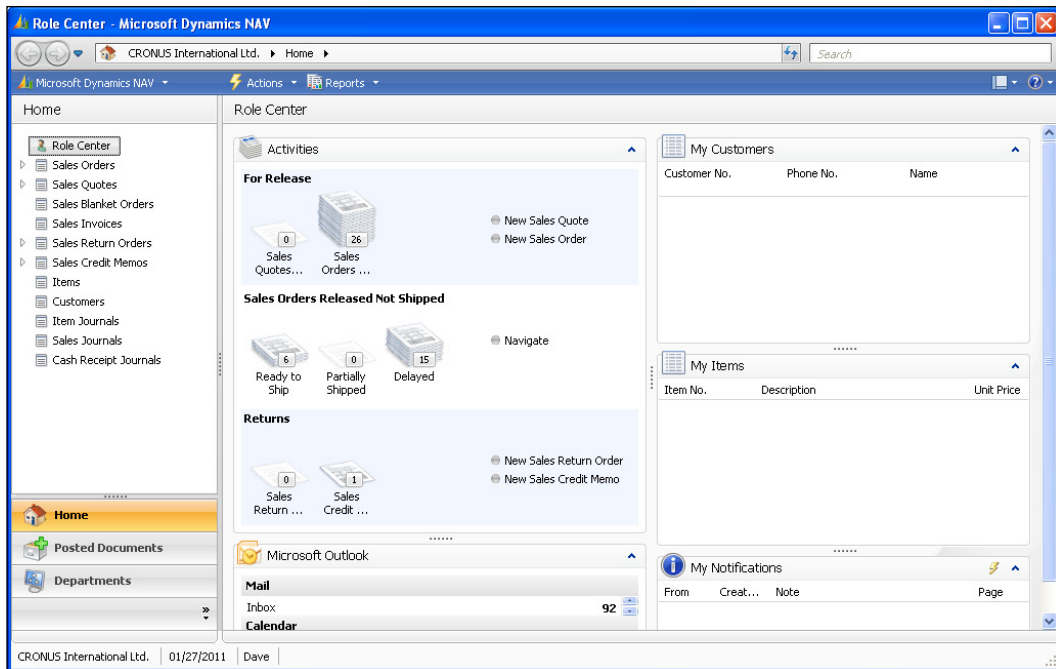
User interfaces

The terms two-tier versus three-tier and Classic Client versus Role Tailored Client have come up several times already in our discussion of NAV 2009. Let's initially focus on the user look and feel differences, and what that means to you when designing an application.

Following is a sample of what the Classic Client (two-tier) user interface looks like. Note that if the login were for a user with limited access privileges, only the permitted menu options would be displayed. Nevertheless, the basic structure of the display is oriented around the structure of the database and the traditional technician viewpoint of how the system works.



Now let's take a look at the appearance of the Role Tailored Client. The same comment applies about the system displaying only the permitted functions. However, the basic structure of the display here is oriented around a definition of the Role (and therefore the tasks) of the specific user who has logged in. Someone whose role centers around Order Entry will see a different RTC home page than the user whose Role centers around Invoicing, even though both are primarily focused in what we used to think of more globally as Sales & Receivables.

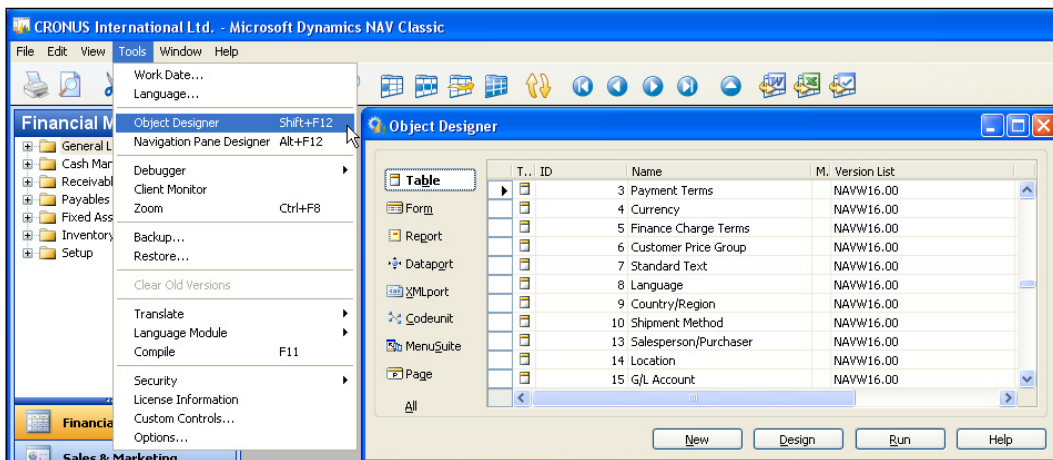


Obviously the user look and feel has changed dramatically from the Classic Client to the RTC. The design approach for our enhancements must follow the new RTC style. In some ways this will be a more challenging task, especially for those of us who are purely technical developers without much knowledge of the individual user's point of view.

In order to do a good job of fitting the system to a particular customer, we must have a good understanding of the duties performed by different roles within that customer's organization. This means we need more diagnostic effort at the frontend of our system design and implementation planning. Perhaps we should always have done that, but since our design model was based on how our product worked, rather than how the customer's operation worked, we could get away with doing less. No more. It's not within the scope of this book to discuss that diagnostic effort in any detail. Nevertheless, it is very important that it be done and done well. In this book, we will concentrate on how to address the requirements for Roles once they are defined.

An introduction to development

As part of our study of NAV development tools and techniques, we are going to do some exercises here and there. There's no better way to learn than to try out the tools in an environment where you can't break anything important. We're going to do development work for a simple system. As discussed earlier, we will access C/SIDE through the **Tools | Object Designer** menu option, as shown in the following screenshot:



Our scenario for development exercises

Our organization is a small, not-for-profit one. The name is **ICAN**, which stands for **International Community And Neighbors**. We collect materials, mostly food, from those who can afford to share and distribute it to those who have difficulty providing for themselves or their families. A large company has supplied us with a small computer network and a three-user Microsoft Dynamics NAV 2009 system.

We need to track the donations which come in. They may be items, money, or some type of in-kind services. If they are items, we will need to inventory them. We also need to track donors, volunteers, and the people on our "help list" (our clients). A variety of reports and inquiries will be needed. As with any application, new requirements are likely to arise as time goes by. And, in this case, since the purpose of our scenario is to act as a laboratory for experimenting with NAV development tasks, new requirements will arise so that we can do additional experiments.

Getting started with application design

Our design for the ICAN application will start with the initial design of a Donor table, a Donor Card Page, a Donor List Page, and a Donor List Report. Along the way, we will review the basics of each of these NAV object types.

Application tables

Table objects are the foundation of every NAV application. Every project should start by designing the tables. Tables contain the definitions of the data structures, the data relationships within and between the tables, as well as many of the data constraints and validations.

The coded logic in a table triggers not only provides the basic control for the insertion, modification, and deletion of records, but also embodies many of the business rules for an application. Such logic isn't just at the record level but also at the field level. When we embed a large part of the application data handling logic within the tables, it makes the application easier to develop, debug, support, modify, and upgrade.

Designing a simple table

Let's create a simple `Donor` table for our NAV Donor application. First, we will inspect existing definitions for tables containing name and address information. Basic inspection is done by clicking on the **Tables** button in the **Object Designer**, then highlighting the desired table and clicking the **Design** button. Good examples are the `Customer` table (table object 18) and the `Vendor` table (table object 23).

We see some field name and definition patterns that we will generally copy on the basis of "consistency is good". One exception to the copying will be the Primary Key field. In the other tables, that field is `No.`, but we will use `Donor ID`, just to make all our examples stand out from views of standard code.

The `Donor` table will contain the following data fields:

Field names	Definitions
Donor ID	20 character text (code)
Name	50 character text
Address	50 character text
Address 2	50 character text
City	30 character text
State/Province	10 character text
Post Code	20 character text (code)
Country/Region Code	10 character text (code)

In the preceding data field list, three of the fields are defined as Code fields, rather than Text fields. This is because these will be referenced by or will reference to other data tables. Using Code fields limits the contents to a subset of the ASCII character set, making it easier to constrain entries to a range of values. Code fields will be discussed in more detail in Chapter 3, *Data Types and Fields for Data Storage and Processing*.

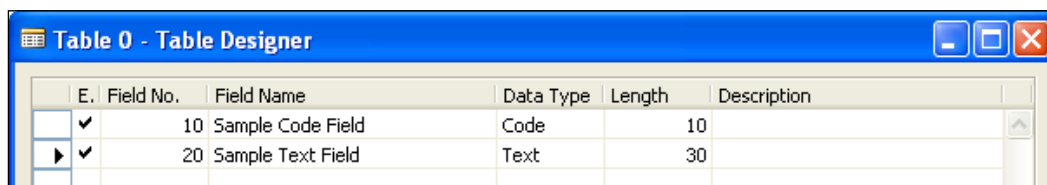
The **Donor ID** will be a unique identifier for our Donor record as it will also be referenced by other subordinate tables. The **Post Code** and **Country/Region Code** will reference other existing tables for validation. We choose the name, size, and data definition of these last two fields based on inspecting the equivalent field definitions in the Customer and Vendor tables.

We will have to design and define any referenced validation tables before we can eventually complete the definition of the Donor table. But our goal at the moment is just to get started.

Creating a simple table

Open the **Object Designer**, click on **Table** (in the left column of buttons) and click on **New** (in the bottom row of buttons). Enter the first field name (**Donor ID**) in the **Field Name** column and then enter the data type in the **Data Type** column. For those data types where length is appropriate, enter the maximum length in the **Length** column (for example text fields require lengths, date or numeric fields do not). Enter **Description** data as desired; these are only for display here as internal documentation.

As you can see in the following screenshot (and will have noticed already if you are following along in your system), when you enter a Text data type, the field length will default to 30 characters. This is simply an 'ease-of-use' default, which you should override as appropriate for your design. The 30 character **Text** default and 10 character **Code** default are used because this matches many standard application data fields of those data types.



E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>	10	Sample Code Field	Code	10	
<input checked="" type="checkbox"/>	20	Sample Text Field	Text	30	

Field numbering

The question often arises as to what field numbering scheme to use. Various systems follow a variety of standard practices, but in NAV, when you are creating a new table from scratch, it is a good idea to increment the **Field No.** by 10, as you can see in the preceding screenshot. The default increment for **Field No.** is 1. For a group of fields (such as an address block) where you are certain you will never add any intervening fields, you could leave the increment at 1, but there is no cost for using the larger increment.

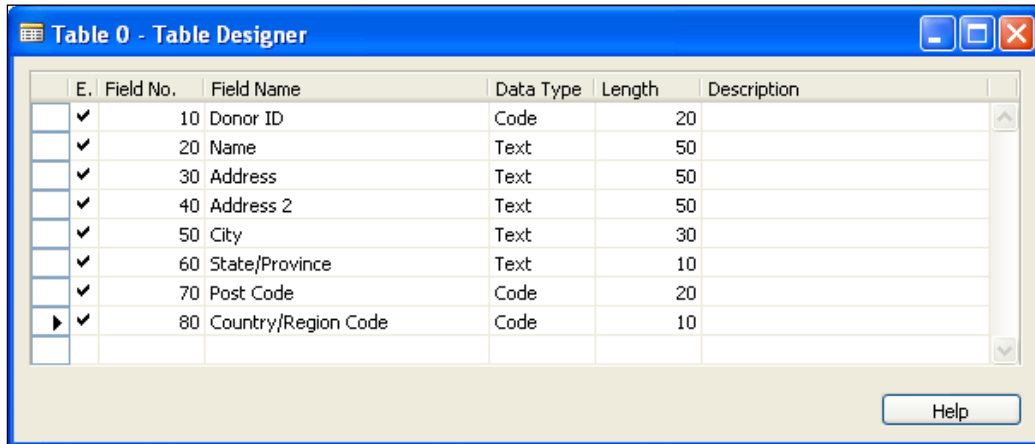
The numeric sequence of fields determines the default sequence in which data fields will display in a wide variety of situations, especially in the Classic Client. An example would be the order of the fields in any list presented to the user for setting up data filters in the Classic Client. This default sequence can only be changed by renumbering the fields. The compiler references each field by its **Field No.** not by its **Field Name**, so the renumbering of fields can be difficult once you have created other routines that reference back to these fields. At that point, it is generally better to simply add new fields where you can fit them without any renumbering.



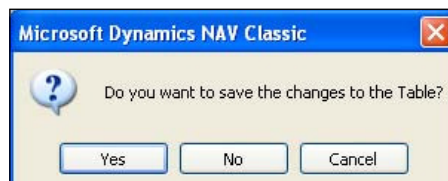
In fact, it can be irritatingly painful to renumber fields at any point after a table has been defined and saved. In addition to the field numbers controlling the sequence of presentation of fields, the field numbers control bulk data transfer (those transfers that operate at the record level rather than explicitly field to field transfer—for example the `TRANSFERFIELD` instruction). In a record-level transfer, data is transferred from each field in the source record to the field of the same number in the target record.

Obviously, it's a good idea to define an overall standard for field numbering as you start. Doing so makes it easier to plan your field numbering scheme for each table. Your design will be clearer for you and your user, if you are methodical about your design planning before you begin writing code (try to avoid the Ready-Fire-Aim school of system development). The increment of **Field No.** by 10 allows you to insert new fields in their logical sequence as the design matures. While it is not required to have the data fields appear in any particular order, it is frequently convenient for testing and often clarifies some of the user interactions.

When you have completed this first table, your definition should look like the following screenshot:

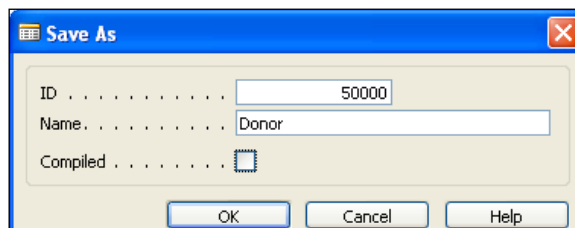


At this point, you can exit and save your Donor Table. The easiest way to do this is to simply press *Esc* until you are asked to save your changes.



When you respond by clicking on **Yes**, you will be asked for the object number and name you wish to assign. In a normal development situation, you will plan ahead what Object Number and descriptive Object Name you want to use.

In this case, we will use table Object No. 50000 and name it Donor. We are using 50000 as our Table Number just because it is the first (lowest) number available to us for a custom table through our Table Designer granule license.



Note that NAV likes to compile any object as it is saved, so the **Compiled** option is automatically checked. A compiled object is one that can be executed. If the object we were working on was not ready to be compiled without error, we could unselect the **Compiled** option in the **Save As** window, as shown in the preceding screenshot.



Be careful, as uncompiled objects will not be considered by C/SIDE when changes are made to other objects. Until you have compiled an object, it is a "work in progress", not an operable routine. There is a Compiled flag on every object that gives its compilation status. Even when you have compiled an object, you have not confirmed that all is well. You may have made changes that affect other objects which reference the modified object. As a matter of good work habit, recompile all objects before you end work for the day.

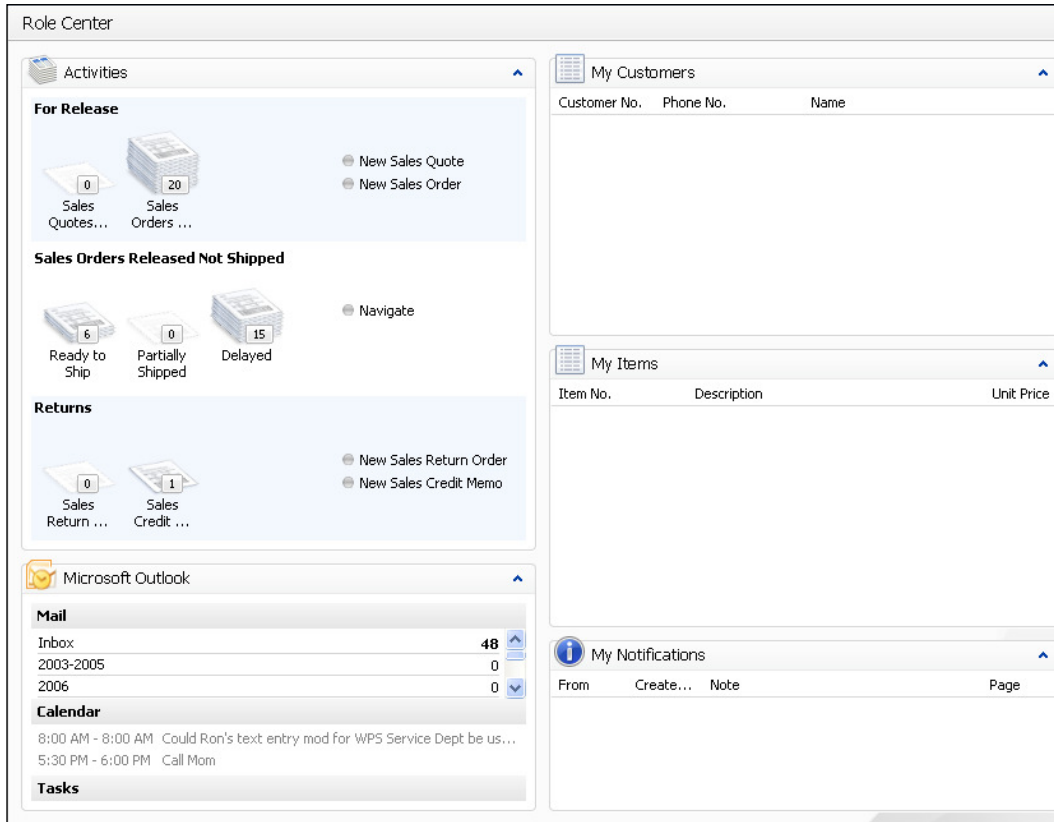
Pages/Forms

Pages are for the Role Tailored Client. Forms are for the Classic Client. If you are in the challenging position of maintaining a dual personality system (that is, running both clients), you will be forced to maintain equivalent versions of both Pages and Forms. We aren't going to address that situation in depth in this book as we are not going to cover the design and development of Forms in detail. In this book, we will focus on Pages and cover Forms just enough so that you can create a few to use as maintenance and debugging tools.



If you also need in-depth exposure to the world of developing for the Classic Client, you should refer to *Programming Microsoft Dynamics NAV*, Packt Publishing, (visit www.packtpub.com) about NAV development for the Classic Client. That book is based on NAV V5.0 SP1, but is fully applicable to NAV 2009 Classic Client development.

Pages fulfill several basic purposes. They provide views of data or processes designed for on-screen display only. They provide key points of user data entry into the system. They also act as containers for action items (menu options). The first page with which a user will come in contact is a Role Center Page. The Role Center Page provides the user with a view of work tasks to be done. The Role Center Page should be tailored to the job duties of each user, so there will be a variety of Role Center Page formats for any firm.



There are several basic types of display/entry pages in NAV:

- **List pages**
- **Card pages**
- **Document pages**
- **Journal/Worksheet pages**

There are also page parts (they look and program like a page, but don't stand alone) as well as user interfaces that appear as pages, but are not Page objects. The latter user interfaces are generated by various dialog functions.

List pages

List pages display a simple list of any number of records in a single table. The **Customer List** page in the following screenshot shows a subset of the data for each customer displayed. The Master record list shows fields intended to make it easy to find a specific entry. List pages/forms often do not allow entry or editing of the data. Journal/Worksheet pages look like List pages, but are intended for data entry.

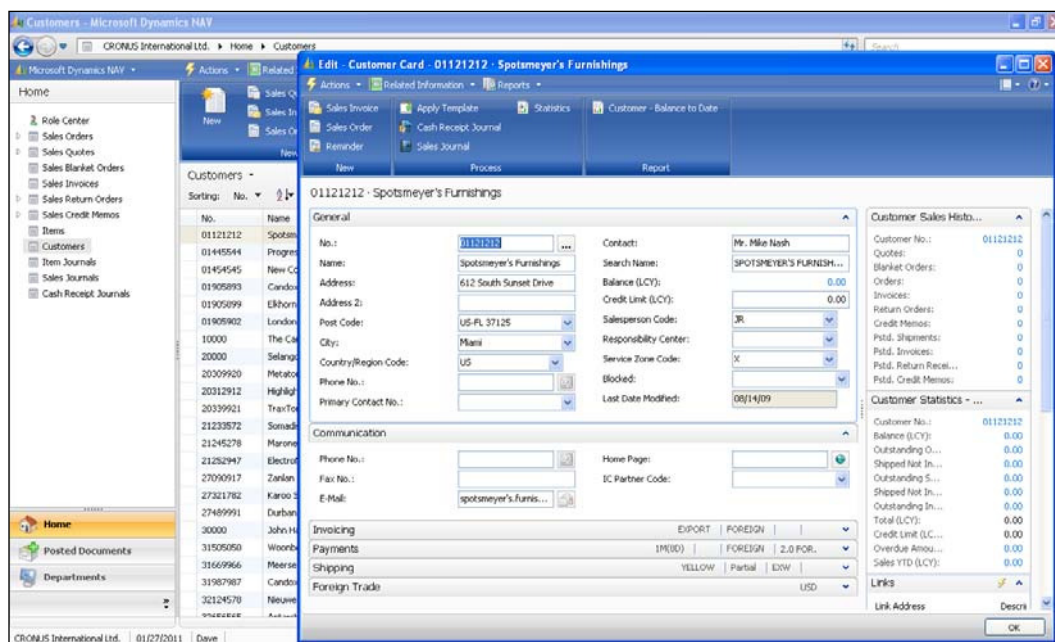
No.	Name	Responsibility	Location C...	Phone No.	Contact	Search Name
01121212	Spotsmeyer's Furnishings		YELLOW		Mr. Mike Nash	SPOTSMEY...
01445544	Progressive Home Furnishings		YELLOW		Mr. Scott Mitchell	PROGRESS...
01454545	New Concepts Furniture		YELLOW		Ms. Tammy L. McDonald	NEW CONC...
01905893	Candoxy Canada Inc.		YELLOW		Mr. Rob Young	CANDODY ...
01905899	Elkhorn Airport		YELLOW		Mr. Ryan Danner	ELKHORN ...
01905902	London Candoxy Storage Ca...		YELLOW		Mr. John Kane	LONDON C...
10000	The Cannon Group PLC	BIRMINGHAM	BLUE		Mr. Andy Teal	THE CANN...
20000	Selangorian Ltd.				Mr. Mark McArthur	SELANGOR...
20309920	Metatorad Malaysia Sdn Bhd		YELLOW		Mrs. Azleen Samat	METATORA...
20312912	Highlights Electronics Sdn Bhd		GREEN		Mr. Mark Darrell Bolland	HIGHLIGHT...
20339921	TraxTonic Sdn Bhd		YELLOW		Mrs. Rubina Usman	TRAXTONI...
21233572	Somadis		YELLOW		M. Syed ABBAS	SOMADIS
21245278	Maronegoce		BLUE		Mme. Fadoue AIT MOUSSA	MARONEG...
21252947	ElectroMAROC		YELLOW			ELECTROM...
27090917	Zanlan Corp.		YELLOW		Mr. Denik Stenerson	ZANLAN C...
27321782	Karoo Supermarkets		YELLOW		Mr. Pieter Wycoff	KAROO SU...
27489991	Durbandit Fruit Exporters		YELLOW		Mr. Eric Lang	DURBANDI...
30000	John Haddock Insurance Co.				Miss Patricia Doyle	JOHN HAD...
31505050	Woonboulevard Kuitenbrouwer		YELLOW		Maryann Barber	WOONBOU...
31669966	Meersen Meubelen		YELLOW		Michael Vanderhyde	MEERSEN ...
31987987	Candoxy Nederland BV		YELLOW		Rob Verhoff	CANDODY ...
32124578	Nieuwe Zandpoort NV		YELLOW		Kevin Verboort	NIEUWE Z...
32455555	Ambacore		YELLOW		Michael Zeman	AMTACORE

Card pages

Card pages display one record at a time. These are generally used for the entry or display of individual table records. Examples of frequently accessed card pages include Customer Card for customer data, Item Card for inventory items, and G/L Account Card for general ledger accounts.

Card pages often have FastTabs (multiple pages with each tab focusing on a different set of related customer data). FastTabs can be expanded or collapsed dynamically, allowing the specific data visible at any time to be controlled by the user.

Card pages for Master records display all the required data entry fields. Typically, they also display summary data about related activity so that the Card page can be used as the primary inquiry point for Master records. The following screenshot is a sample of a standard Customer Card:



Document pages

Another page style within NAV consists of what looks like a Card page plus a List page. An example is the **Sales Order** page as shown in the following screenshot. In this example, the upper portion of the page is in the style of a Card page with several tabs showing Sales Order data fields that have a single occurrence on the page (in other words, do not occur in a repeating column). The lower portion of the Document page is in the style of a List page showing a list of all the line items on the Sales Order (all fields are in repeating columns).

Line items may include product to be shipped, special charges, comments, and other pertinent order details. The information to the right of the data entry is related data and computations that have been retrieved and formatted. On top of the page, the information is for the Ordering customer and the bottom right contains information for the item on the selected line.

Type	No.	Description	Location C...	Quantity	Reserved Qua...	Unit of Mea...	Unit Price Excl...	Line Amount E...	Li
Item	LS-MAN-10	Manual for Loudspeakers	WHITE	4		PCS			
Item	LS-75	Loudspeaker, Cherry, 75W	WHITE	10		PCS	79.00	790.00	

Journal/Worksheet pages

Journal and Worksheet pages look very much like List pages. They display a list of records in the body of the page. Many have a section at the bottom that shows details about the selected line and/or totals for the displayed data. These pages may also include a Filter pane and perhaps a FactBox. The biggest difference between Journal/Worksheet pages and basic List pages is that Journal and Worksheet pages are designed to be used for data entry. An example of a Worksheet page, the Requisition Worksheet in Purchasing, is shown in the following screenshot. This Worksheet assists the user in determining and defining what purchases should be made.

Edit - Req. Worksheet - DEFAULT - Default Journal Batch

Actions | Related Information | Reports

Process: Calculate Plan, Carry Out Action Message, Order Tracking, Dimensions, Item Tracking Lines

Report: Inventory Availability, Inventory Purchase Orders, Status, Inventory - Availability Plan

Name: DEFAULT

Type	No.	Action Mes...	A...	Description	Location C...	Or
Item	LS-10PC	New	✓	Black	SILVER	
Item	LS-10PC	New	✓	Black	WHITE	
Item	LS-120	New	✓	Loudspeaker, Black, 120W	SILVER	
Item	LS-150	New	✓	Loudspeaker, Cherry, 15...	SILVER	
Item	LS-150	New	✓	Loudspeaker, Cherry, 15...	WHITE	
Item	LS-2	New	✓	Cables for Loudspeakers	WHITE	
Item	LS-2	New	✓	Cables for Loudspeakers	WHITE	

Description: Loudspeakers, ... Buy-from Vendor Name: Lewis Home Furniture

Item Details - Replenish...

Item No.: LS-10PC
 Replenishment System: Purchase
Purchase
 Vendor No.:
 Vendor Item No.:
Production
 Manufacturing Policy: Make-to-St...
 Routing No.:
 Production BOM No.:

OK

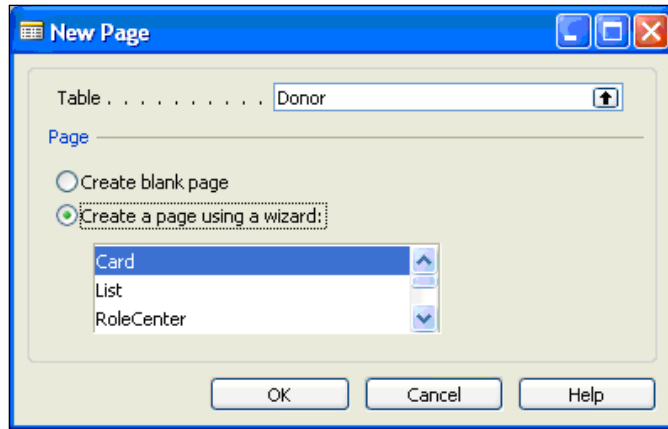
Standard elements of pages

A page consists of Page properties and triggers, Controls, Control properties and triggers. Data controls generally are either labels displaying constant text or graphics, or containers that display data or other controls. Controls can also be elements such as buttons, action items, and page parts. While there are a few instances where you must include C/AL code within page or page control triggers, in general it is good practice to minimize the amount of code embedded within pages. Most of the time, any data-related C/AL code can (and should) be located within the table object rather than the page object.

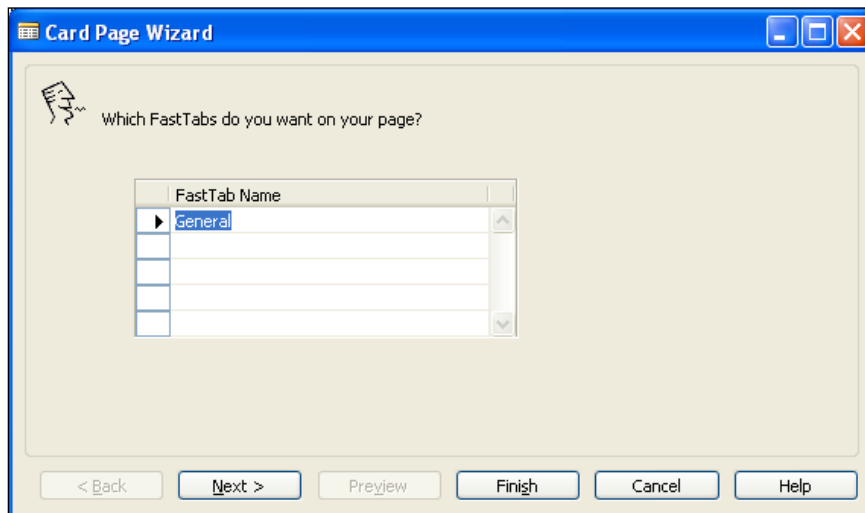
Creating a Card page

Let us try creating a Card page and a List page for the table we created earlier. The NAV IDE contains object generation tools (Wizards) to help you create relatively fully formed Classic Client forms and reports. Creating pages and reports for the Role Tailored Client is less about Wizards and more about actually using the Designer tools, but the Wizards are still very useful.

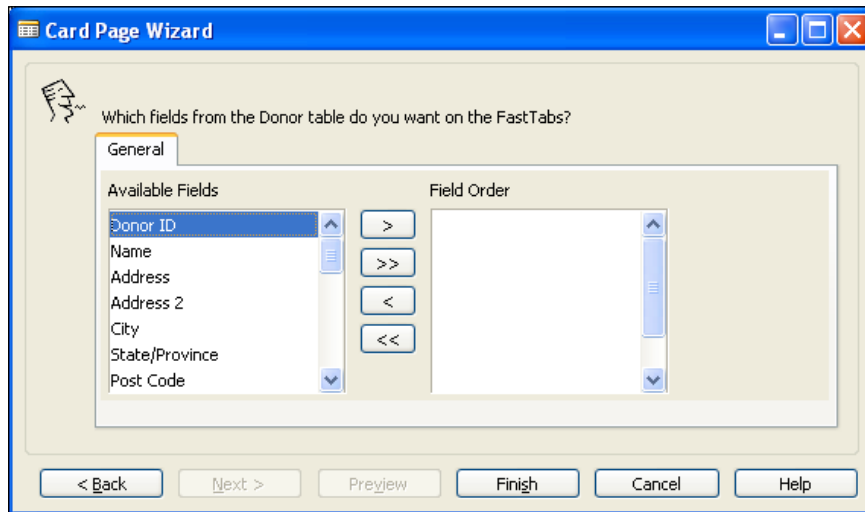
Open the **Object Designer**, click on **Page** and then click on **New**. The Page Wizard's screen will appear. Enter the name (**Donor**) or number (50000) of the **Table** with which the page is to be associated (bound). Choose the option **Create a page using a wizard**. Choose **Card** as shown in the following screenshot. Click on **OK**.



Next, you will encounter the following Card Page Wizard screen:



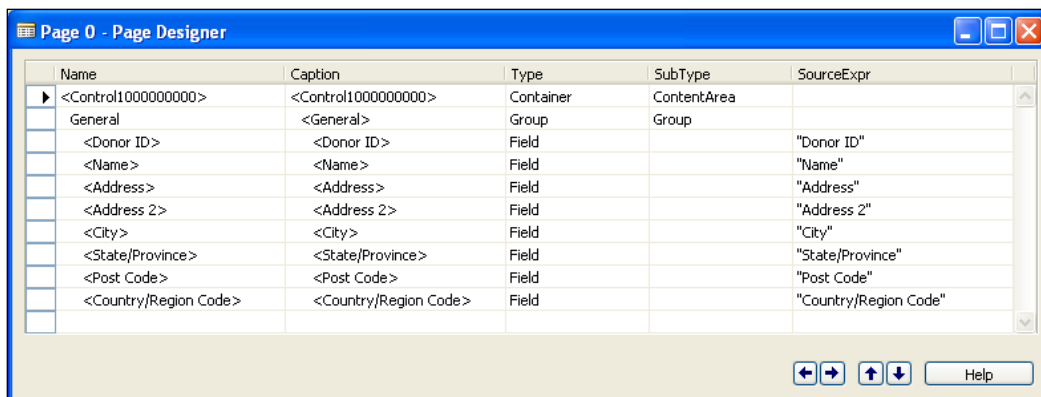
As this is our first Page creation effort, we will keep it simple. We will accept the single suggested FastTab of `General` and click on **Next** to proceed. That will take us to the following screen.



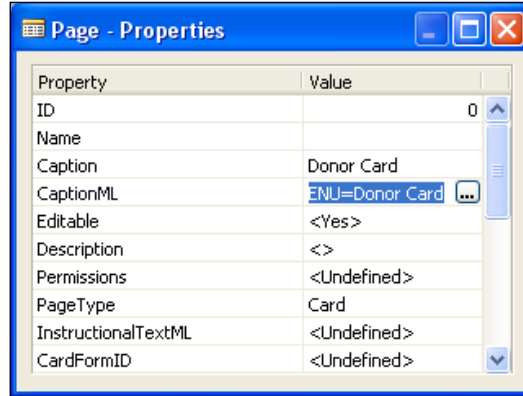
Now it's time to add all the data fields for our Donor table to the page. Click on the button shown in the following image. That will move all of the **Available Fields** to the **Field Order** window.



Now click on the **Finish** button. The Page wizard will close and you will be looking at the Page Designer form showing the generated page code, as shown in the following screenshot.



Place your cursor on the first blank line at the bottom of the list of control definitions. Right-click and select **Properties** to open the **Page-Properties** screen as shown below (you could also press *Shift + F4* or click on the Properties icon at the top of the screen). Type the page name to be displayed "**Donor Card**" into the **Caption** field, it will default into the **MLCaption** field too, as follows:



Escape from that screen, and *Escape* from the Page Designer screen.

A **Save Changes** form will be displayed, just like the one we saw when we finished defining our Donor table. We have the same task now as we did then, to assign an ID (object number) and Name. Let's make this Page number 50000 and call it Donor Card. Leave the Compiled checkbox checked and click on **OK** so that the Page will be compiled (and error checked). Assuming you haven't received any error messages, you have now created your first NAV 2009 page object. Congratulations!

If you are doing your work with the original release of NAV 2009, you have to test a new card page by running it from the system Run option. From the **Start** menu, select **Run**. In the command box, enter the following:

```
Dynamicsnav:///runpage?page=50000
```

If you are using NAV 2009 SP1 or later, you can test your page by simply highlighting it and clicking on the **Run** button at the bottom of the Object Designer screen (the **Run** button is programmed to automatically submit that long command to the system).

This will invoke the Role Tailored Client, open it up, and call up your new page 50000. You should see a screen that looks like this.

The screenshot shows a window titled "View - Donor Card" with a blue header bar. Below the header is a toolbar with an "Actions" dropdown menu. The main content area is titled "Donor Card" and contains a "General" tab. The form has two columns of input fields: Donor ID, Name, Address, and Address 2 on the left; and City, State/Province, Post Code, and Country/Region Code on the right. A "Close" button is located at the bottom right of the window.

Experiment a little. Expand the card page to full screen. Enter some Donor data for two or three donors (hint: click on the Actions icon at the top of the page and choose the **New** option). You might want to open the page with the Page Designer using the Design button, add one of the data fields so it will appear a second time on the card. This time, change the Caption field for that data item. Save and compile. Run a test. Before you quit experimenting, go back and remove any extras that you had added, so that later your Donor Card will be what we created initially.

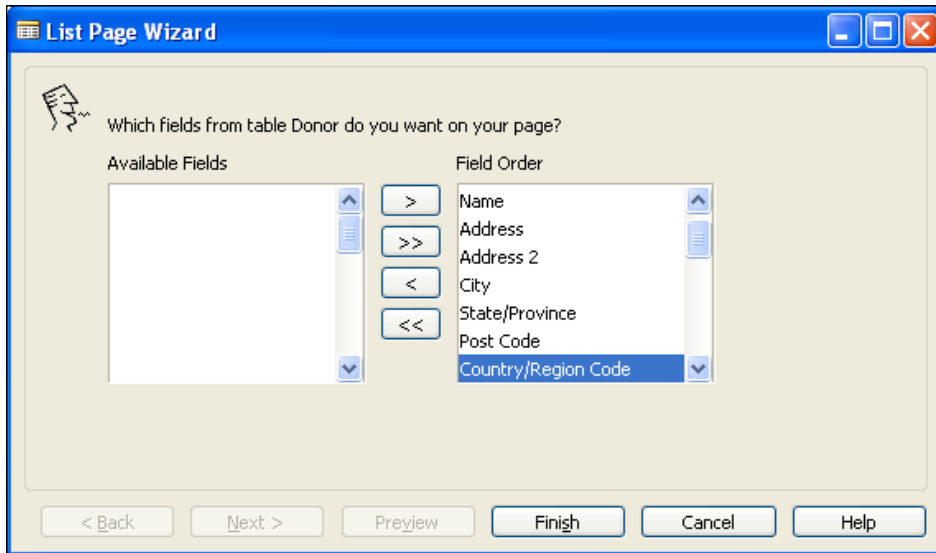
You could create a second copy of the page so that you could experiment, then discard it later. First, open Page 50000 in **Design** mode. Then click on **File | Save As** and save the page object as Page number 50090 with a different name such as Test Donor Card. Compile it, then make whatever changes you want, deleting the experimental page when you are done experimenting.

When you begin testing, you will find that a lot of features have been created automatically as part of the Card page, but you are still very limited in what you can do. For example, without a properly set up List Page, it's harder to move around in your table and quickly find different Donors, especially if there are more than just a few records in the table. This difficulty would be even more significant for your users. That's why List pages are the primary data navigation interface in the application system. So let's create a List page for our Donor table.

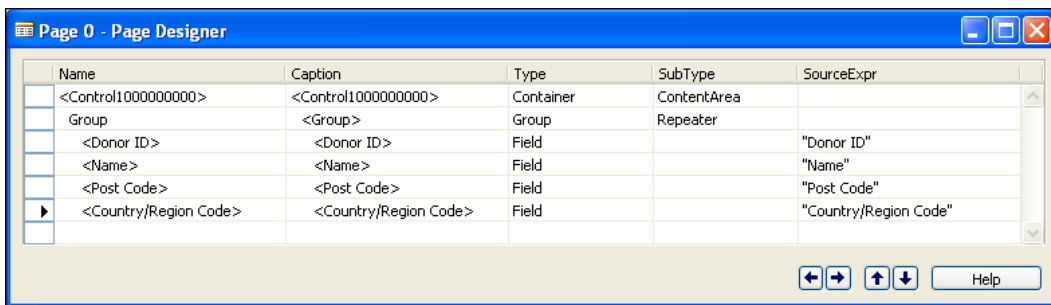
Creating a List page

The process of creating a Donor List page is essentially the same as the one we just went through for the Donor Card page. Open the **Object Designer**, click on **Page**, and then click on **New**. The Page Wizard's first screen will appear. Enter the name (Donor) or number (50000) of the table with which you want the page to be associated. Choose the option **Create a page using a wizard**. Choose **List** and click on **OK**.

You will see a List Page Wizard form that looks just like the second screen you encountered when you were creating a Card page. Unlike the Card page, we will use the single field selection button (>) to choose just the four individual fields we want to appear in our List page. The image below shows which fields to choose.



Now click on the **Finish** button. The Page wizard will close and you will be looking at the Page Designer form showing the generated material.



Place your cursor on the first blank line at the bottom of the list of fields, right-click, then select **Properties**. Type the page name to be displayed "**Donor List**" in the **Caption** field, it will default into the **MLCaption** field too. In that same property list, in the **CardFormID** field, enter the name (or ID number) of the Card page that should be used to display any entry that will be displayed on this List page. In this case, that will be **Donor Card**.

After you've made your property entries, the Page Properties form will look like the following screenshot:

Property	Value
ID	0
Name	
Caption	Donor List
CaptionML	ENU=Donor List
Editable	<Yes>
Description	<>
Permissions	<Undefined>
PageType	List
InstructionalTextML	<Undefined>
CardFormID	Donor Card
DataCaptionExpr	<Undefined>
SourceTable	Donor

In the Classic Client, the Card form is the first point of contact with the information in a table. When you invoke the List form from the Card form, a property on the table determines what List form is activated. In the RTC, the List page is the first point of contact with the information in a table. When you invoke the Card page from the List page, the CardFormID property determines what Card page to activate.

We can always return to the created page in the Page Designer and easily add fields we left off or remove something we decide is not needed. In addition, NAV list pages include a feature that allows you to have some field columns with the **Visible** property set to **FALSE** initially. This property is field specific, controls whether the column for a data field displays on screen or not, and can be personalized by users.

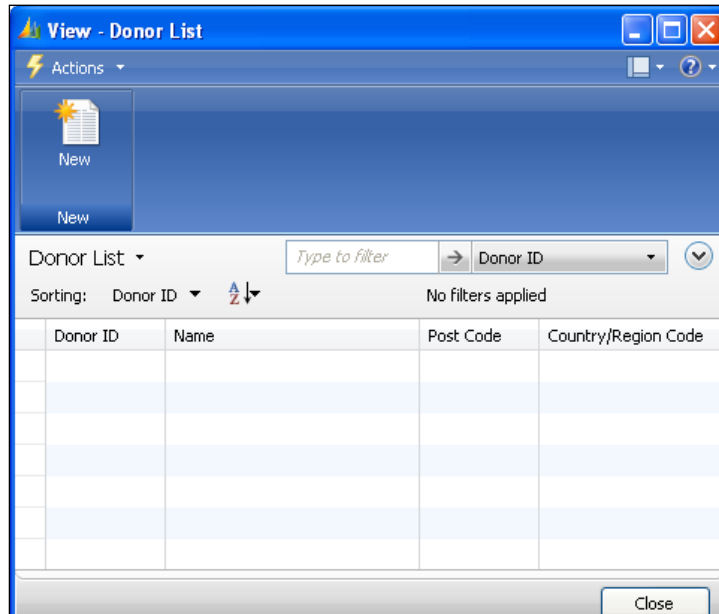
In order to save and compile the new page, just press *Esc* key and respond with **Yes** to the Save Changes form. Enter the Page number (ID) you want to assign (use 50001) and name (use Donor List). If you reused the Page object number 50000, you would have overwritten the Card page you created earlier.

ID	50001
Name.	Donor List
Compiled	<input checked="" type="checkbox"/>

OK Cancel Help

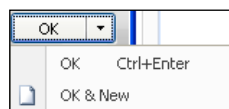
At this point, we have a data structure (Table 50000 – Donor), a page to enter and maintain data, (Page 50000 – Donor Card) and a page to display or inquire into a list of data (Page 50001 – Donor List). Let us use our Donor List and Donor Card to enter some data into our table, edit it, and review it.

In a full application, we would be accessing our pages from a Role Center page. But for now, we will just test our pages directly through the Run option. This will invoke the Role Tailored Client, open it up and call up page 50001. You should see a screen that looks like this:



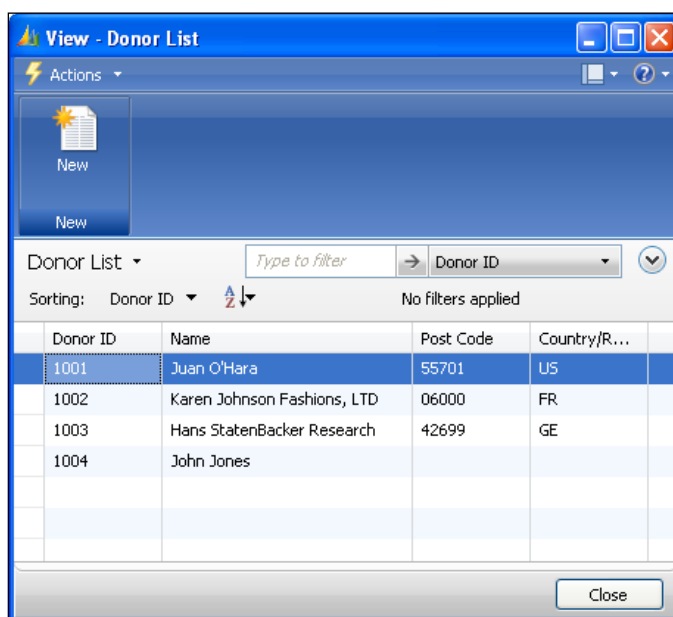
Notice the **New** command icon that has been included on the Action pane for your new page. That was provided automatically, without you needing to code it. Very convenient. Click on **New**. Your Donor Card page will open up in Add mode. Now is a good time to create some elementary test data. Enter the data for a Donor. Click on **OK**. Your new Donor record is displayed in the Donor List page. Hopefully you're having a good time, after all, success is enjoyable, even in small amounts.

Click on **New** again. Enter another Donor test record. This time, instead of clicking on the **OK** button, click on the option symbol (the inverted triangle) at the right of the button. You will see that you have two entry completion options.



If you simply click on **OK** or you press *Ctrl + Enter*, the control will return to the list page. But if you click on the **OK & New** option, the record just entered will be saved and you will return to an empty New Card page ready for the next record to be entered. Obviously, this second option is handy for situations requiring repetitive data entry.

After you have entered two or three Donor entries, go back to the Donor List page. Highlight one of the Donor entries, then double-click it. You will end up back in the Donor Card page in Edit mode with the highlighted Donor record loaded, ready to be changed. Change it and then click on **OK**. You return to the Donor List page with the Donor record updated with your changes similar to what's shown in the following screenshot:



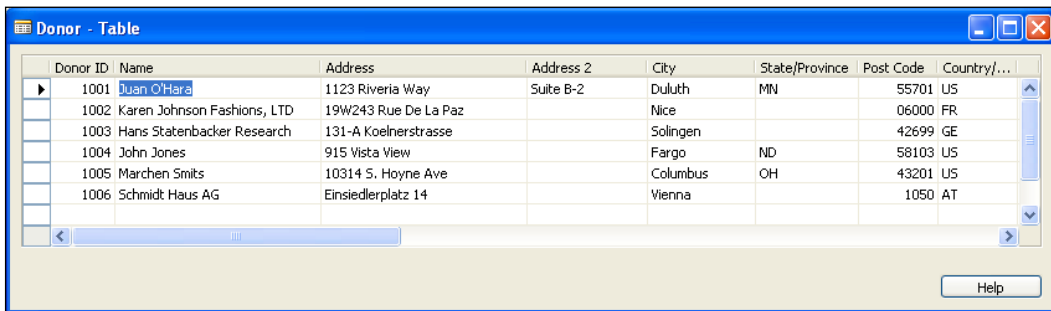
Keyboard shortcuts

Before we move on to creating a simple report, let's take a quick look at the list of keyboard shortcuts that are available in NAV 2009. From the RTC, call up Microsoft Dynamics Help. Use Search to find **Keyboard Shortcuts**. This will give you a list of the many keyboard shortcuts that are available within the Role Tailored Client.

Change focus to the Classic Client, where you've been doing your development work. Access the **Help** menu option in the header. Choose **Overview of F Keys**. This will display a list of the common keyboard shortcuts used within the Classic Client. You will see that this list of shortcuts is totally different from the one in the RTC (and much shorter). This is because there was a Microsoft product management decision to have NAV keyboard shortcuts more like those in Microsoft Excel. You have the challenge of remembering two sets of shortcuts in the same system and using the right ones at the right time.

Run a table

Hopefully, you are beginning to get a feeling for the ease of use and power of C/SIDE and NAV. Let's return to the Object Designer in the Classic Client. You've been viewing the Donors through the normal pages in the same manner as would a user, through the application Pages. As you have access to the Object Designer, you can also inspect, even manipulate data directly within the table. Simply access the Table Designer again, highlight the Donor table (50000) and **Run** the table.



Donor ID	Name	Address	Address 2	City	State/Province	Post Code	Country/...
1001	Juan O'Hara	1123 Riveria Way	Suite B-2	Duluth	MN	55701	US
1002	Karen Johnson Fashions, LTD	19W243 Rue De La Paz		Nice		06000	FR
1003	Hans Statenbacker Research	131-A Koelnerstrasse		Solingen		42699	GE
1004	John Jones	915 Vista View		Fargo	ND	58103	US
1005	Marchen Smits	10314 S. Hoyne Ave		Columbus	OH	43201	US
1006	Schmidt Haus AG	Einsiedlerplatz 14		Vienna		1050	AT

Running a table creates a default Classic Client form that includes all the displayable fields in the table. Now you can see the same Donor records you just entered. Dependent on the specific design of the table (about which you will learn more in subsequent chapters), you can freely modify the contents of the table.

Reports

Report objects can be used for several purposes, sometimes more than one purpose at a time. One purpose is the display of data. This can be on-screen (called Preview mode) or output to a printer device driver (Print mode). The displayed data may be read from the database, or may be the result of significant processing. As you would expect, a report may be extracting data from a single table (for example, a list of customers) or a combination of several tables (for example, a sales analysis report).

Other report objects have the purpose of only processing data, without any display formatted output of the processed results. Typically, such reports are referred to as batch processing reports. Reports can also be used as containers for program logic to be called from other objects or executed as an intermediate step in some sequence of processes. Normally, this task would be handled by Codeunit objects, but you can also use report objects if you want to.

Report formats are limited to a combination of your imagination, your budget, and the capabilities of NAV reporting. NAV reporting is very powerful and flexible in terms of what data can be reported, plus the various types of filtering and combining can be done. In the Classic Client, NAV reporting is relatively limited in terms of formatting. However, reports designed to be run in the Role Tailored Client use RDLC capabilities similar to those available from SQL Server Reporting Services. Consequently, they have a great deal of flexibility.

One standard way to create a new Role Tailored Client report is to create a report for the Classic Client, and then "transform" it (an embedded process that converts an object for the RTC) and then reformat it for the RTC. It may seem a roundabout method, but, depending on the report, it's sometimes the easiest approach. If it so happens that you prefer to only create the report in the Classic Client format, the RTC will run that one too, but without any of the new RTC report features.

Common report formats in NAV (both clients) include document style (for example Invoices or Purchase Orders), list style (for example, Customer List, Inventory Item List, Aged Accounts Receivable), and label format style (for example, a page of name and address labels). There is only one Report object. It can contain formatting layout information for only the Classic Client or for both the Classic and Role Tailored Clients. From a practical point of view, if you are going to run a report only from the RTC, the Classic Client format information can be very minimal (that is not particularly useful for Classic Client reporting).

A significant aspect of the NAV report object is the built-in "read-then-process" looping structure, which automates the sequence of "read a record, process the record, then read the next record". When manually creating or enhancing reports, you can define one data structure as subordinate to another, thus creating nested read-then-process loops. This type of predefined structure has its good points and bad points. The good points usually relate to how easy it is to do the kind of things it is designed for and the bad points relate to how hard it is to do something that the structure doesn't anticipate. We will cover both sides of discussion when we cover Reports in detail in Chapter 5, *Reports*.

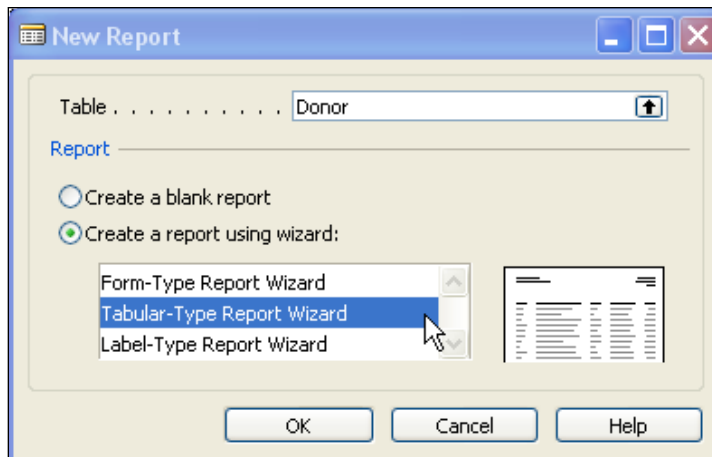
We've taken a quick look at normal RTC data entry and viewing and we've looked at how the data is defined in the table. We will now look at how that data might be extracted and reported.

Creating a List format report

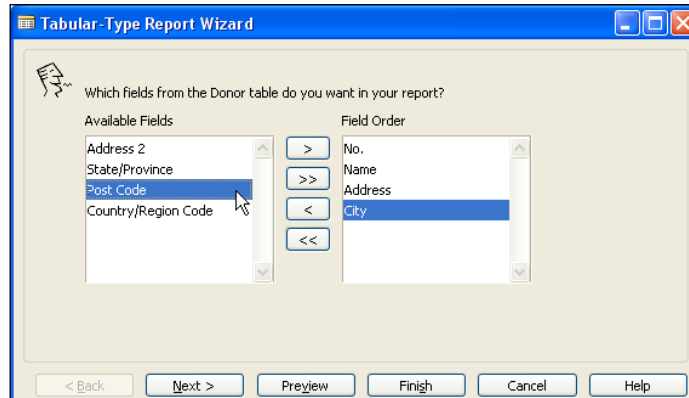
First, we will create a simple list format report based on our Donor table. We will use the Report Wizard. The Report Wizard is quite useful for simple reports. The Report Wizard and Report Designer tools are available to anyone who has a license containing the Report Designer granule.

When you are doing more complex reports, it is often only modestly helpful to start with the Report Wizard. For one thing, the Report Wizard only deals with a single input table. However, even with complex reports, sometimes it is a good idea to create a limited test version of some aspect of a larger report. Then you may create your full report without use of the Wizard, but using the Report Wizard generated code as your model for some aspect of layout or group totaling.

Open the **Object Designer**, click on **Report**, and then click on **New**. The Report Wizard's first screen will appear. Enter the name (Donor) or number (50000) of the table with which you want the report to be associated. Choose the option **Create a report using a wizard**. This time choose a **Tabular-Type Report Wizard** to create a Donor List. Then click on **OK**, as shown in the following screenshot:



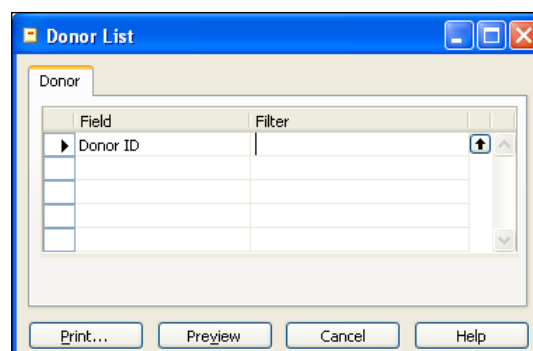
Next, you will be presented with the window shown in the following screenshot for choosing what data fields you want on your report. This is very much like what we saw earlier in the Page Wizard. The order in which you place the fields in the **Field Order** column, top to bottom (as shown in the following screenshot), will be the order in which they appear on your report, left to right.



You can preview your report in the Classic report viewer during the process of its creation, to see if you are getting the layout you want. Often you will perform quite a bit of manual formatting after you finish with the Wizard. As the report preview function utilizes the driver for the current default printer for formatting, make sure you have a default printer active and available before you attempt to preview an NAV report.

After you have chosen the fields you want on your report (some are suggested in the preceding screenshot), click on **Next**. This will bring up a screen allowing you to predefine a processing/printing sequence for this report. You can select from any of the defined keys. At the moment our `Donor` table only has one key so let's choose the **No, I don't want a special sorting of my data**. As you will see later, that will generate a report where you can choose the sort sequence for each run. Click on **Next**.

Now you can choose a List Style or a Document Style layout. Click on each one of them to get an idea of the difference. If you like, **Preview** the report in process for each of these options chosen. Note that when you **Run** a report (and previewing it, is running it), the first thing you see is called the Report request screen. A default Report request screen from the **Donor List** report would look like the following screenshot:

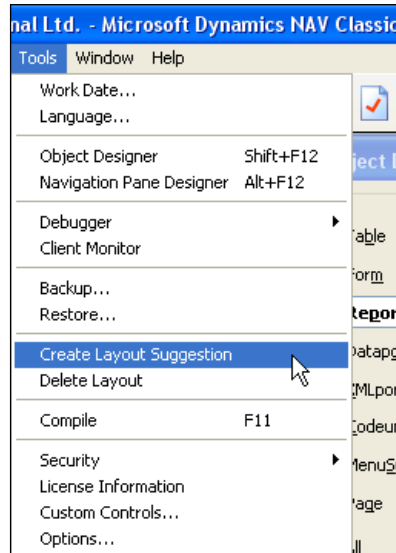


The Report request screen is where the user enters any variable information, data filters, page, printer setups, and desired sort order to control each report run. If no key is selected for the report, the **Sort** button will appear at the bottom right of the screen. This allows the user to choose which data key will apply to a report run. The user can also choose to **Print** the report (output to a physical device or PDF file) or **Preview** it (output to the screen). To start with, just click on **Preview** to see your current layout.

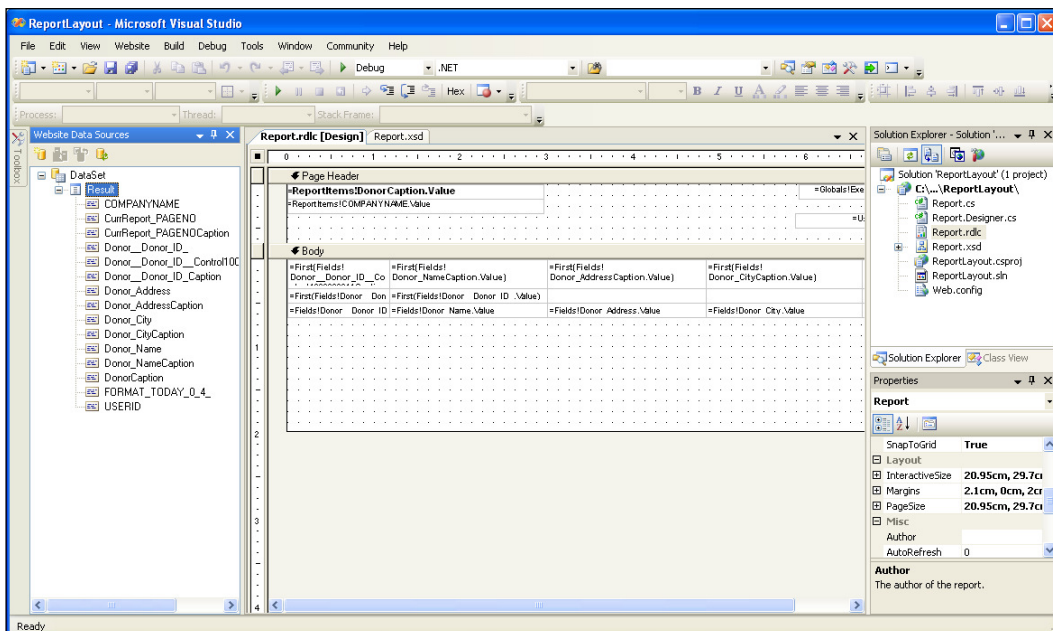
Once you are satisfied with the layout, click on **Finish**. You will now be in the Report Designer, ready to make manual changes. Exit the Report Designer by pressing the close-window icon, saying **Yes**, you do want to save changes. Save your new report as ID 50000 and Name "Donor List". Run your new report, using the Object Designer **Run** button. It should look much like the following:

Donor				September 26, 2009
CRONUS International Ltd.				Page 1
				Dave
Donor ID	Name	Address	City	
Donor ID	1001			
1001	Juan O'Hara	1123 Riviera Way	Duluth	
Donor ID	1002			
1002	Karen Johnson Fashions, LTD			
Donor ID	1003			
1003	Hans StatenBacker Research			
Donor ID	1004			
1004	John Smith	11 Oak Street	Chicago	

Our next step is to transform the Classic Client report layout into an RTC (RDLC) report layout. Highlight Report 5000 and click on **Design**. Then click on **Tools | Create Layout Suggestion** (see the following screenshot).

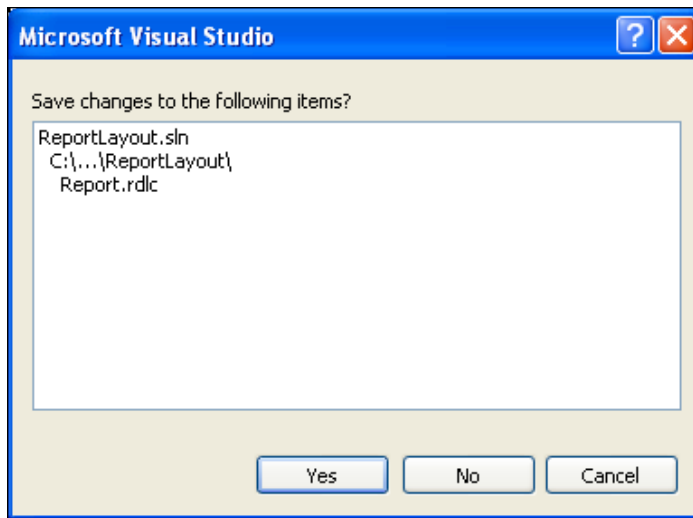


Your system will whirr and hum and buzz and a display similar to the following will appear:



You have been transported along with your transformed report layout into the Microsoft Visual Studio Report Layout tool. This tool gives you access to a considerable variety of graphical effects, interactive reporting functions, and other capabilities. We will explore these in detail in Chapter 5, *Reports*.

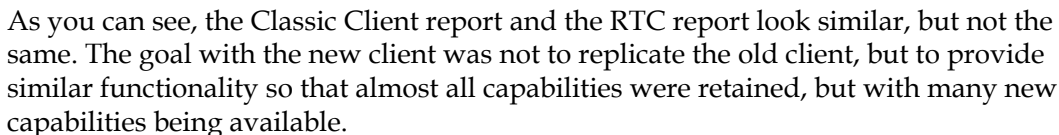
While you could make many different changes to this layout, for now just exit Visual Studio via the *Esc* key. As you exit Visual Studio, you will be presented with the following question on screen:



Click on **Yes**, which will take you back to the Classic Report Designer. Exit from the Report Designer, saving your revised report the same way you saved the original earlier.

By the way, if you had made changes to your layout and saved them, then later invoked the **Create Layout Suggestion** again, that would overwrite your layout, changes, and all. After the first time, access to this tool for a report should be via **View | Layout**.

In order to test the new RTC version of the Donor List report, simply highlight the report line in the **Object Designer** and click on **Run**. The RTC Report Request Screen will display, looking like the following screenshot:



We are done with our introduction to development for now. We will continue with our introductory review of NAV's object types.

Codeunits

A codeunit is a container for "chunks" of C/AL code to be run "inline" or called from other objects. These "chunks" of code are properly called Functions. As we said earlier that you could use a Report object as a container for code segments (that is, functions), why do we need codeunits? One reason is that early in the life of C/SIDE, only codeunits could be used in this way. Somewhere along the line, the C/SIDE developers decided to relax that particular constraint. However, from a system-design point of view, there are very good reasons to use codeunits as the primary containers for functions.

The most important reason to put all callable functions within codeunits is that codeunits can be exposed as Web Services. This allows the functions within a codeunit which has been published as a Web Service to be invoked by external routines operating outside of NAV. Only codeunits and pages can be exposed as Web Services.

A second important reason for using codeunits for callable functions is that the Microsoft provided NAV license specifically limits access to the C/AL code within codeunits differently than that within reports. The C/AL code within a report can be accessed with a "lower level" license than is required to access the C/AL code in a codeunit. If your customer has license rights to the Report Designer, they can access C/AL code within Report objects. A large percentage of installations have Report Designer license privileges. But they cannot access C/AL code within codeunit objects unless they have access to a more expensive license with Developer privileges (that is Application Builder or Solution Developer). As a result, C/AL code within codeunits is more secure from casual changes than is C/AL code within report objects.

A third reason is that the codeunits are better suited structurally to contain only functions. Even though functions could be placed in other object types, the other object types have superstructures that relate to their designed primary use for forms, reports, and so on. The use of such an object primarily as a repository for functions designed to be called from other objects creates code that is often more difficult to interpret, use, and maintain.

Codeunits act only as a container for C/AL coded functions. They have no auxiliary functions, no method of user interaction, and no pre-defined processing. If you are creating one or two functions that are closely related to the primary activity of a particular report, but these functions are needed from both within and outside of the report, then, by all means, include the functions in the report. Otherwise, use a Codeunit.

There are several codeunits delivered as part of the standard NAV product, which are really function libraries. These codeunits consist totally of utility routines, generally organized on some functional basis (for example, associated with Dimensions or with some aspect of Manufacturing or some aspect of Warehouse management). Some developers create their own libraries of favorite special functions and include such a "function library" codeunit in systems on which they work.

If a Codeunit is structured very simply and can operate in a stand-alone mode, it is feasible to test it in the same way one would test a Report or a Page. Highlight the Codeunit and click on the **Run** button. The codeunit will run for a single cycle.

MenuSuites

MenuSuites are the objects that are displayed in the Classic Client as User Menus. They differ considerably from the other object types we have discussed earlier. MenuSuites have a completely different structure; they are also maintained differently. In older versions of NAV, menus were constructed as versions of Form objects. With the release of Version 4.0, MenuSuites were offered as a way of providing a User Interface essentially similar to that found in the Outlook Navigation panel. This is the way they appear in the Classic client. MenuSuites are also maintainable in a limited way by the end user without any special license requirements. In addition, MenuSuites have the advantage of only showing the menu items that the user has permissions to access.

MenuSuite entries do not have maintainable properties or contain triggers. Within MenuSuites, NAV developers lost the ability to embed C/AL code within the menus. The only customizations that can be done with MenuSuites is to add, delete, or edit menu entries.

In the Role Tailored Client the data in the MenuSuites object is presented in the Department Menus. As you might expect, the presentation of the information is different in the RTC. In a later chapter, we will discuss more about how to work with MenuSuites and how to work around some of the constraints. The Role Center provides us with a number of new capabilities in that regard.

Dataports

Dataports exist as separate object types only in the two-tier (Classic) client. In the Role Tailored Client, the functions handled by Dataports in the Classic Client are handled by XMLports.

Dataports are specialized objects designed to export and import data between the two-tier NAV database (whether Classic or SQL Server) and external text files. Dataports allow for a limited set of external data formats, generally focused around what are commonly referred to as "comma separated value" (also known as "comma-delimited" or csv) files. The delimiters don't actually have to be commas, that's just the common name for this file structure.

Dataports can contain C/AL logic that applies to either the importing or the exporting process (or both). The internal structure of a dataport object is somewhat similar to that of a report object combined with a table object. Dataports are driven by an internal read-then-process loop similar to that in reports. Dataports contain field definitions that relate to the specific data being processed.

Dataports are relatively simple and quite flexible tools for importing and exporting data. The data format structure can be designed into the dataport as along with logic for accommodating editing, validating, combining, filtering, and so on of the data as it passes through the dataport. Dataports can be accessed directly from a menu entry, in the same fashion as forms and reports in the Classic Client.

XMLports

At first glance, XMLports are for importing and exporting data, similar to the Dataports. But XMLports differ considerably in their operation, setup, and primary intended usage.

In the two-tier client, XMLport objects can only be used for XML-formatted data. They must be "fired off" by C/AL code resident in some other object type (in other words, a Classic Client XMLport cannot be run from the Object Designer and cannot be run directly through a menu entry).

In the three-tier client, XMLports now handle both XML structured data and other text data that was previously handled by Dataports. The description of a Dataport's functionality now applies to the three-tier client XMLport. The three-tier client XMLport can be run directly from a menu entry.

XML stands for eXtensible Markup Language. XML is a markup language much like HTML. XML was designed to describe data so that it would be easier to exchange data between dissimilar systems, for example, between your NAV ERP system and your accounting firm's financial analysis and tax preparation system.

XML is designed to be extensible, which means that you can create or extend the definition as long as you communicate the revised XML format to your correspondents. There is a standard set of syntax rules to which XML formats must conform. XML is becoming more and more important because most new software uses XML. For example, the new versions of Microsoft Office are quite XML "friendly". All web services communications are in the form of an exchange of XML structured data.

Integration tools

NAV's integration tools are designed to allow direct input and output between NAV databases and external, non-NAV routines. However, they do not allow access to C/AL-based logic. The internal business rules or data validation rules that would normally be enforced by C/AL code or trigger actions or various properties do not come into play when the data access is by means of one of the following integration tools. Therefore, you must be very careful in their use.

- **N/ODBC:** NAV provides the standard ODBC interface between external applications (such as Word, Excel, Delphi, Access, and so on) and the Classic NAV database. This is a separately licensed granule. N/ODBC does not work with the SQL Server database.
- **C/OCX:** This provides the ability to use OCXs to interface with the NAV database. This is also a separately licensed granule.
- **C/FRONT:** This provides the ability to access the NAV database directly from code written in languages other than C/AL. Earlier, this type of interface was primarily coded in C, but beginning with V4.0 SP1, we now have the ability to interface from various .NET languages. In future versions, this capability is likely to expand. This too is a separately licensed granule.
- **Automation:** This allows access to registered automation controller libraries within Windows from in-line C/AL code (for example, C/AL code can directly push data into a Word document template or an Excel spreadsheet template from C/AL). Automation controllers cannot be used to add graphical elements to NAV but they can contain graphical user interfaces that operate outside of NAV. When it is feasible to use an automation controller for interfacing externally, this is a simple and flexible way to expand the capabilities of your NAV system.
- **Web services:** This functionality is described in brief earlier and in much more detail later. Web services are an industry standard API to NAV tables (through published pages) and functions (through published codeunits).

Backups and documentation

As with any system where you can do development work, careful attention to documentation and backing up your work is very important. C/SIDE provides a variety of techniques for handling each of these tasks.

When you are working within the Object Designer, you can back up individual objects of any type or groups of objects by exporting them. These exported object files can be imported in total, selectively in groups or individually, one object at a time, to recover the original version of one or more objects.

When objects are exported to text files, you can use a standard text editor to read or even change them. If, for example, you wanted to change all the instances of the field name **Customer** to **Patient**, you might export all the objects to text and execute a mass "Find and Replace". You won't be surprised to find out that making such code changes in a text copy of an object is subject to a high probability of error, as you won't have any of the many safety features of the C/SIDE editor keeping you from hurting yourself.

You can also use the NAV Backup function to create backup files containing just system objects or including data (a typical full system backup). A developer would typically use backup only as an easy way to get a complete snapshot of all the objects in a system. Backup files cannot be interrogated as to the detail of their contents, nor can selective restoration be done. So, for incremental development backups, object exporting is the tool of choice. As NAV data is relational and highly integrated, it would generally not be good practice to attempt to backup and restore single data tables.

Internal documentation (that is, inside C/SIDE, not in external documents) of object changes can be done in three areas. First is the Object Version List, a field attached to every object, visible in the Object Designer screen. Whenever a change is made in an object, a notation should be added to the Version List. The second area for documentation is the Documentation section that appears in most object types. The third area you can place documentation is inline with modified C/AL code.

In every object type except *MenuSuites*, there is a *Documentation* section at the top of the object. The *Documentation* section is the recommended location for noting a relatively complete description of any changes that have been made to the object. Then, depending on the type of object and the nature of the specific changes, you should also consider annotating each change everywhere it affects the code, so the changes can be easily identified as such by the next developer looking at this object.

In short, everything applies that you have learned earlier about good backup practices and good documentation practices, when doing development in NAV C/SIDE. This holds true whether the development is new work or modification of existing logic.

Summary

In this chapter, we covered some basic definitions of terms related to NAV and C/SIDE. We reviewed the two-tier and the three-tier configuration options. Then we followed with the introduction of eight types of NAV objects (Tables, Pages/Forms, Reports, Codeunits, MenuSuites, Dataports, and XMLports). We also had an introduction to Page and Report Creation through review and hands-on use with the beginning of a NAV application for the ICAN not-for-profit organization. Finally, we looked briefly at the tools that we use to integrate with external entities and we discussed how different types of backups and documentation are handled in C/SIDE. Now that we have covered these basics in general terms, let's dive into the detail of the primary object types.

In the next chapter, we will focus on Tables, the foundation of an NAV system.

Review questions

1. Microsoft Dynamics NAV 2009 is an ERP system. ERP stands for Extended Report Processor. True or False?
2. Which of the following are true about Dynamics NAV 2009? Choose three:
 - a. It has two clients, a Role Tailored Client and a Classic Client
 - b. It supports two databases, the Classic Database and Microsoft SQL Server
 - c. Role Tailored Client reporting primarily uses Excel for formatting
 - d. Support for Web Services is integrated as a standard feature
3. NAV 2009 is a full object-oriented system. True or False?
4. Which of the following choices are object types included in NAV 2009? Choose two:
 - a. Report, Dataport, Page, Codeunit
 - b. Report, XMLport, Form, SQL Server
 - c. PDF, XMLport, Report, Form
 - d. XMLport, Form, Table, MenuSuite
5. Development for NAV 2009 can all be done from within the C/SIDE environment. True or False?
6. All NAV objects except XMLports can contain C/AL code. True or False?
7. Licenses control not only what functions users have access to, but also what objects and object number ranges developers have access to. True or False?
8. The Classic Client is tied to the two tier system and the Role Tailored Client is tied to the three tier system. True or False?
9. NAV 2009 uses Journal tables for transaction data. Ledger tables are used at times for temporary work tables and at other times for permanent, Posted data storage. True or False?
10. Classic Clients run only Forms and the Role Tailored Clients run only Pages. True or False?

11. Which of the following have Wizards to help initiate the design of the object?
Choose three:
 - a. MenuSuites
 - b. Reports
 - c. Pages
 - d. Forms
12. Keyboard shortcuts in the Role Tailored Client are a superset of those in the Classic Client. True or False?
13. A Visual Studio compatible report layout tool is an important part of the NAV 2009 development toolset. True or False?
14. Reports, Forms, Pages, Dataports and Codeunits can all be run directly from the Object Designer screen for testing. MenuSuites, Tables, and XMLports cannot be. True or False?

2

Tables

Design builds the bridge between the black box of technology and everyday practice – Gui Bonsiepe

The basic building blocks of any system are data definitions. In NAV, the data definitions are made up of tables. Within these tables individual data fields exist. Whether you are working on a new add-on or a tightly integrated modification, the first level of detailed design for a NAV application must be the data structure.

In NAV, the table definition can encompass much more than traditional data fields and keys. The table definition should also include a significant portion of the data validation rules, processing rules, business rules, and logic to ensure referential integrity.

In this chapter, we will learn how to design and construct NAV tables. We will review the various choices available and how these choices can affect the subsequent phases of design and development.

Overview of tables

A table provides the basic definition for data in NAV. It is important to understand the distinction between the table (definition and container) and the data (the contents). The **table definition** describes the data structure, validation rules, storage, and retrieval of the data stored in the table. The **data** is the raw material that originates (directly or indirectly) from the user activities and subsequently resides in the table. For example, in the Permissions setup, the data is formally referred to as **Table Data**. The table is not the data, it is the definition of data. However, we commonly refer to the data and the table as if they were one and the same. This is the terminology we will generally use in this book.

Tables are the critical foundation blocks of NAV applications. All permanent data must be stored in a table. As much as possible, key system design components should be embedded in the tables. This means fully utilizing the capability of NAV table objects to contain code, properties, and so on, which will define their content and processing parameters.

You should include code that controls what happens when new records are added, changed, or deleted, as well as how data is validated. All of these should be a part of the table. To a great extent, the table object should include the functions commonly used in various processes to manipulate the table and its data, whether for database maintenance or in support of business logic.

We will soon explore these capabilities more completely through examples and analysis of the structure of table objects. You will find that this approach has a number of advantages:

- Centralization of rules for data constraints
- Clarity of design
- More efficient development of logic
- Increased ease of debugging
- Easier upgrading

Components of a table

A table is made up of Fields, Properties, Triggers (some of which may contain C/AL code), Keys, and SumIndexes. A table definition which takes full advantage of these tools reduces the effort required to construct the other parts of the application. In turn, this will have a considerable impact on the processing speed, efficiency, and flexibility of the application. These components can be combined to implement many of the business rules of the application, as well as the rules for data validation.

A table can have:

- Up to 500 fields
- A defined record size of up to 4 KB (up to 8 KB for SQL Server)
- Up to 40 different keys

Table naming

There are standardized naming conventions defined for NAV. Your modification will fit better within the structure of NAV if you follow these conventions. In all of these cases, the names for tables and other objects should be as descriptive as possible, while keeping to a reasonable length. This is one way to make your work self-documenting (which, of course, reduces the required amount of auxiliary documentation).

Table names should always be singular. The table containing customers should not be named Customers, but Customer. The table we created for our International Community And Neighbors NAV enhancement was named Donor, even though it contains data on many donors.

In general, you should always name a table so it is easy to identify the relationship between the table and the data it contains. Consistent with the principle of being as descriptive as possible, two tables containing the transactions on which a document page is based should normally be referred to as a Header table (for the main portion of the page) and a Line table (for the line detail portion of the page). As an example, the tables underlying a Sales Order page are the Sales Header and the Sales Line tables. The Sales Header table contains all the data that occurs only once for a Sales Order, while the Sales Line table contains the multiple lines from the order. Additional information on table naming can be found in the *Terminology Handbook for C/SIDE* from Microsoft. Other important table design standards information can be found in the online Help files of NAV, and in the *C/AL Programming Guide*, also from Microsoft.

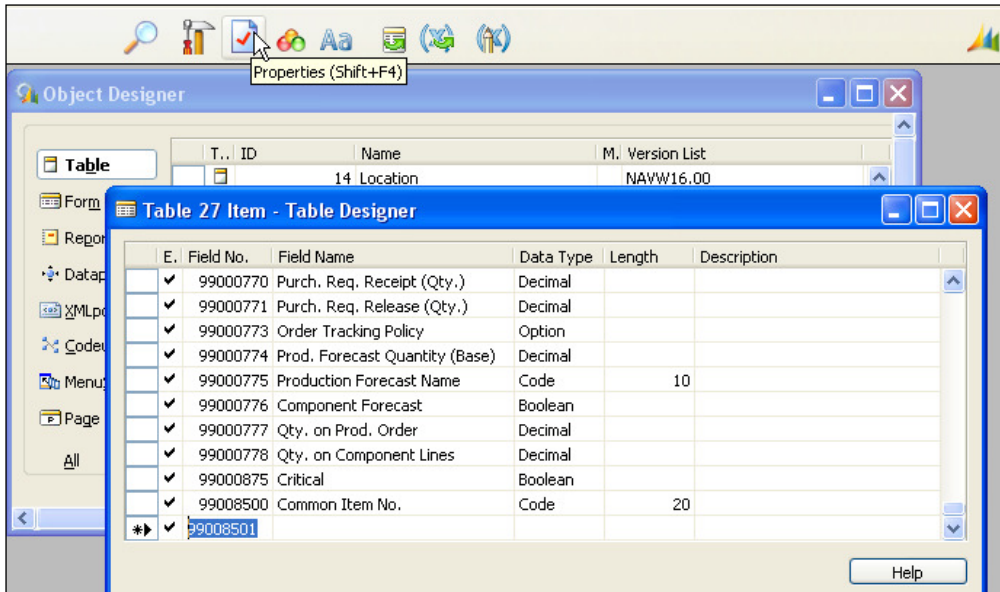
Table numbering

There are no hard and fast rules for table numbering, except that you must only use the table object numbers which you are licensed to use. If all you have is the rights to the Table Designer, you are only allowed to create tables numbered from 50000 to 50009. If you need more table objects, you can purchase table objects numbered up to 99999. ISVs can purchase access to tables in other number ranges.

If you are creating several related tables, ideally you should assign them related numbers in sequential order. But there are no particular limitations to assigning table numbers as long as it makes sense to you. In general, let common sense be your guide to assigning table numbers.

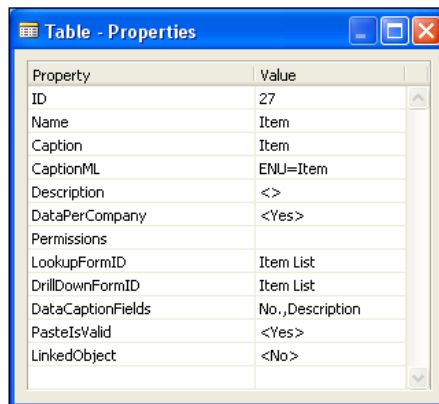
Table properties

You can access the properties of a table while viewing the table in Design mode, highlighting the first blank field line (the one below all the fields), and clicking on the **Properties** icon or pressing *Shift + F4* as shown in the following screenshot:



You can also perform a similar operation via **Edit | Select Object**, and then pressing *Shift + F4*.

This will take you to the **Table - Properties** display. The following screenshot is the **Table - Properties** display for the **Item** table in the demonstration company database, Cronus, which is included in each copy of NAV.



The contents of the screenshot are as follows:

- **ID:** This is the object number of the table.
- **Name:** This is used as the default caption when data from this table is displayed.
- **Caption:** This contains the caption defined for the currently selected language. The default language for NAV is US English.
- **CaptionML:** This defines the MultiLanguage caption for the table. For an extended discussion on the language capabilities of NAV, refer to the section MultiLanguage Functionality in the online *Microsoft Dynamics NAV 2009 Developer Help*.



The online *Microsoft Dynamics NAV 2009 Developer Help* combines information that was previously in the online C/SIDE Help and the hardcopy of *Application Designer's Guide*. It is available from the Classic Client **Help | C/SIDE Reference Guide** and the **MSDN Library**.

- **Description:** This is an optional use property for your documentation.
- **DataPerCompany:** This lets you define whether or not the data in this table is segregated by company (the default), or whether it is common (shared) across all of the companies in the database. The names of tables within SQL Server (not within NAV) are affected by this choice.
- **Permissions:** This allows you to grant users of this table different levels of access (r=read, i=insert, m=modify, d=delete) to the table data in other table objects. For example, users of the Customer table are allowed to read (that is view) the data in the Cust. Ledger Entry table.
- **LookupFormID:** This allows you to define what Form/Page is the default for looking up data in this table.
- **DrillDownFormID:** This allows you to define what Form/Page is the default for drilling down into data that is summarized in this table.



The **LookupFormID** and **DrillDownFormID** properties refer to Forms when the Classic Client is running, but they refer to Pages when the Role Tailored Client is running.

- **DataCaptionFields:** This allows you to define specific fields whose contents will be displayed as part of the caption. For the Customer table, the No. and the Name will be displayed in the caption banner at the top of a Form/Page showing a customer record.

- **PasteIsValid:** This determines if the users are allowed to paste data into the table
- **LinkedObject:** This lets you link the table to a SQL Server object. This feature allows the connection, for data access or maintenance, to a non-NAV system or an independent NAV system. For example, a `LinkedObject` could be an independently hosted and maintained special purpose database, and thus offload that processing from the main NAV system. Many other uses are also feasible.

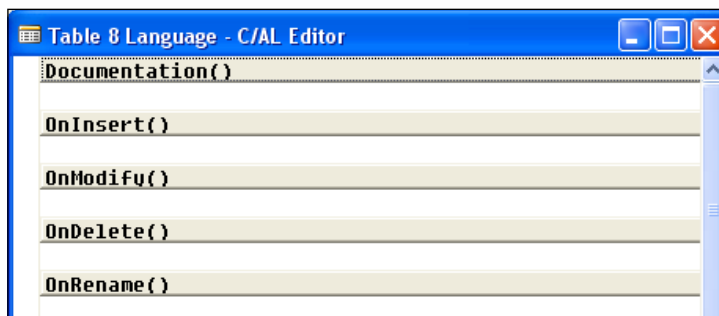
As a developer, it is most likely that you will continually deal with the `ID`, `Name`, `LookupFormID`, and `DrillDownFormID` properties, and occasionally with the `Caption`, `CaptionML`, `DataCaption`, and `Permissions` properties. You will rarely deal with the others.

Table triggers

The first Table section is the Documentation section. This section serves only the purpose of being a place to put whatever documentation you require. No C/AL code is executed in a Documentation section. There are no syntax or format rules here, even though there are recommendations in the *Microsoft C/AL Programming Guide*.

Every change to an object should be briefly documented in the Documentation section. The use of a standard format for such entries makes it easier to create them as well as understand them two years later.

The Documentation section has the same appearance as the triggers that appear in a Table definition. There are four Table triggers, each of which can contain C/AL code, shown in the following screenshot:



The code contained in a trigger is executed prior to the event represented by the trigger. In other words, the code in the `OnInsert()` trigger is executed before the record is inserted into the table. This allows the developer a final opportunity to perform validations and to enforce data structures such as referential integrity.

You can even abort the intended action if data inconsistencies or conflicts are found. The triggers shown in the previous screenshot work in the following manner. These triggers are automatically invoked when record processing occurs as the result of User Interface action. When table data changes occur as the result of C/AL code instruction or data imports, the triggers are not automatically invoked, but they will be dependent on instruction from the code.

- `OnInsert()`: This is executed when a new record is to be inserted in the table through the User Interface. (In general, new records are added when the last field of the Primary Key is completed and focus leaves that field. See the `DelayedInsert` property in the chapter on Pages for an exception).
- `OnModify()`: This is executed when any field other than a Primary Key field in a record is changed, determined by the `xRec` (before record) copy being different from the `Rec` (current record) copy. During your development work, if you need to see what the before value of a record or field is, you can always reference the contents of `xRec` and compare that to the equivalent portion of `Rec`, the current value in memory.
- `OnDelete()`: This is executed before a record is to be deleted from the table
- `OnRename()`: This is executed when some portion of the Primary Key of the record is about to be changed. Changing any portion (that is, the contents of any field) of the Primary Key is considered a Rename action. This maintains referential integrity. Unlike most systems, NAV allows the Primary Key of any master record to be changed and automatically maintains all the affected references from other records.

It is interesting to note that there is an apparent inconsistency in the handling of data integrity by NAV. On one hand, the Rename trigger automatically maintains referential integrity by changing all the references back to a record whose Primary Key is changed (renamed). However, if you deleted that record, NAV doesn't do anything to maintain referential integrity. In other words, child records could be orphaned by a deletion, and left without any parent record. As the developer, you are responsible for ensuring this aspect of referential integrity on your own.

When you write C/AL code that updates a table within some other object (for example, a Codeunit, a Report, and so on.), you can control whether or not the applicable table update trigger fires (executes). For example, if you were adding a record to our Donor table and used the following C/AL code, the `OnInsert()` trigger would fire.

```
Donor . INSERT (TRUE) ;
```

However, if you use either of the following C/AL code options instead, the `OnInsert()` trigger would not fire and none of the attendant logic would be executed.

```
Donor.INSERT(FALSE);
```

or

```
Donor.INSERT;
```

Default logic such as enforcing Primary Key uniqueness will still happen whether or not the `OnInsert()` trigger is fired.

Keys

Table keys are used to identify records, and to speed up filtering and sorting. Having too few keys may result in painfully slow inquiries and reports. However, each key incurs a processing cost, because the index containing the key must be updated every time information in the record's key fields change. Key cost is measured in terms of increased index maintenance processing. The determination of the proper number and design of keys for a table requires a thorough understanding of the types and frequencies of inquiries, reports, and other processing for that table.

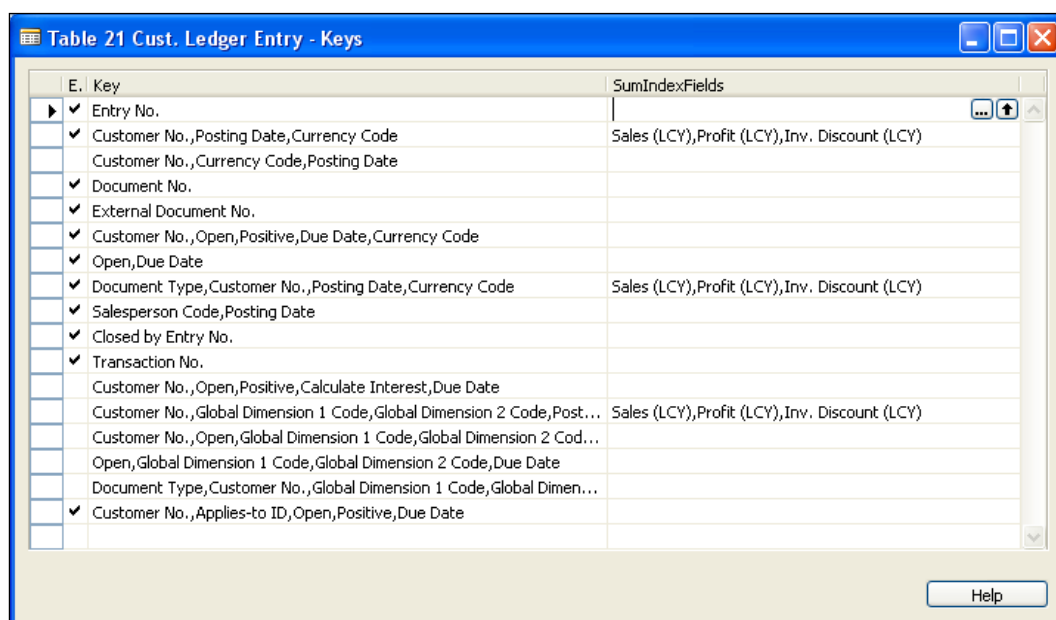
Every NAV table must have at least one key – the Primary Key. The Primary Key is always the first key in the key list. By default, the Primary Key is made up of the first field defined in the table. In many of the Reference (for example, LookUp) tables there is only one field in the Primary Key and the only key is the Primary Key. An example is shown by the **Payment Terms** table in the following screenshots:

E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>	1	Code	Code	10	
<input checked="" type="checkbox"/>	2	Due Date Calculation	DateFor...		
<input checked="" type="checkbox"/>	3	Discount Date Calculation	DateFor...		
<input checked="" type="checkbox"/>	4	Discount %	Decimal		
<input checked="" type="checkbox"/>	5	Description	Text	50	
<input checked="" type="checkbox"/>	6	Calc. Pmt. Disc. on Cr. Memos	Boolean		

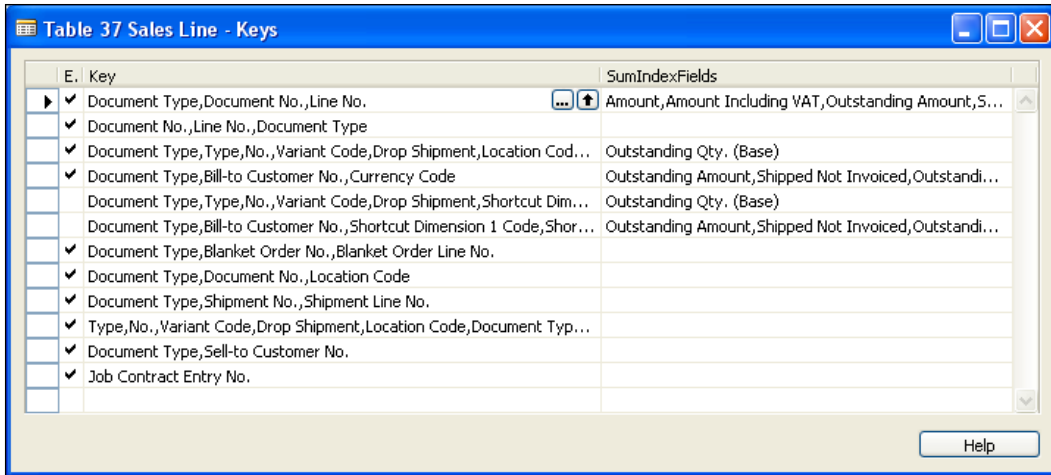
E.	Key	SumIndexFields
<input checked="" type="checkbox"/>	Code	

The Primary Key must have a unique value in each table record. You can change the Primary Key to be any field, or combination of fields up to 20 fields, but the uniqueness requirement must be met. It will automatically be enforced by NAV, in other words, NAV will not allow you to add a record in a table with a duplicate Primary Key.

If you look at the Primary Keys in the supplied tables, you will note that many of them consist of or terminate in a Line No., an Entry No., or some other data field whose contents serve to make the key unique. For example, the **Cust. Ledger Entry** table in the following screenshot uses the **Entry No.** alone as the Primary Key. It is a NAV standard that **Entry No.** fields contain a value that is unique for each record.



The Primary Key of the **Sales Line** table shown in the following screenshot is made up of multiple fields, with the Line No. of each entry as the terminating key field. In NAV, Line No. fields are assigned a unique number within the associated document. The Line No. combined with the preceding fields in the Primary Key (usually including fields such as Document Type and Document No.) makes each Primary Key entry unique. This is handled by C/AL code within the standard product and is not an inherent framework feature.



E. Key	SumIndexFields
Document Type, Document No., Line No.	Amount, Amount Including VAT, Outstanding Amount, S...
Document No., Line No., Document Type	
Document Type, Type, No., Variant Code, Drop Shipment, Location Cod...	Outstanding Qty. (Base)
Document Type, Bill-to Customer No., Currency Code	Outstanding Amount, Shipped Not Invoiced, Outstandi...
Document Type, Type, No., Variant Code, Drop Shipment, Shortcut Dim...	Outstanding Qty. (Base)
Document Type, Bill-to Customer No., Shortcut Dimension 1 Code, Shor...	Outstanding Amount, Shipped Not Invoiced, Outstandi...
Document Type, Blanket Order No., Blanket Order Line No.	
Document Type, Document No., Location Code	
Document Type, Shipment No., Shipment Line No.	
Type, No., Variant Code, Drop Shipment, Location Code, Document Typ...	
Document Type, Sell-to Customer No.	
Job Contract Entry No.	

All keys except the Primary Key are secondary keys. There are no uniqueness constraints on secondary keys. There is also no requirement to have any secondary keys. If you want a particular secondary key not to have duplicate values, you must code the logic to check for duplication before completing the new entry.

The maximum number of fields that can be used in any one key is 20. At the same time, the total number of different fields that can be used in all of the keys cannot exceed 20. If the Primary Key includes three fields (as in the preceding screenshot), then the secondary keys can utilize up to seventeen other fields (20 minus 3) in various combinations, plus any or all of the fields in the Primary Key. If the Primary Key has 20 fields then the secondary keys can only consist of different groupings and sequences of these 20 fields. The 20 field constraint results from the fact that all secondary keys have the Primary Key fields appended, behind the scenes, to the end of the key.

Some other significant key attributes include Keyscan and SQL Server-specific properties.

- Key scan can be enabled or disabled. Disabled keys are not automatically updated until they are enabled again. You can either enable or disable keys manually from the Table Designer key viewing screen or through program control. There is a **Key Group** property that can be set up by the developer to allow users to manually enable and disable groups of keys at one time (that is by Key Group). A Key Group could be a "group" of just one key. NAV 2009 provides functions to allow Key Groups to be disabled or enabled in code. This can be very useful for managing keys that are only needed for particular, infrequently used reports. For example, if a key or set of keys that is only used for monthly reports is disabled during the month, no index maintenance processing will be required. Then, at the end of month, the key(s) can be enabled and the necessary index recreation processing be done as part of the monthly batch.
- As Microsoft strongly encourages more and more of NAV installations to be based on SQL Server, as opposed to the native C/SIDE database, more SQL Server-specific parameters are being added to NAV. If you are developing modifications for a single installation that is using SQL Server, you would be wise to tailor what you do to SQL Server. Otherwise, while there are still two database options, you should design your enhancements so that they will work well on both database options. When a design choice needs to be made, the choice should obviously be made in favor of the better design for the SQL Server database. The key properties can be accessed by highlighting a key in the Keys form, then clicking on the **Properties** icon or pressing *Shift + F4*.

SumIndexFields

Since the origination of NAV as Navision, one of its unique capabilities has been the SIFT feature. **SIFT** stands for **SumIndexField Technology**. SumIndexFields are decimal fields attached to a table key definition. These fields serve as the basis for FlowFields (automatically accumulating totals) and are unique to NAV. This feature allows NAV to provide almost instantaneous responses to user inquiries for summed data related to the SumIndexFields.

SumIndexFields are accumulated sums of individual fields (for example, columns). In the C/SIDE database, these are maintained automatically by NAV during updates of the data. As the totals are pre-calculated, they provide high-speed access to inquiries. In the NAV 2009 SQL Server database version, SIFT values are calculated on demand through the use of SQL Indexed Views. Future versions may handle this differently.

If, for example, users wanted to know the total of the **Amount** values in a **Ledger** table, the **Amount** field could be attached to any (or all) keys. FlowFields can be defined in another table as display fields that take the advantage of the SumIndexFields to give the users rapid response to **Ledger Amount** inquiries relating to those keys.

In a typical system, thousands or even hundreds of thousands of records might have to be processed to give such results. Obviously, this could be very time consuming. In NAV, using the C/SIDE database, as few as two records need to be accessed to provide the requested results. In the C/SIDE database, SumIndexFields are stored as part of the key structure. This makes them very quick to access for calculation, and relatively quick to access for updating.

Because the SIFT functionality is not natively built into SQL Server, FlowField inquiries require more processing when using the SQL Server. However, they still provide the same logical advantages. The use of Indexed Views also gives very fast results.

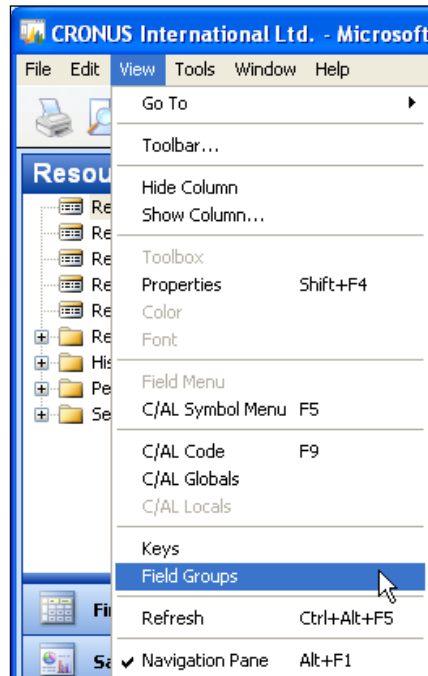
In general, excessive use of keys or SIFT fields can negatively affect system performance for two reasons. The first, which we already discussed, is the index maintenance processing load. The second is the table locking interference that can occur when multiple threads are requesting update access to a set of records that affect SIFT values. Conversely, the lack of necessary keys or SIFT definitions can also cause performance problems. In a nutshell, you should be careful in your design of keys and SIFT fields. We will discuss FlowFields in more detail in a later chapter.

Field Groups

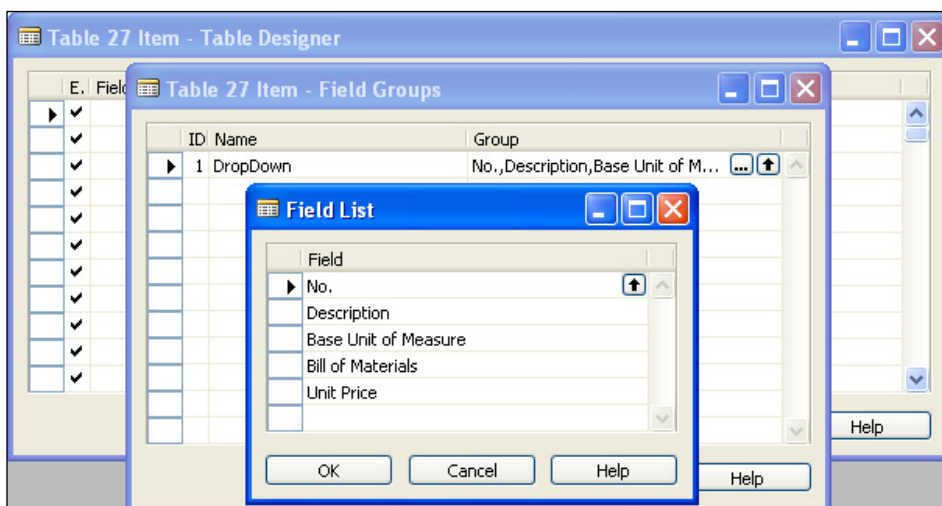
When a user begins to enter a data element in a field where the choices are constrained (for example an Item No., a Salesperson Code, a Unit of Measure code, a Customer No., and so on), good design dictates that the system will help the user by displaying the universe of acceptable choices. Put simply, when you try to enter such data, a lookup list of choices will be displayed.

In the Classic Client, the identification of the lookup form is defined by the contents of the **LookUpFormID** property. The form is a separately designed form object, just like all other forms. In the Role Tailored Client, the lookup page is generated dynamically (on the fly) when its display is requested by the user's effort to enter data in a field that references a table. The format of the lookup page is a basic list format (more on that in the chapter on Pages). The fields that are included in that page are either defined by default, or by an entry in the Field Groups table.

The Field Groups table is a component of the NAV table definition in the same way that the Keys table is a component of the NAV table definition. In fact, the Field Groups table is accessed very similar to the Keys, via **View | Field Groups** as shown in the following screenshot.



If we look at the Field Groups for **Table 27 – Item**, we see the following:



The lookup page (also referred to as the DropDown) created by this particular Field Group is shown in the following screenshot on the Sales Order page containing fields such as **No.** and **Description**, in the same sequence as in the Field Group definition.

Type	No.	Description	Location C...	Quantity	Reserved Qua...	Unit of
Item	1900-5	PARIS Guest Chair, black	BLUE			PCS

No.	Description	Base Unit ...	B...	Unit Price
1000	Bicycle	PCS	<input type="checkbox"/>	4,000.00
1001	Touring Bicycle	PCS	<input type="checkbox"/>	4,000.00
1100	Front Wheel	PCS	<input type="checkbox"/>	1,000.00
1110	Rim	PCS	<input type="checkbox"/>	0.00
1120	Spokes	PCS	<input type="checkbox"/>	0.00
1150	Front Hub	PCS	<input type="checkbox"/>	500.00
1151	Axle Front Wheel	PCS	<input type="checkbox"/>	0.00
1155	Socket Front	PCS	<input type="checkbox"/>	0.00

Buttons: New, Advanced, Set as default filter column

Additional fields on the right: 02/27/11, 2, 02/04/11, NATIONAL

If no Field Group is defined for a table, the system defaults to using the Primary Key plus the Description field.

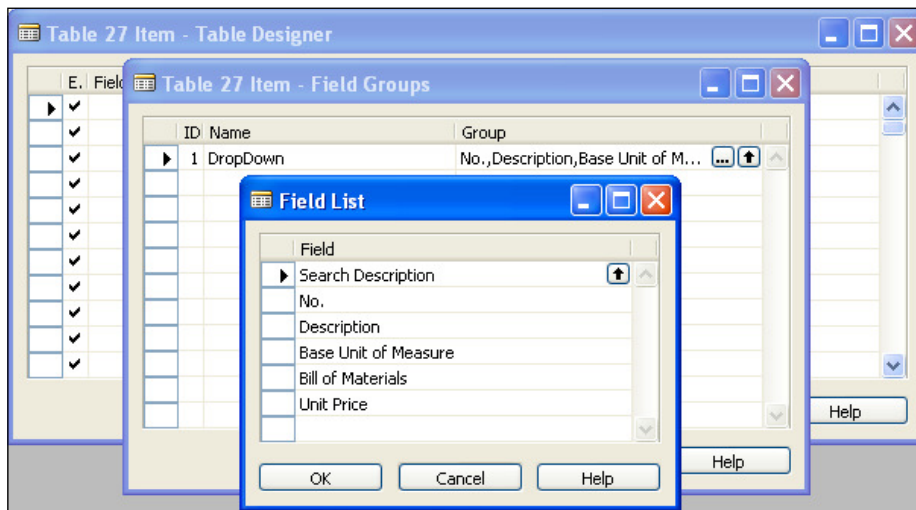
Since Field Groups can be modified, they provide another opportunity for an enhanced user interface. As you can see in the preceding screenshot, the standard structure for the fields in a Field Group is to have the Primary Key appear first. The user can choose any of the displayed fields to be the default filter column, the *defacto* lookup field.

As a system option, the Lookup/DropDown feature provides a **find-as-you-type** capability, where the set of displayed choices filters and re-displays dynamically as the user types, character by character. The filter applies to the default filter column. The disadvantage of this feature is the potential for the repetitive filtering to burden system hardware resources.

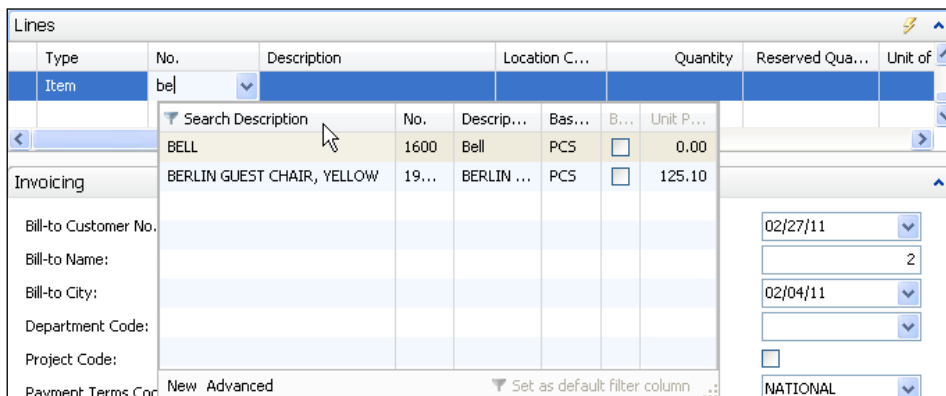
Whatever field is used for the lookup, the referential field defined in the page determines what data field contents are copied into the target table. In the preceding image example, the reference table/field is the Item table/field "No." and the target table/field is the Sales Line table/field "No."

As developers, we can change the order of appearance of the fields in the DropDown page. We can also change what fields are displayed, and add or delete fields by changing the contents of the Field Group. For example, we could add a capability to our page that mimics the standard Alternate Search capability of the Classic Client (where if the match for an Item No. isn't found in the **No.** field, the system will look for a match in the defined Alternate Search field).

Consider this situation: The customer has a system design where the Item No. contains a basic, hard to remember, sequentially assigned code to uniquely identify each item. But the **Search Description** field contains a product description that is relatively easy for the users to remember. When the user types, the find-as-you-type feature helps them focus and find the specific Item No. to be entered for the order. In order to support this, we simply need to add the **Search Description** field to the Field Group for the Item table as the first field in the sequence. The following screenshot shows that change in the Item Field Group table.



The effect of this change can be seen in the following screenshot which shows the revised DropDown page. The user has begun entry in the **No.** field, but the lookup is focused on the newly added **Search Description** field. Find-as-you-type has filtered the displayed items down to just those that match the data string entered so far.



The result of our change is to allow the user to lookup items by their **Search Description**, rather than by the less memorable **Item No.** Obviously, any of the fields in the Item table could have been used, including custom fields added for this implementation.

Expanding our sample application

Before moving on, we need to expand the design of our ICAN application. Our base Donor table design has to be enhanced, both by adding and changing some fields. We also need to add some reference tables.

Some of our design decisions here will be rather arbitrary. While we want to work with a realistic design, our primary goal is to create a design that we can use as a learning tool. We want to define data structures that can serve as a logical base for several typical application functions. If you see some capabilities missing and you want to add them, you should feel free to do so. Adjust these examples as you wish in order to make them more meaningful to you.

Creating and modifying tables

In Chapter 1, we created a Donor table for our ICAN application. At that time, we included the minimum fields to give us something to start with. Now, let's add a few more data fields to the Donor table and create an additional reference table. Then, we will revise our Donor table to refer to this new table.

Our new data fields are shown in the following table:

Field no.	Field name	Description
1000	Phone No.	Telephone number
1010	Alt. Phone No.	Alternate telephone number
1020	E-mail	Primary email address
1030	Donor Type	Choose one entry from a maintainable list, which will initially include Individual/Family, Business, and Institution
1040	Contact Method	Choose from a predefined list of options: Mail, Phone, Email
1050	Solicitation OK	A Boolean Yes/No entry
1060	Recognition Level	Based on the last year's donations, an option of Friend, Benefactor, Patron, or Leader
1070	Max Recognition Level	The highest recognition level achieved; an option of Friend, Benefactor, Patron, or Leader

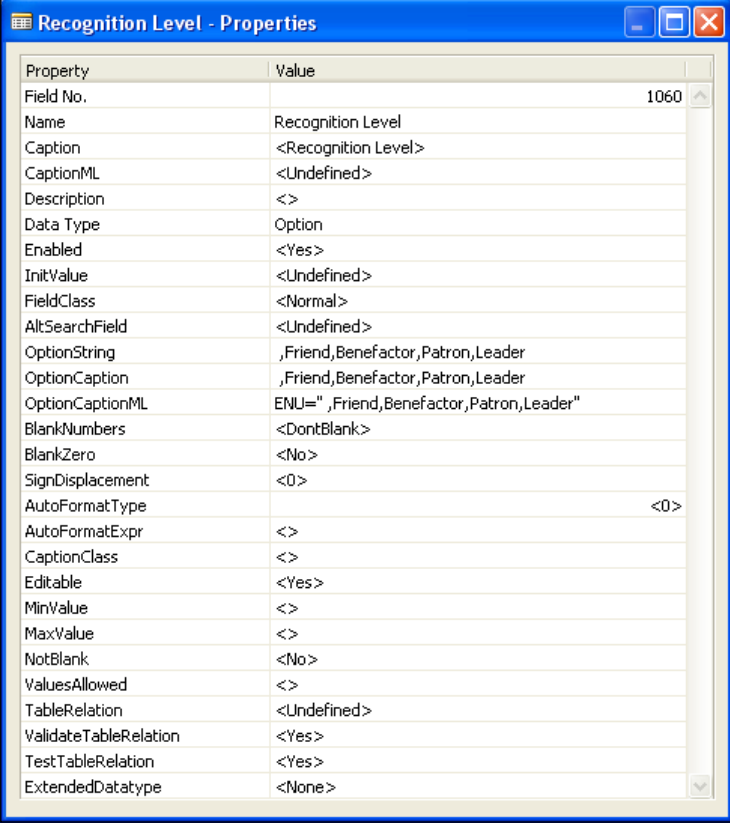
Field no.	Field name	Description
1080	Matching Gift Donor	If this Donor is employed by a firm with a Matching Gift policy, this will have the Donor No. of that Donor
1090	Newsletter	A Boolean Yes/No entry
1100	Date Added	The date this Donor entry was added
1110	Status	Option of Active or Inactive
1120	Alpha Name	Name entered to support desired alpha sort order
1130	Volunteer	Boolean

As these are new data fields describing the Donor, we will assign field numbers in a new range (1000 and up). Before **Phone No.**, we will leave numbering space for additional Donor descriptive fields to be inserted later, if any are needed.

Your task at this point is to open up your NAV, get to the **Object Designer | Tables** and find your Donor table (number 50000, remember?). Highlight the Donor table and click on the **Design** button. When you are done, the bottom part of your Donor table should look similar to the following screenshot:

E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>	10	Donor ID	Code	20	
<input checked="" type="checkbox"/>	20	Name	Text	50	
<input checked="" type="checkbox"/>	30	Address	Text	50	
<input checked="" type="checkbox"/>	40	Address 2	Text	50	
<input checked="" type="checkbox"/>	50	City	Text	30	
<input checked="" type="checkbox"/>	60	State/Province	Text	10	
<input checked="" type="checkbox"/>	70	Post Code	Code	20	
<input checked="" type="checkbox"/>	80	Country/Region Code	Code	10	
<input checked="" type="checkbox"/>	1000	Phone No.	Text	30	PN.01
<input checked="" type="checkbox"/>	1010	Alt. Phone No.	Text	30	PN.01
<input checked="" type="checkbox"/>	1020	E-mail	Text	80	PN.01
<input checked="" type="checkbox"/>	1030	Donor Type	Code	10	PN.01
<input checked="" type="checkbox"/>	1040	Contact Method	Option		PN.01
<input checked="" type="checkbox"/>	1050	Solicitation OK	Boolean		PN.01
<input checked="" type="checkbox"/>	1060	Recognition Level	Option		PN.01
<input checked="" type="checkbox"/>	1070	Max Recognition Level	Option		PN.01
<input checked="" type="checkbox"/>	1080	Matching Gift Donor	Code	20	PN.01
<input checked="" type="checkbox"/>	1090	Newsletter	Boolean		PN.01
<input checked="" type="checkbox"/>	1100	Date Added	Date		PN.01
<input checked="" type="checkbox"/>	1110	Status	Option		PN.01
<input checked="" type="checkbox"/>	1120	Alpha Name	Text	50	PN.01
<input checked="" type="checkbox"/>	1130	Volunteer	Boolean		PN.01

Next, we need to fill in the `OptionString` and `Caption` for the four `Option` fields. Highlight the **Recognition Level** field, and then click on the **Properties** icon or press *Shift + F4*. Enter the **OptionString** as shown in the next screenshot; don't forget the leading space followed by a comma to get a blank option. Copy and paste the same information into the **OptionCaption** property. The **OptionCaptionML** property will be automatically filled in. The resulting properties should look similar to the following screenshot:

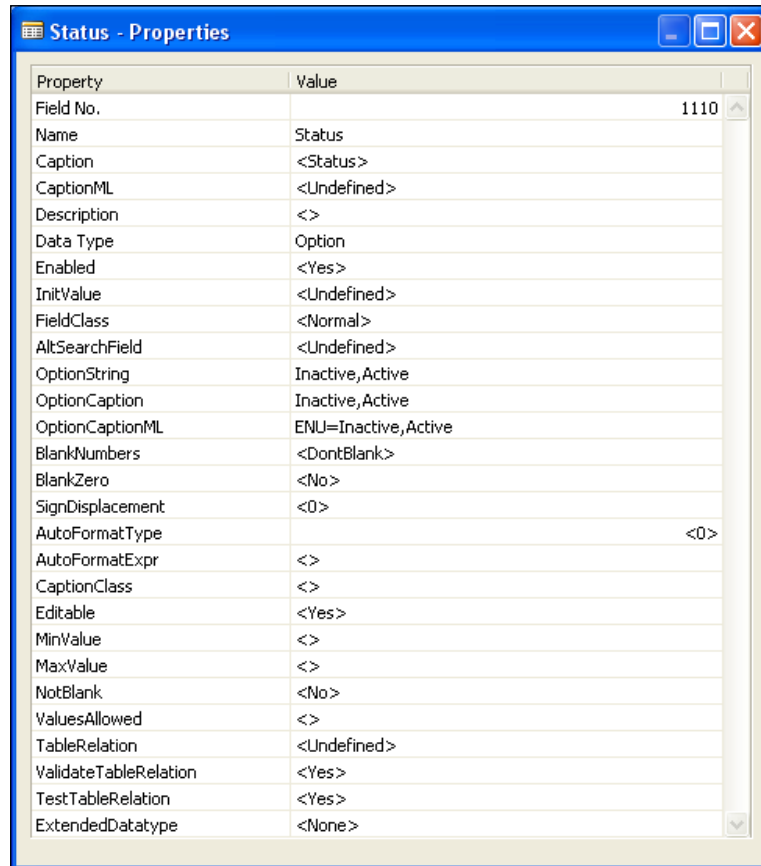


The screenshot shows a window titled "Recognition Level - Properties" with a list of properties and their values. The properties are listed in a table with two columns: "Property" and "Value".

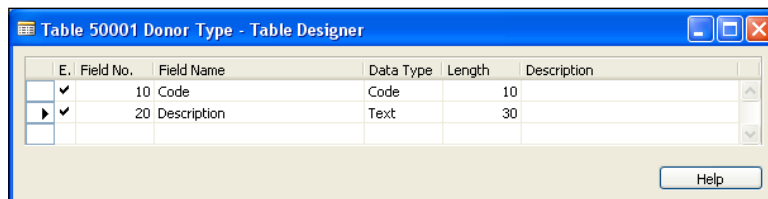
Property	Value
Field No.	1060
Name	Recognition Level
Caption	<Recognition Level>
CaptionML	<Undefined>
Description	<>
Data Type	Option
Enabled	<Yes>
InitValue	<Undefined>
FieldClass	<Normal>
AltSearchField	<Undefined>
OptionString	,Friend,Benefactor,Patron,Leader
OptionCaption	,Friend,Benefactor,Patron,Leader
OptionCaptionML	ENU=" ,Friend,Benefactor,Patron,Leader"
BlankNumbers	<DontBlank>
BlankZero	<No>
SignDisplacement	<0>
AutoFormatType	<0>
AutoFormatExpr	<>
CaptionClass	<>
Editable	<Yes>
MinValue	<>
MaxValue	<>
NotBlank	<No>
ValuesAllowed	<>
TableRelation	<Undefined>
ValidateTableRelation	<Yes>
TestTableRelation	<Yes>
ExtendedDatatype	<None>

The same change, of the same `OptionString`, should be applied to the **Max. Recognition Level** field.

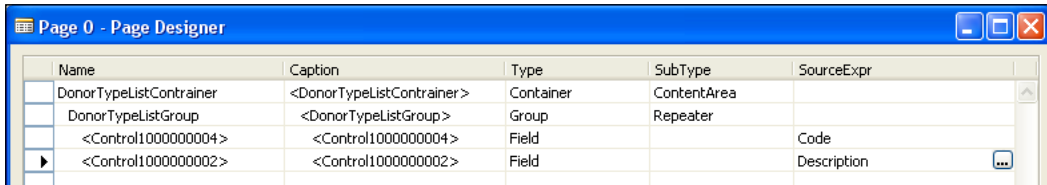
Now, similarly, enter the **OptionString** of **Inactive, Active** for the **Status** field. We are not going to have a blank option here, as we will let the default Status be Inactive (a design decision). Your resulting Status field Option properties should look like the following screenshot:



Next we want to define the reference table we are going to tie to the **Donor Type** field. We will keep this table very simple, just containing **Code** as the unique key field and a text **Description**. You should create a new table, define the two fields, and save this as **Table 50001 Donor Type**, shown as in the following screenshot:

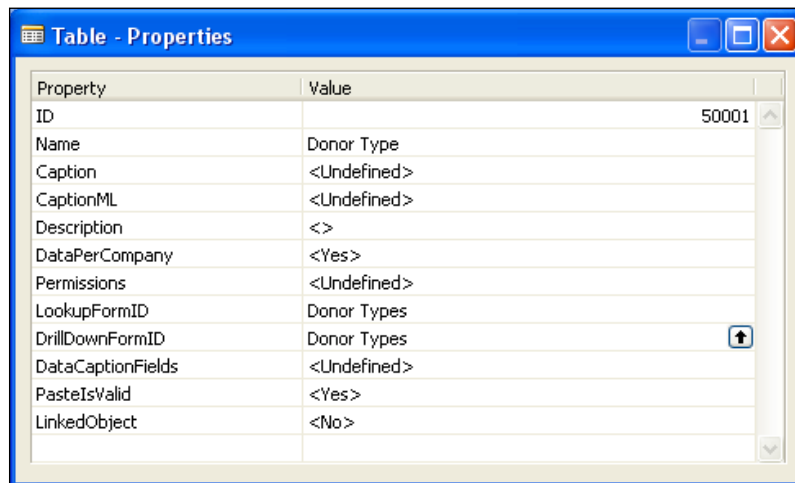


The next step is to use the **Page Designer** to create a List page for this table (Page 50002 – Donor Types). You should be able to move through this pretty quickly. Click on **Pages**, click on **New**, enter **50001** in the **Table** field, then choose the **Create a blank page of type:** option, and finally choose **List**. Populate the page with all (both) the fields from your Donor Type table. Your designed page should look like the following screenshot:



Exit the **Page Designer**, saving the page as number **50002**, named **Donor Types**. Test the page by running it from the Object Designer's **Run** button. This will invoke the Role Tailored Client (if it is not already running), then call up your new page 50002.

Return to the Donor Type table, set the **Table - Properties** of LookupFormID and DrillDownFormID to the new page we have just created. As a reminder, you will use **Design** to open the table definition, focus on the empty line below the **Description** field, and either click on the **Properties** icon or press **Shift + F4**. In the values for the two **FormID Properties**, you can enter either your Page name (Donor Types) or the Form/Page Object Number (50002). Either one will work and gives the result shown in the following screenshot. Then, exit and save the table as compiled.



Assigning a TableRelation property

Finally, open the Donor table again. This time highlight the **Donor Type** field and access its **Properties** screen. Highlight the TableRelation property and click on the ellipsis button (the three dots). You will see the **Table Relation** screen with four columns. The middle two columns are headed "Table" and "Field". In the top line, the Table column, enter **50001** (the table number) or **Donor Type** (the table name). In the same line, the Field column, click on the ellipsis button (...) and choose **Code**. Exit the Table Relation screen (by pressing the *Esc* key) and you will return to the **Donor Type - Properties** page that looks as shown in the following screenshot. Exit and save the modified table.

Property	Value
Field No.	1030
Name	Donor Type
Caption	<Donor Type>
CaptionML	<Undefined>
Description	<>
Data Type	Code
Enabled	<Yes>
DataLength	10
InitValue	<Undefined>
FieldClass	<Normal>
AltSearchField	<Undefined>
AutoFormatType	<0>
AutoFormatExpr	<>
CaptionClass	<>
Editable	<Yes>
NotBlank	<No>
Numeric	<No>
CharAllowed	<Undefined>
DateFormula	<No>
ValuesAllowed	<>
SQL Data Type	<Undefined>
TableRelation	"Donor Type".Code
ValidateTableRelation	<Yes>
TestTableRelation	<Yes>
ExtendedDatatype	<None>

Creating Forms for testing

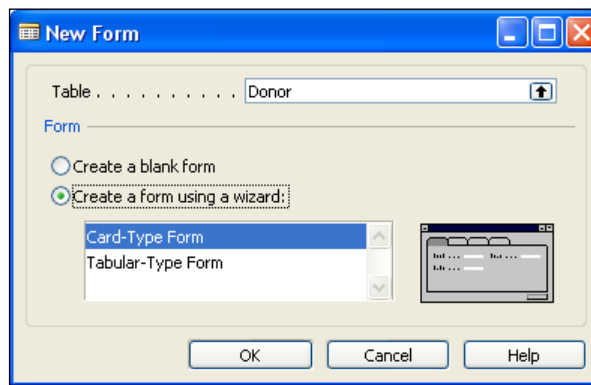
Until now, we have ignored the possibility that we might need forms to make some of our testing easier. Now we've reached the point in our system development process when it would be useful to have forms for the tables we have created.

Creating forms is quite easy, especially when the details of the layout are not critical. When you are creating forms for production use by users it is likely that layout will be important, but when the forms are just for use in entering and viewing test data, layout is generally not critical.

The process you follow to create a form using the Form Designer Wizard is similar to the first part of creating a page. The second part of the process, the placement of fields, is done outside the Wizard for a page, but for forms that is also done within the Wizard.

Creating a Card form

Open the Object Designer, click on **Form**, and then click on **New**. The Form Wizard's first screen will appear. Enter the name (**Donor**) or number (**50000**) of the table with which you want the form to be associated (bound). Choose the option **Create a form using a wizard**. Choose **Card-Type Form**. Then click on **OK**.

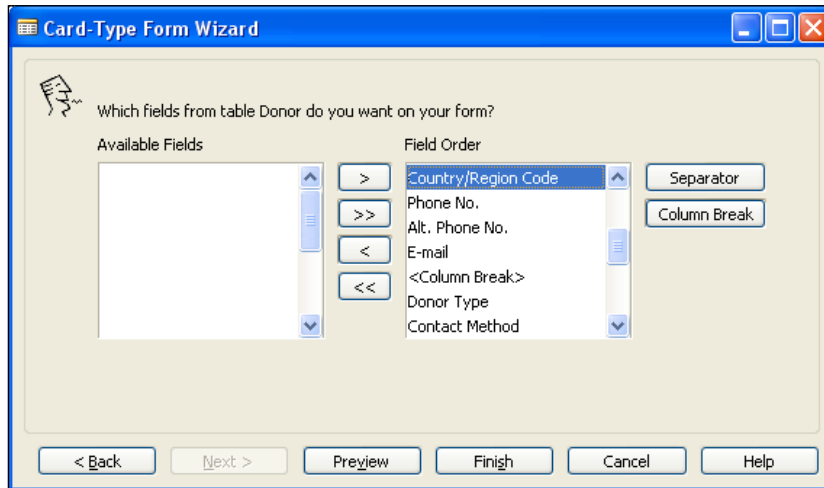


The resulting screen allows us the option of creating a plain form (no tabs) or a tabbed form with one or more tabs. We can name the tabs on this screen. As this form is just for our use in testing, and because all of our table fields will fit on one tab, choose the **No, I want a plain form** option, and then click on **Next**.

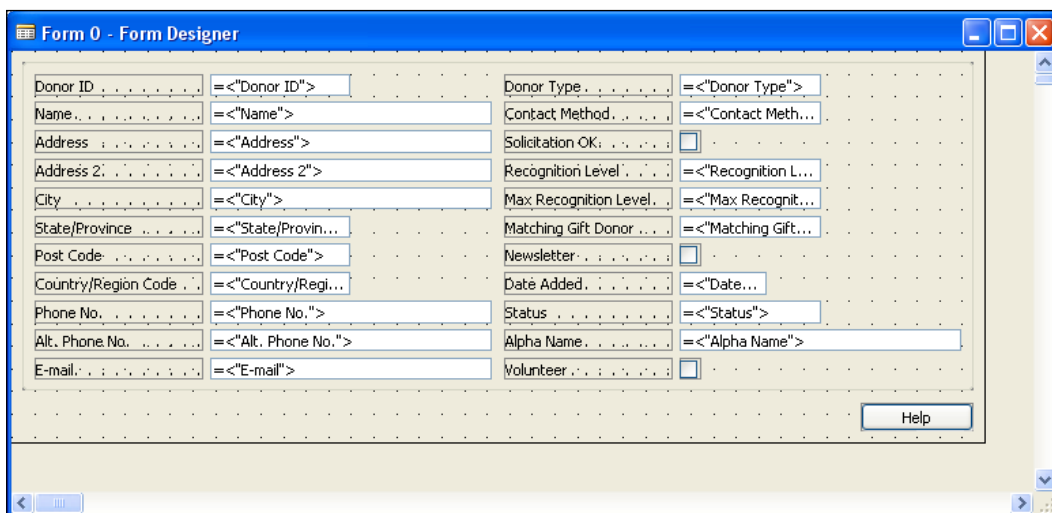
The next step is to choose what fields will appear on our new form and how they will be placed. In this instance, we're going to use all the fields and simply put them into two columns.

The left column in the next screen, headed **Available Fields**, lists all the fields from the source table that have not yet been chosen for the Form object. The right column, headed **Field Order**, lists all the fields that have been chosen, in the order in which they will appear on the Form.

In the following screenshot, you can see that we've selected all of the fields and split the columns between contact information and the control/tracking information. It's not a very elegant design, but it will suit our simple needs. Put all the fields in the table on the form.



At any point during your work you can take a sneak peek at what you are creating (that is, click on the **Preview** button). When you've got all the fields on your new form and you're satisfied with the layout, click on **Finish**. You will see a screenshot like the following, showing this generated form object in the **Form Designer**.



We're now through with the Form Wizard and have transitioned into the Form Designer. As this form is just for our use in testing, we're going to leave our form as is. If you later want to experiment with other layouts, this would be good way to practice using some of the C/SIDE tools.

If we wanted to modify the form manually because we were designing for production use by end users, we could do that now. Even if we did want to make some manual changes, it would be a good idea, before proceeding further, to first save what we've done thus far. We do that much the same as we did when we saved our table and various pages earlier. Press *Esc* and respond with **Yes** to the **Do you want to save the changes to the Form** query.

Now enter the Form number (ID) you want to assign (50000) and name (Donor Card). We're using the number 50000 so that the Donor Card form will have the same object number as the Donor Card page. As mentioned earlier, that will allow whichever client is running to find the appropriate object for data display. After you've got the form saved, we will make the Donor List form.

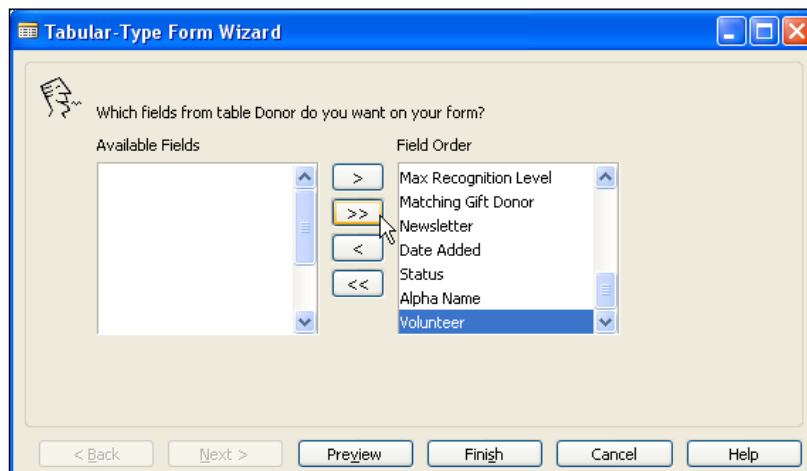
Creating List Forms

Just as for the Card form, we will use the Form Wizard to create a tabular Donor List form. Open the Object Designer, click on **Form**, and click on **New**. Once again, the Form Wizard's first screen will appear. Enter the name (**Donor**) or number (**50000**) of the table with which you want the form to be associated. Choose the **Create a form using a wizard** option. This time choose the option to create a **Tabular-Type Form**. Then click on **OK**. The **Tabular-Type Form Wizard** will appear as shown in the following screenshot:

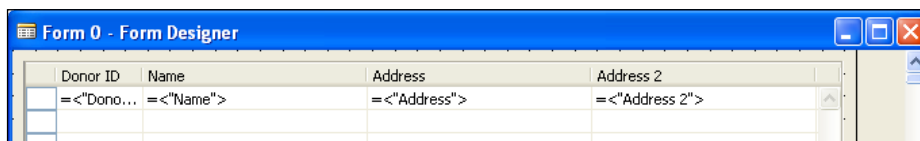


Now you have the opportunity to choose which data fields will appear on each line of the tabular display. When List forms are designed for some type of referential lookup, generally they don't contain all of the data fields available, especially when working with a larger table. However, as we are making this form for our use in testing, we will include all of the fields in the table. Note that the Form Wizard functions essentially the same for Tabular forms, as it does for Card forms.

Remember we can always return to the created form and easily add the fields we left off, or remove something we decide is no longer needed. Also, NAV forms include a feature which allows you to have some field columns identified as **Not Visible** by default. This property is field specific and controls whether the column for a data field displays on screen or not.



As with the Card form, at any point during your work, you can take a sneak peek at what you are creating (click on the **Preview** button). If you feel like experimenting, you could move fields on and off the form, or put fields in a different order. If you do experiment, use **Preview** to check the effects of your various actions. If your form gets hopelessly confused (which happens to all of us sometimes), just press *Esc*, be careful not to save the results, and then start over. When you've got all the desired fields on your new form and are satisfied with the layout, click on the **Finish** button. We're now done with the Form Wizard for our new List form and have transitioned into the **Form Designer** as shown in the following screenshot:



If we wanted to modify the form manually, we could do that now. As before, just press the *Esc* key and respond with **Yes** to the **Do you want to save the changes to the Form** query. Enter the Form number (ID) you want to assign (**50001**) and name (**Donor List**). If you reused the Form object number 50000, you would have overwritten the Card form you created earlier.

At this point, we have a data structure (**Table 50000 – Donor**), a form and page to enter and maintain data (**Form 50000 – Donor Card**), and a form and page to display or inquire into a list of data (**Form 50001 – Donor List**). To check out how your forms look, simply run them directly from the Object Designer.

Our final test form at this point will be a list form for the Donor Types table. The process to create a Donor Types list form is exactly the same as it was for the Donor List form, except that we select the Donor Type table as the source table at the beginning. Briefly, the steps are:

- Go to **Object Designer | Form | New**
- Enter Donor Type for the table and choose **Tabular-type Form**, and then click on **Next**
- Include all the fields and exit the Wizard, saving your form as **50002, Donor Types**

Test your new form 50002 by running it using the **Run** button running it by using the Run button.

The ZUP file

On a tabular form, even the non-programmer user can change the Visible property of each column to create a customized version. This user customization is tied to the individual user login and recorded in the user's **ZUP file**. ZUP files record user specific system state information so it can be retrieved when appropriate. Use of the ZUP file in this manner applies to the Classic Client, not to the Role-Tailored Client. In the Role-Tailored Client, user customization (aka tailoring) information is stored in tables within the database.

In addition to user screen changes, the ZUP file records the identity of the most recent record in focus for each screen, the contents of report selection criteria, request form field contents, and a variety of other information. When the user returns to a screen or report, either in the same session or after a logout and return, the data in the ZUP file helps to restore the state of various user settings. This feature is very convenient for the user.

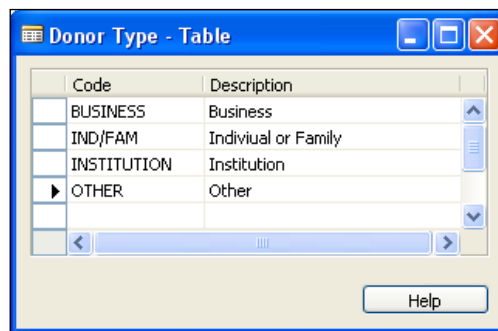
ZUP files are not used to store the tailoring that users do to pages within the Role-Tailored Client. Those changes are recorded as metadata (data about data) within the database. We will review that part of NAV in the chapter on Pages.

Testing a TableRelation property

Just before we started creating our forms for testing, we had assigned a TableRelation property in the Donor table. To check that the TableRelation is working properly, run the Donor table (that is, highlight the table name and click on the **Run** button) and scroll to the right until you have the cursor in the **Donor Type** field. You could also run the Donor List form and have almost exactly the same view of the data. This is because the **Run** of a table creates a temporary list form which includes all the fields in the table; basically the same as the form we created using the Form Wizard.

If all has gone according to plan, the **Donor Type** field will display a **Lookup** button (the upward pointing arrow button). If you click on that button or press *F6*, you should invoke **Form 50002 – Donor Types**, which you have just created.

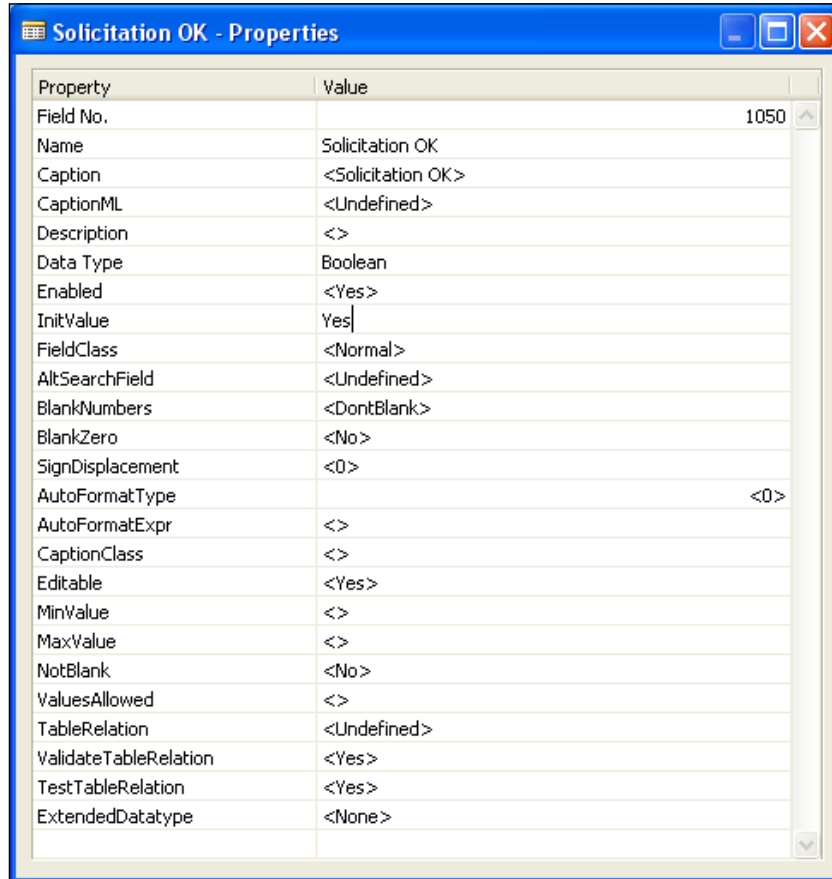
While you are in that form, go ahead and make some entries like the examples shown in the following screenshot:



Another Donor Table field that we can further define at this point is **Solicitation OK**, a simple Boolean field. Normally, Boolean fields default to No (False) and we want **Solicitation OK** to default to **Yes** (True). In other words, we are going to be optimistic when we enter a new Donor and assume that, most of the time, we will have permission to solicit them for future donations.

Setting the default for a field to a specific value simply requires setting the **InitValue** property to the desired value. In this case, that value is set to **Yes**. Using the Table Designer, design the Donor table, access the Properties screen for the **Solicitation OK** field. After you have filled in the value, exit the **Properties** screen, exit the Table Designer, and save the changes.

The result looks like the following screenshot:

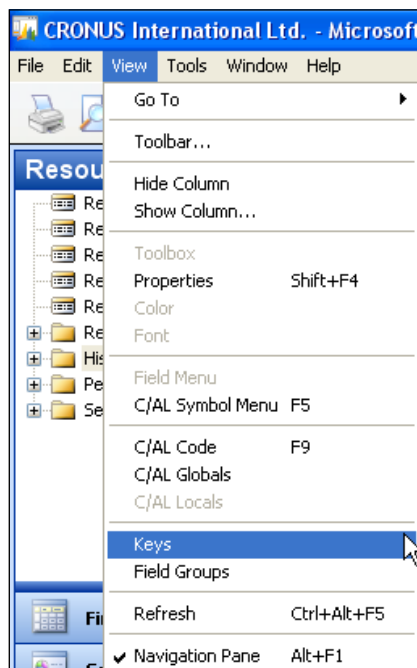


If you want to test what you've just done, you should add a new donor and see if the **Solicitation OK** field (which will appear as a blank for No or a checkmark for Yes) defaults to Yes.

After you've completed a successful test of the **Solicitation OK** field, make the same change to the **InitValue** property for the other two Boolean fields in the Donor table. Those are the **Newsletter** and **Volunteer** fields.

Adding Secondary keys

Let us add a couple of additional keys to our Donor table. Our original Donor table has a Primary Key consisting of just the Donor ID. You might find it useful to be able to view the Donor list geographically or alphabetically. To make that change, you will have to access the window for maintenance of a table's key by selecting **View**, then **Keys** from the menu bar at the top of the screen, as shown in the following screenshot:

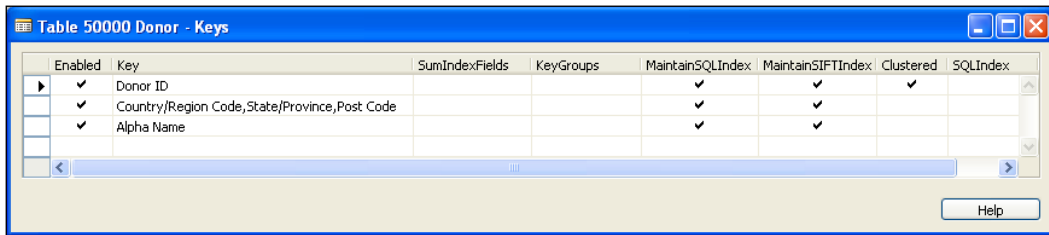


Once you have displayed **Keys**, you can then change the existing keys or add new ones. In order to add a new key, highlight the first blank line (or press *F3* to create an additional blank line above an existing key) and then click the ellipsis button (the one with three dots) to access the screen that will allow you to select a series of fields for your key.



You can also use the lookup arrow if you want to enter a single key field or want to enter several key fields one at a time. However, the end result will be the same. You should choose the tool option (ellipsis or arrow) that is easier for you to use. The fields will control the sort order of the table on this screen—for example, the top field is the most important, the next field is the second most important, and so on. When you exit this screen, make sure that you click **OK** otherwise your changes will be discarded.

For a good geographical Key, you might choose **Country/Region Code, State/Province**, or **City**, and for an alphabetical key you should choose **Alpha Name**, the field we created for this purpose. As these are secondary keys, you do not have to worry about duplicate entries. And remember, the Primary Key is automatically and invisibly appended to each secondary key. If the purpose of the key is only to provide a sort sequence for display or reporting, you can set the **Maintain SQLIndex property** to Unchecked (that is No), as the sorting will be provided by SQL Server on the fly. The index does not need to be maintained, thus providing more efficient processing.



In the chapter on Pages, we will create a new version of the Donor page which will show the new fields we have just added. For the time being, you can experiment updating the Donor table by using the **Run** button from the Object Designer. Don't forget to experiment with the sort feature by clicking on the **Sort** icon or pressing *Shift + F8*. In the later chapters, we will add C/AL code to both Table and Field triggers, explore SumIndexFields and other features, and build a more fully featured application.

Adding some activity-tracking tables

Our ICAN organization is a well organized and productive group. We track information about our donors and clients. We track the gifts we receive and the assistance that has been provided. We keep a record of requests for help that haven't been met as well as gifts we've received that haven't yet been allocated to meet a need. We also manage our campaigns of solicitation, send out a regular newsletter, and seek out volunteers for specific projects.

We aren't going to cover all these features and functions in the following detailed exercises. However, it's always good to have a fuller view of the system on which you are working, even if you are only working on one or two components. In this case, the parts of the system not covered in detail in our exercises will be opportunities for you to extend your studies and practice on your own.

Of course, any system development should start with a Design Document that completely spells out the goals and the functional design details. The *Microsoft Dynamics Sure Step* project methodology is a very useful toolset to utilize for project management. Neither system design nor project management will be covered in this book, but when you begin working on production projects, both these areas will be critical to your success.

Based on these requirements, we need to expand our application design. So far we have defined a minimal Donor table, one reference table (Donor Type), and created forms and pages for each of them. Ideally, you have also entered some test data and then added a few additional fields to the Donor table (which we will not add to our forms).

Now we will add some more reference tables, plus add a couple of **Ledger** (activity history) tables relating to Donor activities. Following that, we will also create some forms and pages to utilize our new data structures.

Our ICAN application will now include the following tables:

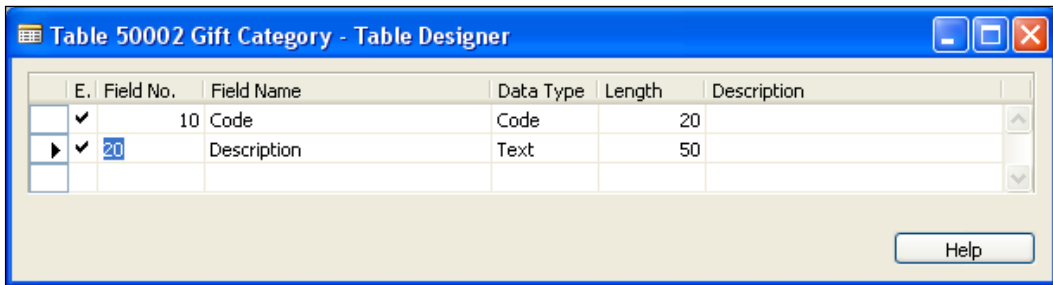
- **Donor:** A master list of all donors
- **Donor Type:** A reference list of possible types of donors
- **Client:** A master list of all clients
- **Gift Ledger:** A detailed history of all the gifts received and allocated
- **Campaign:** A master list of campaigns of solicitation
- **Aid Request Ledger:** A master list of requests for aid received from and granted to our clients
- **Gift Categories:** A reference list of descriptive categories of gifts

Remember, the purpose of this example system is for you to follow along in a hands-on basis within our system. You might want to try different data structures and other object features. For example, you could add functionality to track volunteer activity, perhaps even detailing the type of activity.

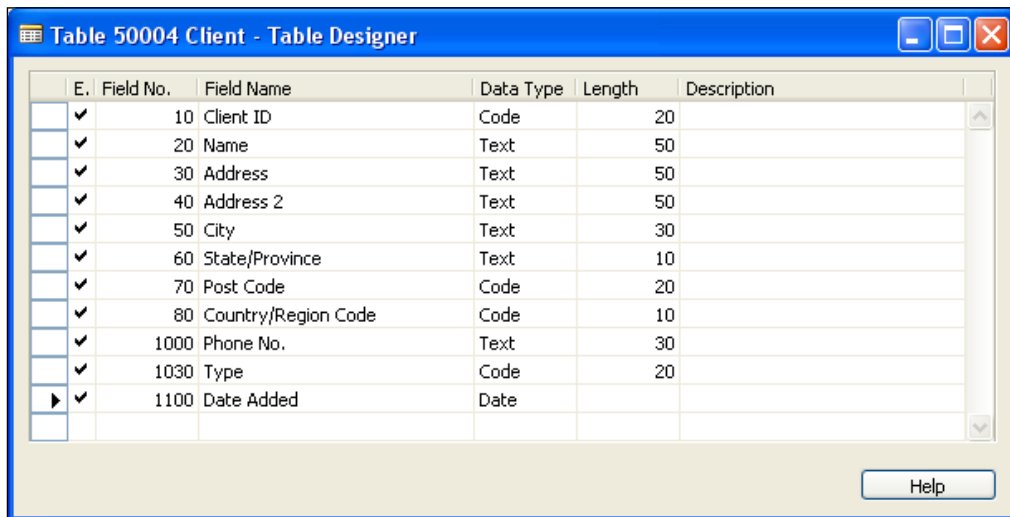
For the best learning experience, you should be creating each of these objects in your development system to learn by experimenting. In the course of these exercises, it will be good if you make some mistakes and see some new error messages. That's part of the learning experience. A test system is the best place to learn from mistakes at the minimum cost.

New tables

We will add one more reference table, which will contain the possible **Gift Category** for the donations received, as shown in the following screenshot:



We also need master tables for Clients and Campaigns. We'll start by creating a **Client** master table (**Table 50004**) that looks like the following:

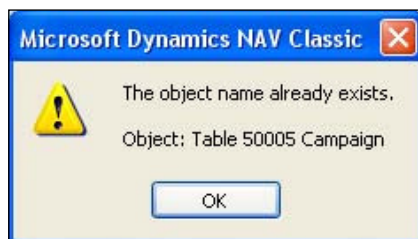


If you examine the Client table, you will see that every field has the same **Field No.** and specifications as one of the Donor table fields. Both tables describe individuals in our community. We can observe that from time to time, those who were generous in their giving fell on hard times and applied for assistance. Conversely, those who had been helped often became donors when they had something to share.

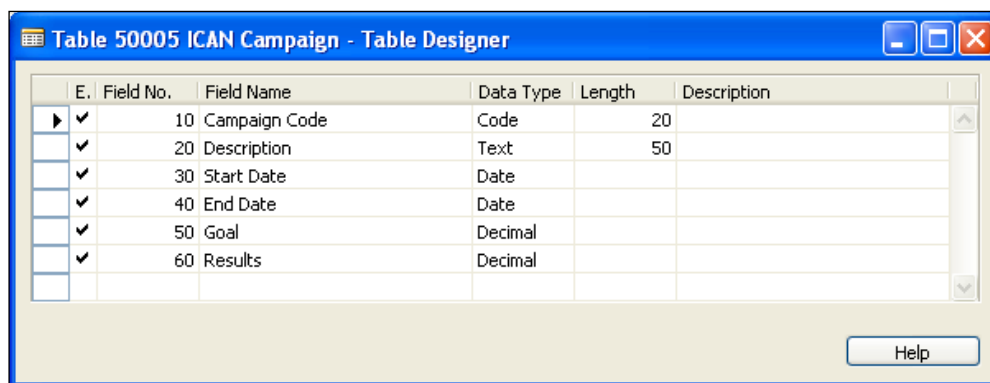
From a design point of view, we wanted to make it technically easy to copy data from one of these master tables to the other. Designing the matching fields to have the same **Field No.** and other specifications will make such copying easy.

Our Campaign master table has a relatively simple structure. If your creative juices are flowing and you wish to add additional functionality, this table is a good place to do that.

When we create our new table and attempt to save it as with the name Campaign, we find out that NAV already has a table by that name. NAV does not allow duplicate object names for objects of the same type. The error message is:



We will change the name for this table to ICAN Campaign and rename our new table definition. At a minimum, the new table should contain fields similar to those in the following screenshot:



Finally, we need two more tables. In order to track the history of gifts and pledges received, we need a ledger table. We need the same type of table to track the history of the requests for assistance and the grants of such requests. For these, we will create a Gift Ledger table and an Aid Ledger table.

The Gift Ledger should look like the following screenshot:

Table 50006 Gift Ledger - Table Designer

E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>	10	Entry No.	Integer		
<input checked="" type="checkbox"/>	20	Date	Date		
<input checked="" type="checkbox"/>	30	Description	Text	50	
<input checked="" type="checkbox"/>	40	Gift or Pledge	Option		
<input checked="" type="checkbox"/>	50	Category	Code	20	
<input checked="" type="checkbox"/>	60	Estimated Value	Decimal		
<input checked="" type="checkbox"/>	70	Hours	Decimal		
<input checked="" type="checkbox"/>	80	Donor ID	Code	20	
<input checked="" type="checkbox"/>	90	CAN Campaign	Code	20	
<input checked="" type="checkbox"/>	100	Anonymous	Boolean		
<input checked="" type="checkbox"/>	110	Matching Donor ID	Code	20	

Help

In a fully featured production system, we would need to deal with the issue of tracking the actual conversion of pledge promises into tangible gifts. If pledges were made but not fulfilled, we would want to have processes by which we followed this up. We aren't going to take our example system to that level of completeness however. This is a good example of a capability that you might want to add to the design on your own.

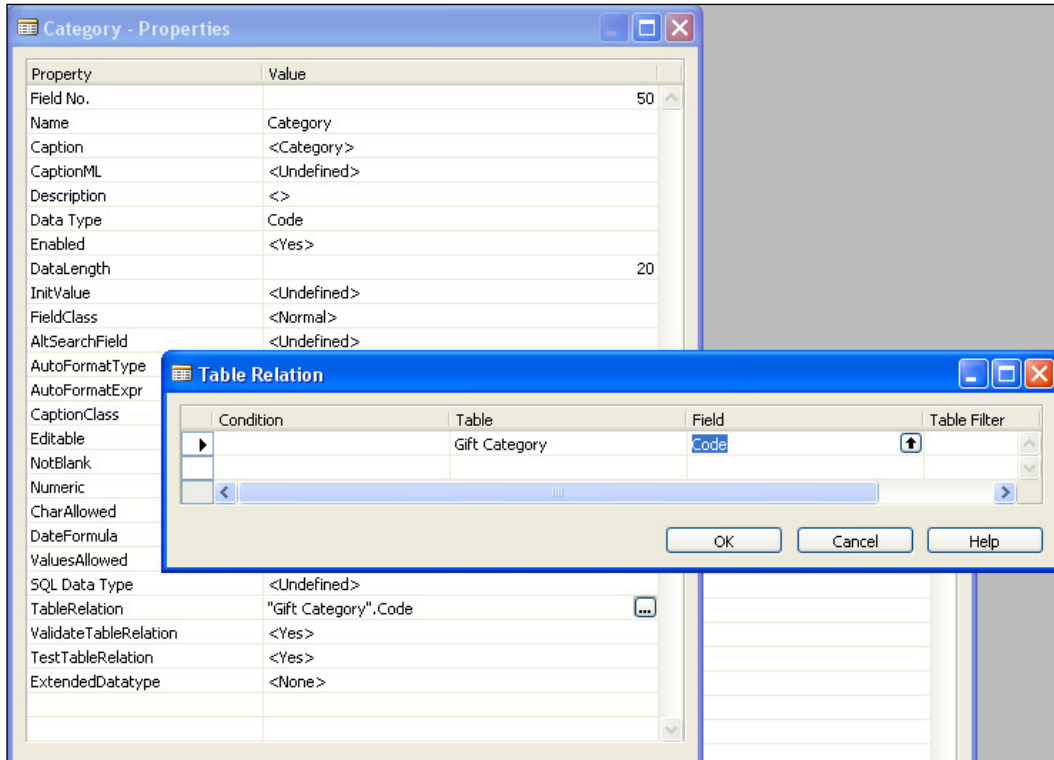
The Aid Ledger table definition should be similar to that shown in the following screenshot:

Table 50007 Aid Ledger - Table Designer

E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>	10	Entry No.	Integer		
<input checked="" type="checkbox"/>	20	Date Requested	Date		
<input checked="" type="checkbox"/>	30	Date Provided	Date		
<input checked="" type="checkbox"/>	40	Description	Text	50	
<input checked="" type="checkbox"/>	50	Request or Aid	Option		
<input checked="" type="checkbox"/>	60	Category	Code	20	
<input checked="" type="checkbox"/>	70	Estimated Value	Decimal		
<input checked="" type="checkbox"/>	80	Client ID	Code	20	
<input checked="" type="checkbox"/>	90	Gift Source Entry No.	Integer		

Help

In order to operate these tables in a relational fashion, the references to other related tables must be defined. In each case, the **Donor ID** field must refer to the Donor table and the **Category** field must refer to the Gift Category table. The following screenshot shows the forms involved in defining the **Category** field reference in the Gift Ledger table.



Similarly, you need to add a **TableRelation** property to the **Donor ID** field, pointing to the Donor table. Other fields for which table relations should be defined for the Gift Ledger table are as follows:

- ICAN Campaign to the ICAN Campaign table—to track what brought in this gift
- Matching Donor ID to the Donor table—for gifts where an employer matches employee contributions

Fields in the Aid Ledger table for which table relations should be defined are:

- Category to the Gift Category table
- Client ID to the Client table
- Gift Source Entry No. to the Gift table – that can match the use of specific gifts when appropriate to do so

In addition to the information that is required to tie each entry back to the related master table (**Donor ID** and **Client ID** respectively) and the data we are tracking for ledger purposes (for example, type of activity, value, and so on), we also include the **Description** field.



Based on the principles of relational database normalization, it might be reasonable to suggest that the **Description** field should not be duplicated into these ledger records. Even more vivid examples of non-normalized tables are the Sales Order Header and Sales Order Line tables. These have a lot more fields duplicated from the source tables. There is a considerable number of instances in NAV (and other similar systems) where duplicating data into related files is the better design decision.

There are two primary reasons for this. First, this approach allows the user to tailor the data each time it is used. In this case, it might mean that the description of the volunteer activity could be edited to provide more specific detail of what was done. Second, it allows easier and faster processing of the table data in question. In this case, if you want to sort the data in the **Description** field for reporting purposes, you should have the **Description** in the table; if you want to filter out and review only entries with certain key words in the **Description**, you should also have the **Description** in the table. Obviously, there are quite a number of advantages to putting ease of access and processing efficiency ahead of the principles of database normalization.

We have not yet discussed how we will get data into ledger tables through NAV standard processing. We will deal with that later. To start with, for initial test data entry, we will simply use default tabular forms that will take us through our first few testing steps. This is one way to tiptoe into a full system development effort. It allows us to validate our base table/data design (our foundation) before we spend too much effort on building production system functions and user interfaces.

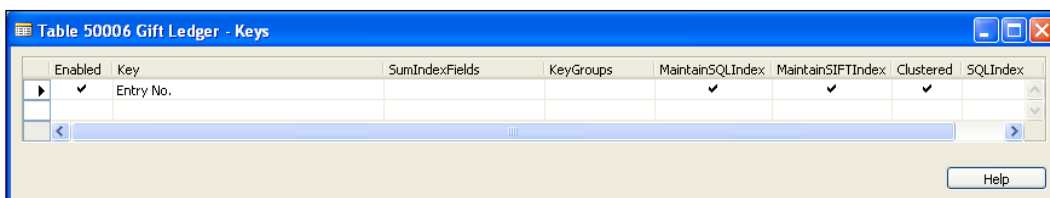
On this basis, we need to now create forms for all the tables for which we have not already created forms. Note that we are creating forms, not pages. We will create pages for the users of our production system, but right now we're just creating tools for our use as developers in order to enter test data.

For the purpose of test data entry, when we get to that point, you should now create List forms for the following tables using the Form Wizard:

- Client Master table
- Campaign Master table
- Gift Ledger table
- Gift Category reference table
- Aid Request Ledger table

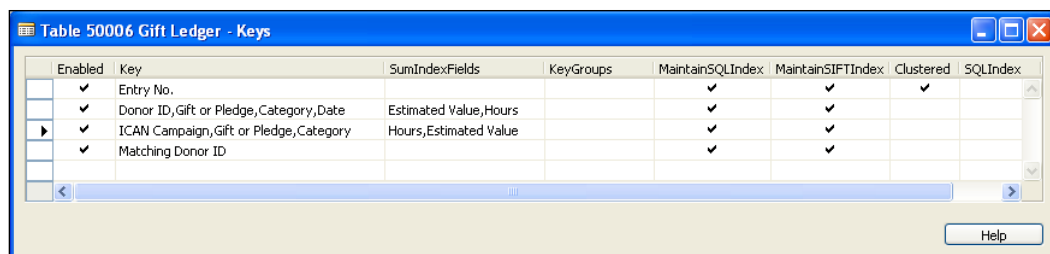
Keys and SumIndexFields in our examples

The following screenshot displays the **Keys** (only one was initially defined, and that one by default) for our **Gift Ledger**:



Enabled	Key	SumIndexFields	KeyGroups	MaintainSQLIndex	MaintainSIFTIndex	Clustered	SQLIndex
<input checked="" type="checkbox"/>	Entry No.			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

As we want to have a quick and easy access to the total **Estimated Value** of gifts and pledges as well by various groupings as the volunteer **Hours** contributed, we must define these fields as being SumIndexFields associated with the appropriate keys, as shown in the following screenshot:



Enabled	Key	SumIndexFields	KeyGroups	MaintainSQLIndex	MaintainSIFTIndex	Clustered	SQLIndex
<input checked="" type="checkbox"/>	Entry No.			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
<input checked="" type="checkbox"/>	Donor ID, Gift or Pledge, Category, Date	Estimated Value, Hours		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
<input checked="" type="checkbox"/>	ICAN Campaign, Gift or Pledge, Category	Hours, Estimated Value		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
<input checked="" type="checkbox"/>	Matching Donor ID			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		

This activates NAV's **SIFT (Sum Index Flow Technology)** feature for those keys and SumIndexFields. What we have defined will allow us to access **Estimated Value** or **Hours** by Donor or by Campaign. We could do the same thing for any key structure that we want to define and maintain. Our example also illustrates it does not matter in what sequence the SumIndexFields appear.

Some criteria that do matter are the number of SumIndexFields and the relationship of a SumIndexField with the key to which it is associated. More SumIndexFields equals more processing required to maintain them. The fields in the SIFT key that provide the greatest number of unique values for the data should be closest to the left (beginning) of the key for better performance.

If the **MaintainSIFTIndex** is not checked, then SQL Server will calculate the SIFT total on demand by passing the base data. This is a good choice for a SIFT total that will be used only occasionally.

There is a processing cost for every additional key or SumIndexField. Each must either be maintained every time their table is updated or they must be constructed when there is a request for data retrieval in which they are involved.

Table integration

Now that we have put together the supporting tables, we can update the Donor table to integrate with these and begin to take advantage of the structure that we are building.

Table 50000 Donor - Table Designer

E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>	10	Donor ID	Code	20	
<input checked="" type="checkbox"/>	20	Name	Text	50	
<input checked="" type="checkbox"/>	30	Address	Text	50	
<input checked="" type="checkbox"/>	40	Address 2	Text	50	
<input checked="" type="checkbox"/>	50	City	Text	30	
<input checked="" type="checkbox"/>	60	State/Province	Text	10	
<input checked="" type="checkbox"/>	70	Post Code	Code	20	
<input checked="" type="checkbox"/>	80	Country/Region Code	Code	10	
<input checked="" type="checkbox"/>	1000	Phone No.	Text	30	PN.01
<input checked="" type="checkbox"/>	1010	Alt. Phone No.	Text	30	PN.01
<input checked="" type="checkbox"/>	1020	E-mail	Text	80	PN.01
<input checked="" type="checkbox"/>	1030	Donor Type	Code	10	PN.01
<input checked="" type="checkbox"/>	1040	Contact Method	Option		PN.01
<input checked="" type="checkbox"/>	1050	Solicitation OK	Boolean		PN.01
<input checked="" type="checkbox"/>	1060	Recognition Level	Option		PN.01
<input checked="" type="checkbox"/>	1070	Max Recognition Level	Option		PN.01
<input checked="" type="checkbox"/>	1080	Matching Gift Donor	Code	20	PN.01
<input checked="" type="checkbox"/>	1090	Newsletter	Boolean		PN.01
<input checked="" type="checkbox"/>	1100	Date Added	Date		PN.01
<input checked="" type="checkbox"/>	1110	Status	Option		PN.01
<input checked="" type="checkbox"/>	1120	Alpha Name	Text	50	PN.01
<input checked="" type="checkbox"/>	1130	Volunteer	Boolean		PN.01
<input checked="" type="checkbox"/>	1200	Estimated Gift Value	Decimal		PN.02
<input checked="" type="checkbox"/>	1201	Hours Volunteered	Decimal		PN.02

Help

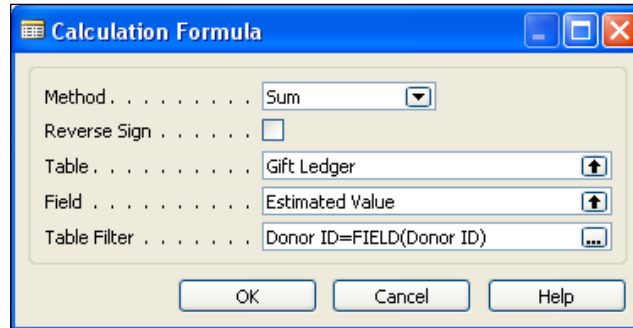
In the preceding screenshot, the **Description** column for the objects entered contains a Version tracking code. A **Description** value of **PN.01**, **PN.02**, and so on indicates that a field was added in modification one and then added or changed in modification two. The goal is to keep track of the circumstances under which the object is modified, and to identify what the current update level is for each object. In this case, we have chosen a version code consisting of **PN** (for **Programming NAV**) and a two-digit number referring to a modification instance when the object is created or changed.

You can make up your own version identification codes, but you should be consistent in their use. You may prefer to follow the general conventions used by Microsoft in the product by assigning a sequence of major and minor codes to identify the organization, the subsystem, and the modification version. Version codes should be tied to comments inside the objects and should allow you to maintain external documentation describing the purpose of various modifications and enhancements.

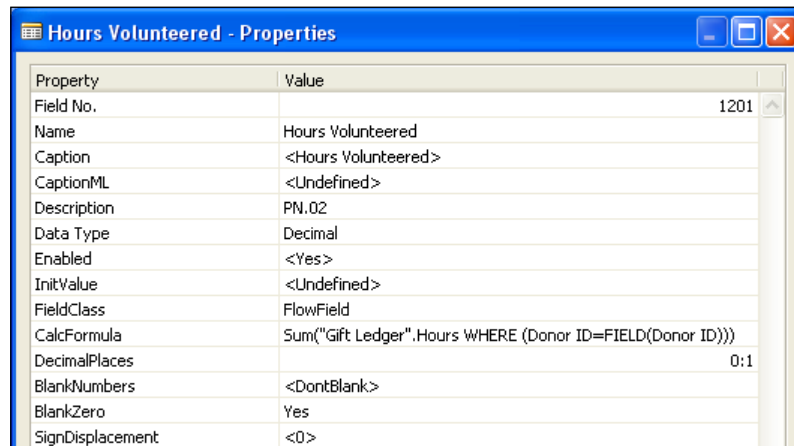
Our two new fields in the preceding screenshot are FlowFields. Now let us take a quick look at each of them individually. The following screenshot shows the **Properties** for the **Estimated Gift Value** FlowField.

Property	Value
Field No.	1200
Name	Estimated Gift Value
Caption	<Estimated Gift Value>
CaptionML	<Undefined>
Description	PN.02
Data Type	Decimal
Enabled	<Yes>
InitValue	<Undefined>
FieldClass	FlowField
CalcFormula	Sum("Gift Ledger", "Estimated Value" WHERE (Donor ID=FIELD(Donor ID)))
DecimalPlaces	<Undefined>
BlankNumbers	<DontBlank>
BlankZero	<No>
SignDisplacement	<0>
AutoFormatType	<0>
AutoFormatExpr	<>
CaptionClass	<>
Editable	<Yes>
MinValue	<>
MaxValue	<>
NotBlank	<No>
ValuesAllowed	<>
TableRelation	<Undefined>
ValidateTableRelation	<Yes>
ExtendedDatatype	<None>

The following screenshot illustrates how a FlowField is defined. When you click on the **CalcFormula** property, an ellipsis icon will appear and clicking on it will give you the **Calculation Formula** screen. In the following screenshot, you can see how the values appearing in the preceding screenshot **CalcFormula** property were originally defined.



In the following screenshot, the Hours Volunteered FlowField is shown:



Each of these FlowFields is a **Sum** and each of them is the sum of the data **WHERE (Donor ID = FIELD(Donor ID))**. In other words, this is a summation of all the applicable data belonging to the Donor being processed.

The other properties that you should note here are the **DecimalPlaces** and **BlankNumbers** properties. They have been set to control the display of the data, so that it will appear in a format other than that which would result from the default values. You should adjust them to whatever values your subjective sense of visual design dictates. But remember, consistency with other parts of the system should be an overriding criterion for the design of an enhancement.

Types of tables

For our discussion, we will break the table types into three groups. As a developer, you can change the definition and the contents of the first group (that is Wholly Modifiable Tables). You cannot change the definition of the second group, but you can change the contents (that is content-modifiable Tables). The third group can be accessed for information, but neither the definition nor the data within is modifiable (that is, these tables are read-only tables).

Wholly modifiable tables

The following are the tables in the Wholly Modifiable Tables group:

Master

The **Master** table type contains primary data (such as Customers, Vendors, Items, Employees, and so on.). These are the tables that should be designed first. If you consider the data definition as the foundation of the system, the **Master** tables are the footings providing a stable base for that foundation. When working on a modification, any necessary changes to **Master** tables should be defined first. **Master** tables always use card forms/pages as their primary user input method. The **Customer** table is a **Master** table. A **Customer** record is shown in the following screenshot:

Edit - Customer Card - 10000 - The Cannon Group PLC

Actions | Related Information | Reports

Sales Invoice | Apply Template | Statistics | Customer - Balance to Date

Sales Order | Cash Receipt Journal

Reminder | Sales Journal

New | Process | Reports

10000 - The Cannon Group PLC

General

No.: 10000

Name: The Cannon Group PLC

Address: 192 Market Square

Address 2:

Post Code: B27 4KT

City: Birmingham

Country/Region Code: GB

Phone No.:

Primary Contact No.:

Contact: Mr. Andy Teal

Search Name: THE CANNON GROUP PLC

Balance (LCY): 168,364.41

Credit Limit (LCY): 0.00

Salesperson Code: PS

Responsibility Center: BIRMINGHAM

Service Zone Code: M

Blocked:

Last Date Modified: 11/5/2008

Communication

Phone No.:

Fax No.:

E-Mail: the.cannon.group.plc@cronu...

Home Page:

IC Partner Code:

Invoicing

NATIONAL | DOMESTIC

Payments

1M(8D) | DOMESTIC | 1.5 DOM.

Shipping

BLUE | Partial | EXW | DHL

Foreign Trade

Customer Sales History

Customer No.:	10000
Quotes:	0
Blanket Orders:	0
Orders:	4
Invoices:	0
Return Orders:	0
Credit Memos:	0
Pstd. Shipments:	6
Pstd. Invoices:	3
Pstd. Return Re...:	1
Pstd. Credit Me...:	1

Customer Statistics - ...

Customer No.:	10000
Balance (LCY):	168,364.41
Outstanding ...:	1,612.50
Shipped Not ...:	525.50
Outstanding ...:	6.63
Shipped Not ...:	525.50
Outstanding ...:	0.00
Total (LCY):	170,502.41
Credit Limit (...):	0.00
Overdue Am...:	-292.84
Sales YTD (L...):	17,100.96

Links

Notes

OK

The preceding screenshot shows how the Card page segregates the data into categories on different FastTabs (for example, **General**, **Communications**, **Invoicing**, and so on.) and includes primary data fields (for example, **No.**, **Name**, **Address**), reference fields (for example, **Salesperson Code**, **Responsibility Center**), and a FlowField (for example, **Balance (LCY)**).

Journal

The Journal table type contains unposted activity detail, data that other systems refer to as "transactions". Journals are where most repetitive data entry occurs. The standard system design has all Journal tables matched with corresponding Template tables (that is a Template table for each Journal table). The standard system includes journals for Sales, Cash Receipts, General Journal entries, Physical Inventory, Purchases, Fixed Assets, and Warehouse Activity, among others.

The transactions in a Journal can be segregated into batches for entry, edit review, and processing purposes. Journal tables always use list pages as their primary user input method. The following screenshots shows two Journal Entry screens. They both use the General Journal table, but appear quite different from each other, with different pages and different templates (templates are explained in the following section).

Posting Date	Document ...	Document ...	Account Type	Account No.	Description	Amount	Bal. Accou...	Bal. Accou...	Applies-to ...	Applies-to ...
1/28/2010	Payment	G02001	Customer	10000	The Cannon Group PLC	-250.00	Bank Account	WWB-EUR	Invoice	

Account Name: The Cannon Group PLC
 Bal. Account Name: World Wide Bank
 Balance: 0.00
 Total Balance: 0.00

Comparing the preceding and following screenshots, the differences not only represent what fields are made visible, but also the logic that applies to data entry.

Batch Name: DEFAULT

Posting Date	Document ...	Document ...	Account Type	Account No.	Description	Gen. Postin...	Gen. Bus. ...	Gen. Prod. ...	Amount
1/28/2010	Invoice	G01001	Customer	10000	The Cannon Group PLC				1,400.00

Account Name: The Cannon Group PLC Bal. Account Name: Balance: 1,400.00 Total Balance: 1,400.00

Template

The Template table type operates behind the scenes, providing control information for a Journal, which operates in the foreground. By use of a Template, multiple instances of a Journal can be tailored for different purposes. Control information contained by a Template includes the following:

- The default type of accounts to be affected (for example, Customer, Vendor, Bank, General Ledger)
- The specific account numbers to be used as defaults, including balancing accounts
- What transaction numbering series will be used
- Default encoding to be applied to transactions for this Journal (for example, Source Code, Reason Code)
- Specific Pages and Reports to be used for data entry and processing of both edits and posting runs

As an example, **General Journal Templates** allow the General Journal table to be tailored in order to display fields and perform validations that are specific to the entry of particular transaction categories, for example, Cash Receipts, Payments, Purchases, Sales, and other transaction entry types. Template tables always use tabular pages for user input. The following screenshot shows a listing of the various General Journal Templates defined in the Cronus International Ltd. demonstration database.

Posting Date	Document	Document No.	Customer No.	Description	Currency	Original Amount	Amount	Remaining Amount	Duration
1/10/2010	Invoice	103005	10000	Order 101001		8,269.04	8,269.04	8,269.04	2/7
1/17/2010	Credit Memo	104001	10000	Credit Memo 104001		-292.84	-292.84	-292.84	1/1
1/17/2010	Payment	2596	10000	Payment 2010		-25,389.25	-25,389.25	0.00	1/1
1/17/2010	Payment	2596	10000	Payment 2010		-50,778.50	-50,778.50	0.00	1/1
1/17/2010	Payment	2596	10000	Payment 2010		-67,704.67	-67,704.67	0.00	1/1
1/25/2010	Invoice	103001	10000	Invoice 103001		8,182.35	8,182.35	8,182.35	2/2
1/20/2010	Invoice	103018	10000	Order 6005		4,101.88	4,101.88	4,101.88	2/2

Ledger

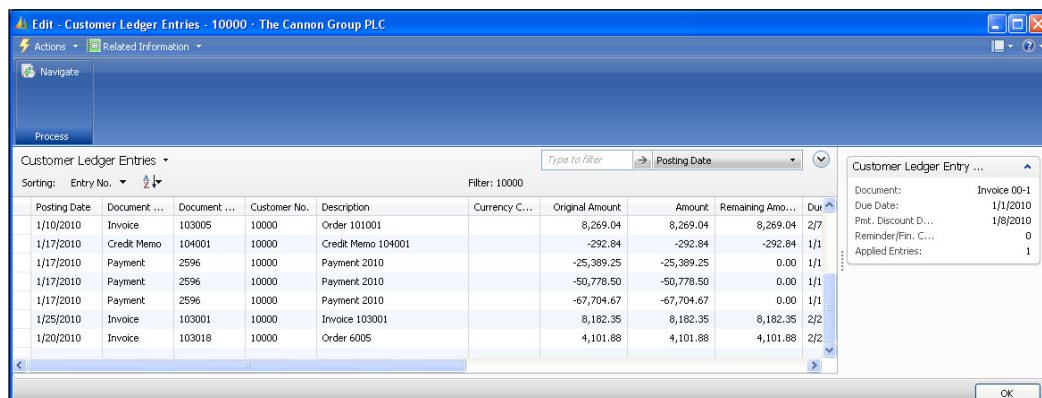
The Ledger table type contains posted activity detail, the data other systems call *history*. The basic data flow is from a Journal through a Posting routine and into a Ledger. A distinct advantage of the way NAV Ledgers are designed is the fact that they allow the retention of all detail indefinitely.

While there are protocols and supporting routines to allow compression of the Ledger data (that is summarization), as long as your system has sufficient disk space, you can (and should) always keep the full historical detail of all activity. This allows users to have total flexibility for historical data analysis.

Most other systems require some type of periodic summarization of data (for example, by accounting period, by month, by year). That summarization into periodic totals (sometimes called buckets) constrains the ways in which historical data analysis can be done. By allowing the retention of historical data in full detail, the NAV System Designer is allowed to be less visionary, because future analytical functionality can still take advantage of this detail. User views of Ledger data are generally through use of List pages. In the end, the NAV approach of long-term data retention in complete detail lets users get as much value out of their data as possible.

Ledger data is considered **accounting data** in NAV. That means you are not allowed to directly enter the data into a Ledger, but must "Post" to a Ledger. Although you can physically force data into a Ledger with your Developer tools, you should not do so. As it is accounting data, it also means that you are not allowed to delete data from a Ledger table; you can compress or summarize data using the provided compression routines, thus eliminating a level of detail, but you cannot eliminate anything that would affect accounting totals for money or quantities.

The following screenshots show a **Customer Ledger Entries** list (financially oriented data) and an **Item Ledger Entries** list (quantity-oriented data). In each case, the data represents historical activity detail with accounting significance. There are other data fields in addition to those shown in the following screenshots. The fields shown are typical and representative. The users can utilize page-customization tools (which we will discuss in the chapter on Pages) in order to change personalized page displays in a wide variety of ways. Here is the **Customer Ledger Entries** list screenshot:



Posting Date	Document No.	Document Type	Customer No.	Description	Currency	Original Amount	Amount	Remaining Amount	Duration
1/10/2010	Invoice	103005	10000	Order 101001		8,269.04	8,269.04	8,269.04	2/7
1/17/2010	Credit Memo	104001	10000	Credit Memo 104001		-292.84	-292.84	-292.84	1/1
1/17/2010	Payment	2596	10000	Payment 2010		-25,389.25	-25,389.25	0.00	1/1
1/17/2010	Payment	2596	10000	Payment 2010		-50,778.50	-50,778.50	0.00	1/1
1/17/2010	Payment	2596	10000	Payment 2010		-67,704.67	-67,704.67	0.00	1/1
1/25/2010	Invoice	103001	10000	Invoice 103001		8,182.35	8,182.35	8,182.35	2/2
1/20/2010	Invoice	103018	10000	Order 6005		4,101.88	4,101.88	4,101.88	2/2

The following is a screenshot of the **Item Ledger Entries** list:



Posting Date	Entry Type	Document No.	Document Type	Item No.	Description	Location Code	Quantity	Invoiced Quantity	Remaining Quantity	Sales Amount (Actual)	Cost Amount (Actual)	Cost Amount (Non-Invent.)	Open
12/31/2009	Positive Adj...		START	1900-S		BLUE	52	52	52	0.00	4,816.50	0.00	✓
12/31/2009	Positive Adj...		START	1900-S		RED	46	46	46	0.00	4,260.75	0.00	✓
12/31/2009	Positive Adj...		START	1900-S		GREEN	47	47	41	0.00	4,353.37	0.00	✓
1/15/2010	Sale	Sales Ship...	102009	1900-S		GREEN	-6	-6	0	750.60	-585.00	0.00	✓
1/17/2010	Purchase	Purchase ...	107021	1900-S		YELLOW	160	160	160	0.00	15,600.00	0.00	✓


In the **Customer Ledger Entries** page, you can see critical information such as **Posting Date** (the effective accounting date), **Document Type** (the type of transaction), **Customer No.**, the **Original** and **Remaining Amount** of the transaction, and (if it were visible, which it is not) **Entry No.**, which uniquely identifies each record. The **Open** entries are those where the transaction amount has not been fully applied, such as an Invoice amount not fully paid or a Payment amount not fully consumed by Invoices.

In the **Item Ledger Entries** page, you can see similar information pertinent to inventory transactions. As previously described, **Posting Date**, **Entry Type**, and **Item No.**, as well as the assigned **Location** for the Item, control the meaning of each transaction. Item Ledger Entries are expressed both in **Quantity** and **Amount** (Value). **Open** entries here are tied to the **Remaining Quantity**, such as, material that has been received but not yet fully shipped out. In other words, the **Open** entries represent current inventory.

Reference

The Reference table type contains lists of codes as well as other validation and interpretation reference data that is used (referred to) by many other table types. Reference table examples are postal zone codes, country codes, currency codes, exchange rates, and so on. Reference tables are often accessed under the **Setup** menu options as they must be set up prior to being used for reference purposes by other tables.

The following screenshots show some sample reference tables for **Locations**, **Countries**, and **Payment Terms**. Each table contains data elements that are appropriate to its use as a reference table, plus, in some cases, fields that control the effect of referencing a particular entry. These data elements are usually entered as a part of a setup process and then updated occasionally as appropriate – that is, they generally do not contain data originating from system activity.

Locations ▾	
Sorting:	Code ▾ 
Code	Name
BLUE	Blue Warehouse
GREEN	Green Warehouse
OUT. LOG.	Outsourced Logis...
OWN LOG.	Own Logistics
RED	Red Warehouse
SILVER	Silver Warehouse
WHITE	White Warehouse
YELLOW	Yellow Warehouse

The **Location List** in the preceding screenshot is a simple validation list of the Locations for this implementation. Usually, they represent physical sites, but depending on the implementation, they can also be used simply to segregate types of inventory. For example, locations could be Refrigerated versus Unrefrigerated or there could be a location for Failed Inspection.

Countries/Regions ▾ Type to filter				
Sorting: Code ▾ A Z		No filters applied		
Code	Name	Address Format	Contact Address Format	
AE	United Arab Emirates	City+Post Code	After Company Name	
AT	Austria	Post Code+City	After Company Name	
AU	Australia	City+County+P...	After Company Name	
BE	Belgium	Post Code+City	After Company Name	
BG	Bulgaria	City+County+P...	After Company Name	
BN	Brunei Darussalam	City+Post Code	First	
BR	Brazil	City+Post Code	First	
CA	Canada	City+Post Code	After Company Name	
CH	Switzerland	Post Code+City	After Company Name	

The **Countries/Regions** list in the preceding screenshot acts as validation data, controlling what Country Code is acceptable. It also provides control information for the mailing **Address Format** (general organization address) and the **Contact Address Format** (for the individual).

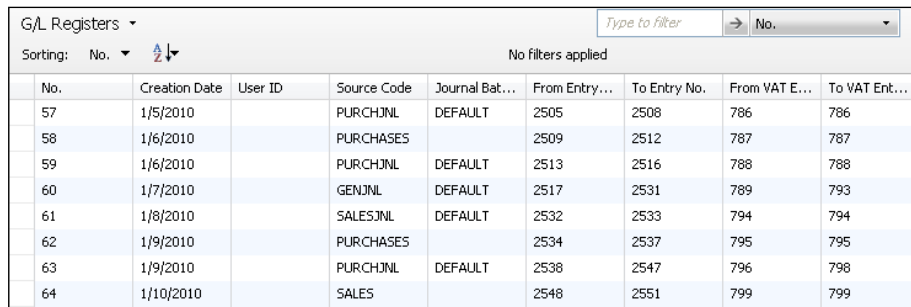
The **Payment Terms** table shown in the following screenshot provides a list of payment terms codes along with a set of parameters that allows the system to calculate specific terms. In this set of data, for example, the **1M (8D)** code will yield payment terms of 1 month with a discount of **2%** applied for payments processed within 8 days of the invoice date. In another instance, **14D** payment terms will calculate the payment as due in 14 days from the date of invoice with no discount available.

Payment Terms ▾ Type to filter → Code						
Sorting: Code ▾ A Z		No filters applied				
Code	Due Date Calculation	Discount Date Calculation	Discount %	Calc. Pmt. Disc. on Cr. Memos	Description	
14 DAYS	14D		0	<input checked="" type="checkbox"/>	Net 14 days	
1M(8D)	1M	8D	2	<input type="checkbox"/>	1 Month/2% 8 days	
21 DAYS	21D		0	<input type="checkbox"/>	Net 21 days	
7 DAYS	7D		0	<input type="checkbox"/>	Net 7 days	
CM	CM		0	<input type="checkbox"/>	Current Month	
COD	0D		0	<input type="checkbox"/>	Cash on delivery	

Register

The Register table type contains a record of the range of transaction ID numbers for each batch of posted Ledger entries. Register data provides an audit of the physical timing and sequence of postings. This, combined with the full detail retained in the Ledger, makes NAV a very auditable system, that is, you can see exactly what activity was done and when it was done.

Another NAV feature, the Navigate function, which we will discuss in detail later, also provides a very useful auditing tool. The Navigate function allows the user (who may be a developer doing testing) to highlight a single Ledger entry and find all the other Ledger entries and related records that resulted from the posting that created that highlighted entry. The user views the Register through a tabular page, as shown in the following screenshot. You can see that each Register entry has the **Creation Date**, **Source Code**, **Journal Batch Name**, and the identifying **Entry No.** range for all the entries in that batch.



No.	Creation Date	User ID	Source Code	Journal Bat...	From Entry...	To Entry No.	From VAT E...	To VAT Ent...
57	1/5/2010		PURCHJNL	DEFAULT	2505	2508	786	786
58	1/6/2010		PURCHASES		2509	2512	787	787
59	1/6/2010		PURCHJNL	DEFAULT	2513	2516	788	788
60	1/7/2010		GENJNL	DEFAULT	2517	2531	789	793
61	1/8/2010		SALESJNL	DEFAULT	2532	2533	794	794
62	1/9/2010		PURCHASES		2534	2537	795	795
63	1/9/2010		PURCHJNL	DEFAULT	2538	2547	796	798
64	1/10/2010		SALES		2548	2551	799	799

Posted Document

The Posted Document type contains the posted copies of the original documents for a variety of data types such as Sales Invoices, Purchase Invoices, Sales Shipments, and Purchase Receipts. Posted Documents are designed to provide an easy reference to the historical data in a format similar to what one would normally store in paper files. A Posted Document looks very similar to the original source document. In essence, a posted invoice will look very similar to the original Sales Order or Sales Invoice. The Posted Documents are included in the **Navigate** function.

The following screenshots show a **Sales Order** before Posting and the resulting **Posted Sales Invoice** document. Both documents are in a Header/Detail format, where the information in the Header applies to the whole order and the information in the Detail is specific to the individual Order Line. As part of the **Sales Order** page, there is information displayed to the right of the actual order. This is designed to make life easier for the user by giving clues to the related data which is available without a separate lookup action.

First, we see the **Sales Order** document ready to be Posted.

101016 · The Cannon Group PLC

General	
No.:	101016
Sell-to Customer No.:	10000
Sell-to Contact No.:	
Sell-to Customer Name:	The Cannon Group PLC
Sell-to Address:	192 Market Square
Sell-to Address 2:	
Sell-to Post Code:	B27 4KT
Sell-to City:	Birmingham
Sell-to Contact:	Mr. Andy Teal
No. of Archived Versions:	0
Posting Date:	2/3/2010
Order Date:	1/28/2010
Document Date:	1/28/2010
Requested Delivery Date:	
Promised Delivery Date:	
Quote No.:	
External Document No.:	
Salesperson Code:	PS
Campaign No.:	
Opportunity No.:	
Responsibility Center:	
Assigned User ID:	
Status:	Released

Customer Sales History -...	
Customer No.:	10000
Quotes:	0
Blanket Orders:	0
Orders:	4
Invoices:	0
Return Orders:	0
Credit Memos:	0
Pstd. Shipments:	6
Pstd. Invoices:	3
Pstd. Return Receipts:	1
Pstd. Credit Memos:	1

Sales Line Details	
Item No.:	1920-5
Availability:	3
Substitutions:	0
Sales Prices:	0
Sales Line Discounts:	0

Notes	

Lines						
Type	No.	Description	Location C...	Quantity	Reserved Qua...	Unit of
Item	1920-5	ANTWERP Conference Table	RED	1		PCS

Invoicing 10000 1M(8D) 2/28/2010

Shipping B27 4KT 1/28/2010 Partial

Foreign Trade

E - Commerce

The following screenshot is that of the **Sales Invoice** document after the **Sales Order** has been posted.

103022 · The Cannon Group PLC

General	
No.:	103022
Sell-to Customer No.:	10000
Sell-to Contact No.:	
Sell-to Customer Name:	The Cannon Group PLC
Sell-to Address:	192 Market Square
Sell-to Address 2:	
Sell-to Post Code:	B27 4KT
Sell-to City:	Birmingham
Sell-to Contact:	Mr. Andy Teal
Posting Date:	2/3/2010
Document Date:	1/28/2010
Quote No.:	
Order No.:	101016
Pre-Assigned No.:	
External Document No.:	
Salesperson Code:	PS
Responsibility Center:	
No. Printed:	0

Lines						
Type	No.	Description	Quantity	Unit of Mea...	Unit Price Excl...	Line Amou
Item	1920-5	ANTWERP Conference Table	1	PCS	420.40	

Invoicing 10000 1M(8D) 2/28/2010

Shipping B27 4KT 1/28/2010

Foreign Trade

BizTalk

Setup

The Setup table type contains system or functional application control information. There is one Setup table per functional application area, for example, one for Sales & Receivables, one for Purchases & Payables, one for General Ledger, one for Inventory, and so on. Setup tables contain only a single record. As a Setup table only has a single record, it can have a null value Primary Key field (this is the way all of the standard NAV Setup tables are designed).

Edit - Sales Receivables Setup

Actions

Sales & Receivables Setup

General

Discount Posting:	All Discounts	Copy Comments Blanket to Order:	<input checked="" type="checkbox"/>
Credit Warnings:	Both Warnings	Copy Comments Order to Invoice:	<input checked="" type="checkbox"/>
Stockout Warning:	<input checked="" type="checkbox"/>	Copy Comments Order to Shpt.:	<input checked="" type="checkbox"/>
Shipment on Invoice:	<input checked="" type="checkbox"/>	Copy Cmts Ret.Ord. to Cr. Memo:	<input checked="" type="checkbox"/>
Return Receipt on Credit Memo:	<input checked="" type="checkbox"/>	Copy Cmts Ret.Ord. to Ret.Rcpt:	<input checked="" type="checkbox"/>
Invoice Rounding:	<input checked="" type="checkbox"/>	Allow VAT Difference:	<input type="checkbox"/>
Ext. Doc. No. Mandatory:	<input type="checkbox"/>	Calc. Inv. Discount:	<input type="checkbox"/>
Appln. between Currencies:	All	Calc. Inv. Disc. per VAT ID:	<input type="checkbox"/>
Logo Position on Documents:	No Logo	Exact Cost Reversing Mandatory:	<input type="checkbox"/>
Default Posting Date:	Work Date	Check Preprint, when Posting:	<input type="checkbox"/>
Default Quantity to Ship:	Remainder	Archive Quotes and Orders:	<input type="checkbox"/>

Dimensions

Customer Group Dimension Code:	CUSTOMERGROUP	Salesperson Dimension Code:	SALESPERSON
--------------------------------	---------------	-----------------------------	-------------

Numbering

Customer Nos.:	CUST	Posted Shipment Nos.:	S-SHPT
Quote Nos.:	S-QUO	Posted Return Receipt Nos.:	S-RCPT
Blanket Order Nos.:	S-BLK	Reminder Nos.:	S-REM
Order Nos.:	S-ORD-1	Issued Reminder Nos.:	S-REM+
Return Order Nos.:	S-RETORD	Fin. Chrg. Memo Nos.:	S-FIN
Invoice Nos.:	S-INV	Issued Fin. Chrg. M. Nos.:	S-FIN+
Posted Invoice Nos.:	S-INV+	Posted Preprint, Inv. Nos.:	S-INV+
Credit Memo Nos.:	S-CR	Posted Preprint, Cr. Memo Nos.:	S-CR+
Posted Credit Memo Nos.:	S-CR+		

OK

Temporary

The Temporary table type is used within objects to hold temporary copies of data. A Temporary table is defined within an object as a variable using a permanent table as the template. That means a Temporary table will have exactly the same data structure as the permanent table after which it is modeled, but with a limited subset of other attributes.

Temporary tables are created empty when the parent object execution initiates, and they disappear along with their data when the parent object execution terminates (that is, when the Temporary table variable goes out of scope). The data in a Temporary table resides in the client system and not in the system database. This provides faster processing because all processing is local.

Temporary tables are not directly visible or accessible to users. They cannot directly be the target of a report object, but can be the primary source table for forms and pages. Temporary tables are intended to be work areas and as such, are containers of data. The definition of a Temporary table can only be changed by changing the definition of the permanent table on which it has been modeled.

Content-modifiable tables

There is only one table type included in the content-modifiable table group.

System

The System table type contains user-maintainable information that pertains to the management or administration of the NAV application system. System tables are created by NAV. You cannot create System tables as they affect the underlying NAV executables. However, with full developer license rights, you can modify these System tables to extend their usage. With full system permissions, you can also change the data in System tables.

An example is the User table, which contains user login information. This particular System table is often modified to define special user access routing or processing limitations. Other System tables contain data on report-to-printer routing assignments, transaction numbers to be assigned, batch job scheduling, and so on. The following are examples of System tables in which definition and content can be modified. The first six relate to system security functions:

1. **User:** The table of identified users and their login password for the Database Server access method
2. **Member Of:** This contains User Security Role information

3. **User Role:** This contains the defined User Security Roles available. Each User Role is made up of a group of individual object permissions : Read, Insert, Modify, Delete, and Execute permissions.
4. **Permission:** The table defining what individual User roles are allowed to do, based on object permission assignments
5. **Windows Access Control:** The table of the Security roles that are assigned to each Windows Login
6. **Windows Login:** The table for Windows Logins that have been created for this database

The following tables are used to track a variety of system data or control structures:

- **Company:** The companies in this database. Most NAV data is automatically segregated by Company
- **Database Key Groups:** This defines all of the key groups that have been set up to allow enabling and disabling table keys
- **Chart:** This defines all of the chart parts that have been set up for use in constructing pages
- **Web Service:** This lists the pages and code units that have been published as web services
- **Profile:** This contains a list of all the active profiles and their associated Role Center pages
- **User Personalization:** This contains information about user personalization that has occurred

The following tables contain information about various system internals. Their explanation is outside the scope of this book:

- **User Menu Level**
- **Send-to Program**
- **Style Sheet**
- **User Default Style Sheet**
- **Record Link**
- **Object Tracking**
- **Object Metadata**
- **Profile Metadata**
- **User Metadata**

Read-Only tables

There is only one table type included in the read-only table group.

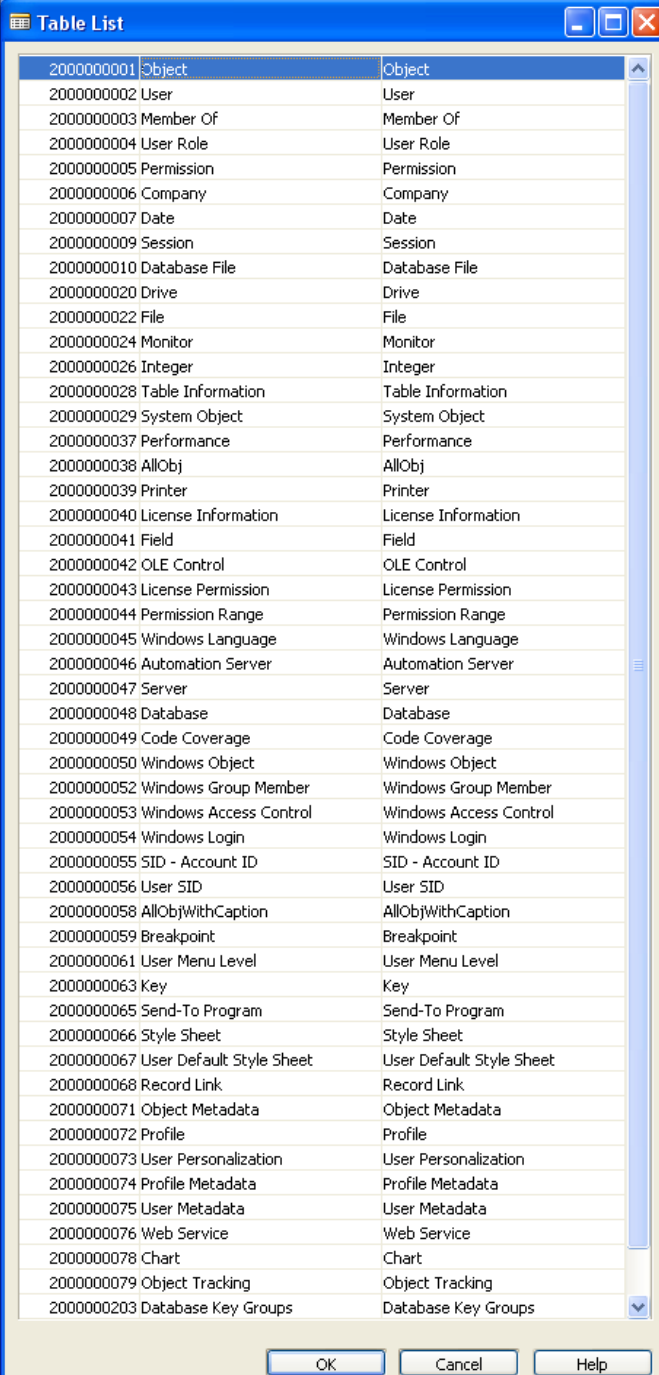
Virtual

The Virtual table type is computed at runtime by the system. A Virtual table contains data and is accessed like other tables, but you cannot modify either the definition or the contents of a Virtual table. Some of these tables (such as the Database File, File, and Drive tables) provide access to information about the computing environment. Other Virtual tables (such as Table Information, Field, and Monitor tables) provide information about the internal structure and operating activities of your database.

Some virtual tables (such as Date and Integer) provide tools that can be used in your application routines. The Date table provides a list of calendar periods (such as days, weeks, months, quarters, and years) to make it much easier to manage various types of accounting and managerial data handling. The Integer table provides a list of integers from -1,000,000,000 to +1,000,000,000. As you explore standard NAV reports, you will frequently see the Integer table being used to supply a sequential count in order to facilitate a reporting sequence.

You cannot see these tables presented in the List of Table objects, but can only access them as targets for Forms/Pages, Reports, or Variables in C/AL code. The knowledge of the existence, contents, and usage of these Virtual tables is not useful to an end user. However, as a developer, you will regularly use some of the Virtual tables. You may find educational value in studying the structure and contents of these tables, as well as be able to create valuable tools with your knowledge of and accessing of one or more Virtual tables.

The following screenshot shows a list of many Virtual and System tables:



The screenshot shows a window titled "Table List" with a list of tables. The list is organized into two columns: "Object" and "Table". The first column contains a numerical ID and the object name, while the second column contains the table name. The list includes various system and virtual tables, such as "Object", "User", "Member Of", "User Role", "Permission", "Company", "Date", "Session", "Database File", "Drive", "File", "Monitor", "Integer", "Table Information", "System Object", "Performance", "AllObj", "Printer", "License Information", "Field", "OLE Control", "License Permission", "Permission Range", "Windows Language", "Automation Server", "Server", "Database", "Code Coverage", "Windows Object", "Windows Group Member", "Windows Access Control", "Windows Login", "SID - Account ID", "User SID", "AllObjWithCaption", "Breakpoint", "User Menu Level", "Key", "Send-To Program", "Style Sheet", "User Default Style Sheet", "Record Link", "Object Metadata", "Profile", "User Personalization", "Profile Metadata", "User Metadata", "Web Service", "Chart", "Object Tracking", and "Database Key Groups".

Object	Table
2000000001 Object	Object
2000000002 User	User
2000000003 Member Of	Member Of
2000000004 User Role	User Role
2000000005 Permission	Permission
2000000006 Company	Company
2000000007 Date	Date
2000000009 Session	Session
2000000010 Database File	Database File
2000000020 Drive	Drive
2000000022 File	File
2000000024 Monitor	Monitor
2000000026 Integer	Integer
2000000028 Table Information	Table Information
2000000029 System Object	System Object
2000000037 Performance	Performance
2000000038 AllObj	AllObj
2000000039 Printer	Printer
2000000040 License Information	License Information
2000000041 Field	Field
2000000042 OLE Control	OLE Control
2000000043 License Permission	License Permission
2000000044 Permission Range	Permission Range
2000000045 Windows Language	Windows Language
2000000046 Automation Server	Automation Server
2000000047 Server	Server
2000000048 Database	Database
2000000049 Code Coverage	Code Coverage
2000000050 Windows Object	Windows Object
2000000052 Windows Group Member	Windows Group Member
2000000053 Windows Access Control	Windows Access Control
2000000054 Windows Login	Windows Login
2000000055 SID - Account ID	SID - Account ID
2000000056 User SID	User SID
2000000058 AllObjWithCaption	AllObjWithCaption
2000000059 Breakpoint	Breakpoint
2000000061 User Menu Level	User Menu Level
2000000063 Key	Key
2000000065 Send-To Program	Send-To Program
2000000066 Style Sheet	Style Sheet
2000000067 User Default Style Sheet	User Default Style Sheet
2000000068 Record Link	Record Link
2000000071 Object Metadata	Object Metadata
2000000072 Profile	Profile
2000000073 User Personalization	User Personalization
2000000074 Profile Metadata	Profile Metadata
2000000075 User Metadata	User Metadata
2000000076 Web Service	Web Service
2000000078 Chart	Chart
2000000079 Object Tracking	Object Tracking
2000000203 Database Key Groups	Database Key Groups

Summary

In this chapter, we have focused on the top level of NAV data structure: tables and their internal structure. We worked our way through the hands-on creation of a number of tables and their data definitions in support of our ICAN application. We briefly discussed Field Groups and how they are used.

We also reviewed most of the types of tables found in the out of the box NAV application. Finally, we identified the essential table structure elements including Properties, Object Numbers, Triggers, Keys, and SumIndexFields.

In the next chapter, we will dig deeper into the NAV data structure to understand how fields and their attributes are assembled to make up the tables. We will also focus more deeply on what can be done with Triggers. Then, we will explore using tables in other object types, heading towards obtaining a full toolkit to perform NAV development.

Review questions

1. Which of the following is a correct description of a table in NAV 2009?
Choose three:
 - a. A NAV table is the definition of data structure
 - b. A NAV table contains data but is not data
 - c. A NAV table can contain C/AL code
 - d. A NAV table only incidentally affects the business rules of a system
2. There is no practical limit to the number of fields or the total records size in a table? True or False?
3. Table numbers intended to be used for customized table objects should only range between 50000 to 59999. True or False?
4. Which of the following are Table triggers?
 - a. OnInsert
 - b. OnChange
 - c. OnDelete
 - d. OnRename
5. Under some circumstances, the primary key in a NAV table does not require unique entries. True or False?
6. NAV table design in the product always insures referential integrity for all parent—child table relationships. True or False?
7. Keys can be enabled or disabled in executable code. This is sometimes very valuable for managing system performance. True or False?
8. SumIndexFields, FlowFields, and SIFT are all closely related terms. True or False?
9. Field Groups defined in tables provide the structure for DropDown displays used in data entry pages. Field Groups can be modified. True or False?
10. Even though the target development is the Role Tailored Client, sometimes it is useful to create Classic Client forms for various testing purposes. True or False?

11. Whether the user is working in the Role Tailored Client or the Classic Client, all status and user tailoring information is retained in the ZUP file. True or False?
12. Which of the following tables can be modified by Partner developers? Choose three.
 - a. Customer
 - b. Date
 - c. User
 - d. Item Ledger Entry
13. Reports cannot use temporary tables as Data Items. True or False?
14. Tables can be created or deleted dynamically similar to the way that files external to the database can be created or deleted dynamically. True or False?
15. SQL Server for NAV has been enhanced to include native support for SIFT functionality. True or False?
16. The NAV database design is not a fully normalized design. Even though more data storage may be consumed, this approach adds to the efficiency of processing. True or False?

3

Data Types and Fields for Data Storage and Processing

Technical skill is mastery of complexity, while creativity is mastery of simplicity – E. Christopher Zeeman

The unavoidable price of reliability is simplicity – C.A.R. Hoare

The design of an application starts at the simplest level, with the data. The data design greatly depends on the types of data your development tool set allows you to use. Since NAV is designed specifically to develop financially-oriented business applications, NAV data types are financially and business oriented.

In this chapter, we will cover data types that you are most likely to use, plus several less frequently used. We will cover many of the field properties with emphasis on field classes—a key property which affects whether the contents of the field are data or information to be interpreted.

Basic definitions

Let's define some basic NAV terminologies to get started:

- **Data type:** This describes/defines what kind of data can be held in this storage element, whether it be numeric (for example, integer or decimal), text, binary, time, date, Boolean, and so forth. The data type defines the constraints placed on what the contents of a data element can be, determines the functions in which that data element can be used (not all of the data types are supported by all functions), and defines what the results of certain functions will be.
- **Fundamental (Simple) data type:** This has a simple structure consisting of a single value at one time, for example, a number, string of text, or a character.

- **Complex data type:** This has a structure made up of or relating to simple data types, for example, records, program objects such as Forms/Pages or Reports, BLOBs, DateFormulas, an external file, an indirect reference variable, and so on.
- **Data Element:** An instance of a data type which may be a Constant or a Variable.
- **Constant:** This is a data element explicitly specified in the code by value, not modifiable 'on the fly', and known in some circles as 'hard wired' data. All of the simple data types can be represented by constants. Examples are 'MAIN', 12.34, and '+01-312-444-5555'.
- **Variable:** This is a data element that can have a value assigned to it dynamically, as the program runs. Except for special cases, a variable will be of a single, unchanging, specific data type.

Fields

A field is the basic element of data definition in NAV – the "atom" in the structure of a system. The elemental definition of a field consists of its number, its description (name), its data type, and, of course, any parameters required for its particular data type. A field is also defined by its properties and the C/AL code contained in its triggers.

Field properties

The specific properties that can be defined for a field depend on the data type. There are a minimum set of universal properties. We will review those first. Then we will review the rest of the more frequently used properties, both those that are data dependent and those that are not. You can check out the remaining properties by using **C/Side Reference Guide Help** from within the Table Designer.

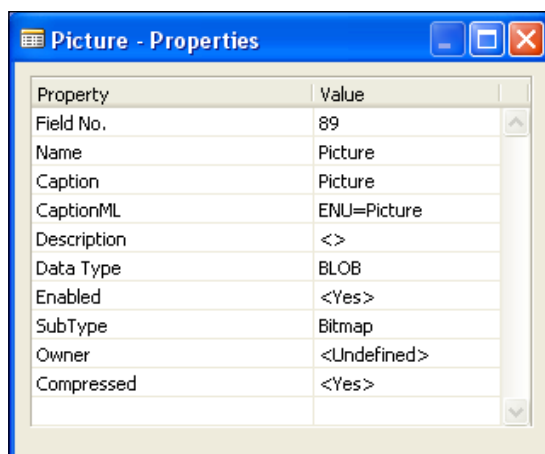
You can access the properties of a field while viewing the table in Design mode, highlighting the field, and then clicking on the **Properties** icon or pressing *Shift + F4*. All of the property screenshots in this section were obtained this way for fields within the standard **Customer** table. As we review various field properties, you will learn more if you follow along in your NAV system using the Object Designer. Explore different properties and the values they can have. Use the **Help** functions liberally to review the help for various properties.

The property value which is enclosed in < > (less than and greater than brackets) is the default value for that property. When you set a property to any other value, < and > should not be present unless they are supposed to be a part of the property value (for example, as part of a Text string value).

All of the fields, of any data type, have the following properties:

- **Field No.:** The identifier for the field within the containing table object
- **Name:** The label by which C/AL code references the field. A name can consist of up to 30 characters, including special characters. The name can be changed at any time and NAV will automatically ripple that change throughout the system. Changing names that are in use in C/AL code can cause problems with some functions such as web services and GETFILTERS where the reference is based on the field name rather than the field number. The name is used as the default caption when data from this table is displayed and no Caption has been defined.
- **Caption:** This contains the defined caption for the currently-selected language. It will always be one of the defined multi-language captions. The default language for a NAV installation is determined by the combination of a set of built-in rules and the languages available in the installation.
- **CaptionML:** This defines the MultiLanguage caption for the table.
- **Description:** This is an optional use property, for your internal documentation only.
- **Data Type:** This defines what type of data format applies to this field (for example, Integer, Date, Code, Text, Decimal, Option, Boolean, and so on)
- **Enabled:** This determines whether or not the field is activated for data handling. The property defaults to **<Yes>** and is rarely changed.

The following screenshot shows the BLOB properties for the **Picture** field in the **Customer** table:



This set of properties, for fields of the BLOB data type, is the simplest set of field properties. After the properties that are shared by all of the data types, appear the BLOB-specific properties – SubType, Owner, and Compressed.

- **SubType:** This defines the type of data stored in the BLOB and sets a filter in the import/export function for the field. The three sub-type choices are Bitmap (for bitmap graphics), Memo (for text data), and User-Defined (for anything else). User-Defined is the default value.
- **Owner:** The usage is not defined.
- **Compressed:** This defines whether the data stored in the BLOB is stored in a compressed format. This format is not supported by the C/SIDE database. There is no documentation on how or when compression is used.

The properties of Code and Text data type fields are quite similar to one another.

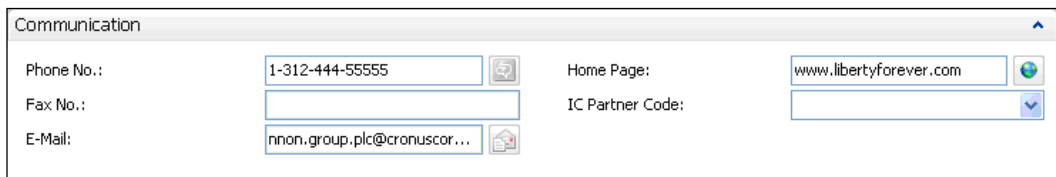
Property	Value
Field No.	1
Name	No.
Caption	No.
CaptionML	ENU=No.
Description	<>
Data Type	Code
Enabled	<Yes>
DataLength	20
InitValue	<Undefined>
FieldClass	<Normal>
AltSearchField	Search Name
AutoFormatType	<0>
AutoFormatExpr	<>
CaptionClass	<>
Editable	<Yes>
NotBlank	<No>
Numeric	<No>
CharAllowed	<Undefined>
DateFormula	<No>
ValuesAllowed	<>
SQL Data Type	<Undefined>
TableRelation	<Undefined>
ValidateTableRelation	<Yes>
TestTableRelation	<Yes>
ExtendedDatatype	<None>

Property	Value
Field No.	2
Name	Name
Caption	Name
CaptionML	ENU=Name
Description	<>
Data Type	Text
Enabled	<Yes>
DataLength	50
InitValue	<Undefined>
FieldClass	<Normal>
AltSearchField	<Undefined>
AutoFormatType	<0>
AutoFormatExpr	<>
CaptionClass	<>
Editable	<Yes>
NotBlank	<No>
Numeric	<No>
CharAllowed	<Undefined>
DateFormula	<No>
Title	<No>
ValuesAllowed	<>
TableRelation	<Undefined>
ValidateTableRelation	<Yes>
TestTableRelation	<Yes>
ExtendedDatatype	<None>

The following are some common properties between the two as shown in the preceding screenshot:

- **DataLength:** This specifies how many characters long the data field is.
- **InitValue:** This is the value that the system will supply as a default when the system initializes the field.
- **AltSearchField:** This allows the definition of a single alternative field in the same table to be searched for a match if no other match is found on a lookup of this data item. For example, you might want to allow customers to be looked up either by their Customer No. or by their Phone No. In that case, in the **No.** field's properties, you would supply the Phone No. field name in the **AltSearchField** field. Then, when a user searches in the **No.** field, NAV will first look for a match in the **No.** field and if it is not found there, it will search the **Phone No.** field for a match. The use of this property can save you a lot of coding, but make sure both the original and alternate search fields have high placement in a key so the lookup will be speedy (optimum placement in a key is as the first element). In the preceding image, the **No.** field **AltSearchField** property value is **Search Name**. **AltSearchField** does not appear to work in the Role Tailored Client. It can be simulated by use of Field Groups, which we will discuss later in this chapter and subsequently.
- **Caption Class:** This defaults to empty. According to the documentation (see *C/Side NAV 2009 Developer Help*), this property controls the contents of the caption associated with a field. This is used, for example, to allow user definition of fields like the Dimensions to have captions such as Job and Department. This can also be used to assist in the translation of captions for a multi-language user interface and to support the RTC version of Matrix page column headings.
- **Editable:** This is set to **No** when you don't want to allow a field to be edited, such as if it is a computed or assigned value field that the user should not change.
- **NotBlank, Numeric, CharAllowed, DateFormula, and ValuesAllowed:** All these allow placing you to place constraints on the specific data that can be entered into this field.
- **SQL Data Type:** This applies to **Code** fields only. **SQL Data Type** allows defining what data will be allowed in this particular Code field and how it will be sorted and displayed. Options are *Varchar*, *Integer*, *Variant*, and *BigInteger*. *Varchar* causes all of the data to be treated as text. *Integer* (and presumably *BigInteger*) allows only numeric data to be entered. *Variant* allows the full set of Code permitted data and causes numeric data to sort after alphanumeric data. In general, once set, this property should not be changed.

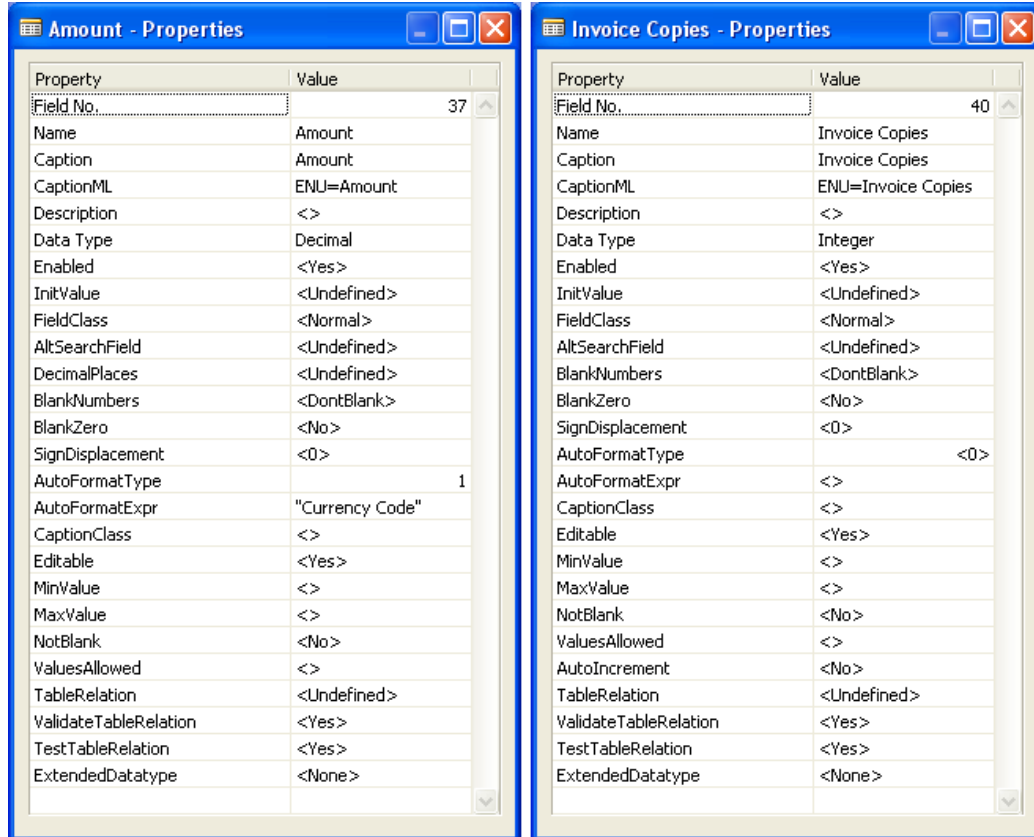
- **TableRelation:** This property is used to specify that the data field relates to data in a particular table. The related table field must be in a Primary Key. The relationship can be conditional and/or filtered. The relationship can be used for validation, lookups, and data-change propagation.
- **ValidateTableRelation:** If **TableRelation** is specified, set this to **Yes** in order to validate the relation when data is entered or changed.
- **TestTableRelation:** An infrequently used property which only controls whether or not the relationship should be tested during a database integrity test.
- **ExtendedDataType:** This property allows the optional designation of an extended data type which automatically receives special formatting and validation, commonly as an email address, a URL, or a phone number. An action icon is also added next to the field, as shown in the following image where there are three fields with **ExtendedDataType** defined.



The screenshot shows a window titled "Communication". It contains five input fields arranged in two columns. The left column has "Phone No.:" with the value "1-312-444-5555", "Fax No.:" which is empty, and "E-Mail:" with the value "nnnon.group.plc@cronuscor...". The right column has "Home Page:" with the value "www.libertyforever.com" and "IC Partner Code:" which is empty. Each field has a small icon to its right: a document with a magnifying glass for phone/fax, a globe for the home page, and a document with a magnifying glass for the email.

Let us take a look at the properties of two more data types, **Decimal** and **Integer**. You should explore them in detail on your own as well. Specific properties related to the basic numeric content of these data types are as follows and are shown in the following screenshot:

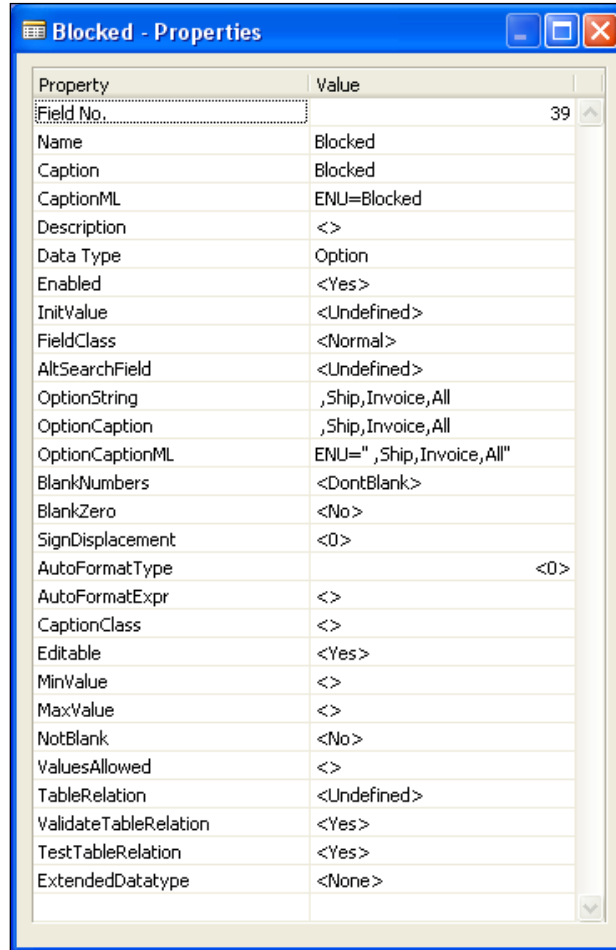
- **DecimalPlaces:** This sets the number of decimal places in a Decimal data item.
- **BlankNumbers, BlankZero, and SignDisplacement:** All these can be used to influence the formatting and display of the data in the field.
- **MinValue and MaxValue:** These can constrain the range of data values allowed. The range available depends on the field data type.
- **AutoIncrement:** This allows setting up of one field in a table to automatically increment for each record entered. When used, which is not often, it is almost always to support the automatic updating of a field used as the last field in a Primary Key, which enables the creation of a unique key. Use of this feature does not ensure a contiguous number sequence. Under some circumstances, use of this feature can lead to table locking conflicts. The automatic functionality should not be overridden in code.



The properties for an **Option** data type, shown in the following screenshot, are like those of the other numeric data types. This is logical since an Option is stored as an integer. They also include data type-specific properties.

- **OptionString:** This spells out the text interpretations for the stored integer values contained in Option data type fields.

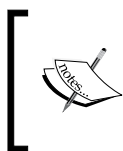
- **OptionCaption** and **OptionCaptionML**: These serve the same captioning and multi-language purposes as other caption properties.



Field numbering

The number of each field within its parent table object is the unique identifier that NAV uses internally to identify that field. You can easily change a field number when you are initially defining a table layout. But after you have a number of other objects (for example, pages, reports, or codeunits) referencing the fields in a table, it becomes challenging, and often almost impossible, to change the numbers of one or more fields. Therefore, you should plan ahead to minimize any need to re-number fields.

You should also be cautioned that, although you can physically delete a field and then re-use its field number for a different purpose, doing so is likely to cause you much grief.



You cannot re-number or delete a field that has data present in the database. You cannot reduce the defined size of a field to less than the largest size of data already present in that field. These limits are the compiler working to protect you and your database.

You should be careful about the field numbers assigned if you are going to use the Classic Client. In that client, the numeric sequence of fields within the table controls the sequence of the field names when they are displayed in field selection lists to users. These field selection lists are presented when a user constructs a data Filter, chooses the Page or Report Request Page field selection option, or views a table in a default display. If the fields in a table are not in a relatively logical sequence, or fields with a similar purpose are not grouped, the system will be harder to understand and therefore, harder to use. This is not true in the Role Tailored Client, where the equivalent displays are in alphabetical sequence based on the field names.

Unfortunately, the above criticism could be made about the field sequence structure of some of the standard system's principle master tables (for example, Customer, Vendor, and Item). This has happened over a decade of changes and functional expansion. During that period of system design change, the original field numbers have largely remained unchanged in support of backward compatibility. At the same time, new related fields have been added in less than ideal related field number sequences. The result is a list of fields presented to the users in a sequence that follows very few logical rules.

For the new fields that you add to tables, which are a part of the standard NAV product, the new field numbers must be in the 50,000 to 99,999 number range, unless you have been explicitly licensed for another number range. Field numbers for fields in new tables that you create may be anything from 1 to 999,999,999 (without the commas).

When fields appear in several related tables (for example, journal and ledger), the same field number should be assigned to each of the tables. Not only is this consistent approach easier for user reference and maintenance, but it also supports the `TRANSFERFIELDS` function, which allows copying data from one record instance to another record instance, even when associated with a different table. The record to record mapping for the copy is based on the field numbers; hence, the importance of using the same number for the same field in different tables.



The key to avoiding either re-numbering fields or presenting field lists in a hard to use sequence is to plan ahead and number the fields properly from the beginning. One aid is to leave frequent gaps in field number sequences within a table, to allow the easy insertion of new fields numbered adjacent to related, previously existent fields.

Changing the data type of a field

What if we wish to change the data type of a field? For example, perhaps we had originally designed the Postal Zone field as an Integer to only handle five digit US zip codes, which are numeric. Then later we decide to generalize and allow postal codes for all countries. In that case, we must change our data field from integer to code, which allows all of the numerals and upper case letters.

How do we solve the data definition—data content inconsistency caused by the change? We have a couple of choices. The first option, which could work in our ICAN database because we have very little data and it's just test data, is simply to delete the existing data, proceed with our change, and then restore the data through keyboard entry. In a few instances, for example, Code to Text, the data types are compatible and the change is trivial. But when dealing with a non-compatible data type change and a significant volume of production data (more typical), you must take a more conservative approach. In this case, more conservative means more work.

Let us look at the steps required for a common example of changing the data type because of a design change. We will assume that the field **70 Post Code** was defined as data type **Integer** and we need to change it to data type **Code**, Length **20**. The steps are as follows:

1. Make sure there is a good, restorable backup of the data to be changed.
2. Create a new, temporary field **90000** named **Temp Post Code**, data type **Code**, and Length **20**. Any allowable field number and unique name would work.
3. Copy the data from the original field **110 Post Code** into the new temporary field **90000**, deleting the data from field **110** as you go, using a Processing Only report object created just for this purpose.
4. Redefine field **110** to new data type.
5. Copy the data from the temporary field **90000** back into the redefined field **110**, deleting the data from field **90000**, using a second Processing Only report object created just for this purpose.
6. Delete the temporary field **90000**.

If we had to renumber the fields, we would have to essentially do the same thing as just described, for each field. Whenever you attempt a change and see a message like the following, you will have to utilize the procedure just described:

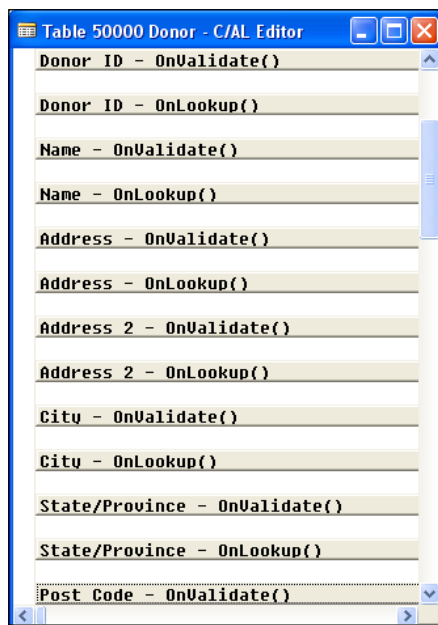


It's a lot of work to make a minor change in a table, especially if this is the result of inadequate design planning. Hopefully, this convinces you of the importance of carefully considering how you define fields and field numbers at the beginning.

By the way, this process of moving data in and out of temporary fields is exactly the process that Upgrade Data Conversions go through to change the field structure of a table in the database to support added capabilities of the new version. A logically similar process is used to split or delete tables during an Upgrade Data Conversion.

Field triggers

To see what field triggers are, let us look at our **Table 50000 Donor**. Open the table in the **Design** mode, highlight the **Donor ID** field, and press **F9**. The window shown in the following screenshot will appear:



Each of the field has two triggers – the `OnValidate()` trigger and the `OnLookup()` trigger – which function as follows:

- `OnValidate()`: The C/AL code in this trigger is executed whenever an entry is made by the user. It can also be executed under program control through use of the `VALIDATE` function (which we will be discussing later).
- `OnLookup()`: The C/AL code in this trigger is executed in place of the system's default Lookup behavior, even if the C/AL code is only a comment. Lookup behavior can be triggered by pressing *F6* (in Classic Client), or *F4* or *Ctrl+U* (in RTC) or by clicking on the lookup arrow in a field as shown in the following RTC screenshot:



If the field's **TableRelation** property refers to a table, then the default behavior is to display a drop-down list to allow selection of an entry to be stored in this field. The list will be based on the Field Groups defined for the table. You may choose to override that behavior by coding different behavior for a special case.



Be careful with the `OnLookup()` trigger. Any entry whatsoever in the body of an `OnLookup()` trigger will eliminate the default behavior. This is true even if the entry is only a comment and there is no executable code present. A comment line could make an intended default lookup fail. Conversely, you could include a comment line to make sure that no `OnLookup` occurs, if that is your goal.

Data structure examples

Some of the good examples of tables in the standard product to review for particular features are:

- Table 18 – Customer, for a variety of data types and Field Classes. This table contains some fairly complex examples of C/AL code in the table Triggers. A wide variety of field property variations can be seen in this table as well.
- Table 50 – Accounting Period, has a couple of very simple examples of Field `OnValidate` trigger C/AL code. For slightly more complex examples, take a look at Table 167 – Job. For much more complex examples, you can look at almost all of the primary master tables such as Customer, Vendor, Item, and so on.

You can find all the tables at **Tools | Object Designer**, by clicking on **Tables**.

Variable naming

Variables in NAV can either be global (with a scope across the breadth of an object) or local (with a scope only within a single function). Variable names should be unique within the sphere of their scope. There must not be any duplication between global and local names. Even though the same local name can be used in more than one function within the same object, doing so may confuse the compiler and will almost certainly confuse the developer that follows you. Therefore, you should make your working variable names unique within the object. In addition, performance can be impacted by declaring complex data types (such as records) locally as this will introduce overhead on each call.

Uniqueness includes not duplicating reserved words or system variables. Refer to the **C/AL Reserved Words** list for a reasonably complete list. A good guideline is to avoid using, as a variable name, any word that appears as an UPPER CASE word in either the C/SIDE Help or any of the published NAV technical documentation. For example, you shouldn't use the word **Page** as a variable name.

Variable names in NAV are not case sensitive. There is a 30-character length limit on variable names. Variable names can contain all ASCII characters except for control characters (ASCII values 0 to 31 and 255) and the asterisk (*, ASCII value 42), as well as some Unicode characters used in languages other than English. Characters outside the standard ASCII set may display differently on different systems.

Note that the compiler won't tell you an asterisk cannot be used in a variable name. It is also a good idea to avoid using the question mark (?, ASCII value 63). This is because both the asterisk and the question mark can be used as **wildcards** in many expressions, especially filtering. We'll discuss more about this later in the book.

Unless the variable name is enclosed in double quotes, the first character of a variable name must be a letter A to Z (upper or lower case) or an underscore (_ ASCII value 95). Alphabets other than the 26-character English alphabet may interpret the ASCII values to characters other than A to Z, and may include additional characters beyond 26. The variable name can be followed by any combination of the legal characters. If you use any characters other than the alphabet, numerals, and underscore, you must surround your variable name with double quotes each time you use it in C/AL code (for example, "Cust List", which contains an embedded space, or "No.", which contains a period). While the Developer Help doesn't tell you that you can't use a double quote character within a variable name, common sense and the compiler tell you not to do so.

Data types

We are going to segregate the data types into relatively obvious groupings. Overall, we will first look at Fundamental (that is, simple) data types, then Complex data types. Within fundamental data types, we will consider Numeric, String, and Time data types, while in complex data types, we will look at data items, data structures, objects, automation, input/output, references, and others.

Fundamental data types

Fundamental data types are the basics from which the complex data types are formed. They are grouped into Numeric, String, and Date/Time data types.

Numeric data

Just like other systems, NAV allows several numeric data types. The specifications for each of the data types are somewhat dependent on the underlying database in use. Our target database is SQL Server, the only database compatible with the Role Tailored Client environment. For more details on the SQL Server-specific representations of various data elements, you should refer to the C/SIDE Reference Guide online help. The various numeric data types are as follows:

- **Integer:** An integer number ranging from -2,147,483,648 to +2,147,483,647
- **Decimal:** A decimal number in the range of +/- 999,999,999,999.99. Although it is possible to construct larger numbers, errors such as overflow, truncation, or loss of precision might occur. In addition, there is no facility to display or edit larger numbers.
- **Option:** A special instance of an integer, stored as an integer number ranging from 0 to +2,147,483,547. An option is normally represented in the body of your C/AL code as an option string. You can compare an option to an integer in C/AL rather than using the option string, but that is not a good practice because it eliminates the self-documenting aspect of an option field.

An option string is a set of choices listed in a comma-separated string, one of which is chosen and stored as the current option. The currently selected choice within the set of options is stored as the ordinal position of that option within the set. For example, selection of an entry from the option string of red, yellow, blue would result in the storing of 0 (red), 1 (yellow), and 2 (blue). If red were selected, 0 would be stored in the variable; and if blue were selected, 2 would be stored. Quite often, an option string starts with a blank to allow an effective choice of "none chosen".

- **Boolean:** These are stored as 1 or 0, programmatically referred to as True or False, but sometimes referred to in properties as Yes or No. Boolean variables may be displayed as Yes or No, ☒ or blank, and sometimes as True or False.
- **Binary:** This is just what its name indicates, binary data. Binary data was supported in previous versions but not in the NAV 2009 Role Tailored Client.
- **BigInteger:** 8-byte Integer as opposed to the 4 bytes of Integer. BigIntegers are for very big numbers (from -263 to 263-1).
- **Char:** A numeric code between 0 and 256 representing an ASCII character. To some extent, Char variables can operate either as text or as numbers. Numeric operations can be done on Char variables. Char variables can also be defined with character values. Char variables cannot be defined as permanent variables in a table, but only as working storage variables within C/AL objects.

String data

The following are the data types included in String data:

- **Text:** This contains any string of alphanumeric characters. In a table, a Text variable can be from 1 to 250 characters long; in working storage within an object a Text variable can be from 1 to 1024 characters long (storage in the database may differ). But when calculating the 'length' of a record for design purposes (relative to the maximum record length of 4096 characters), the full defined field length should be counted.
- **Code:** This contains any string of alphanumeric characters ranging from 1 to 250. All of the letters are automatically converted to uppercase when entered.

Date/Time data

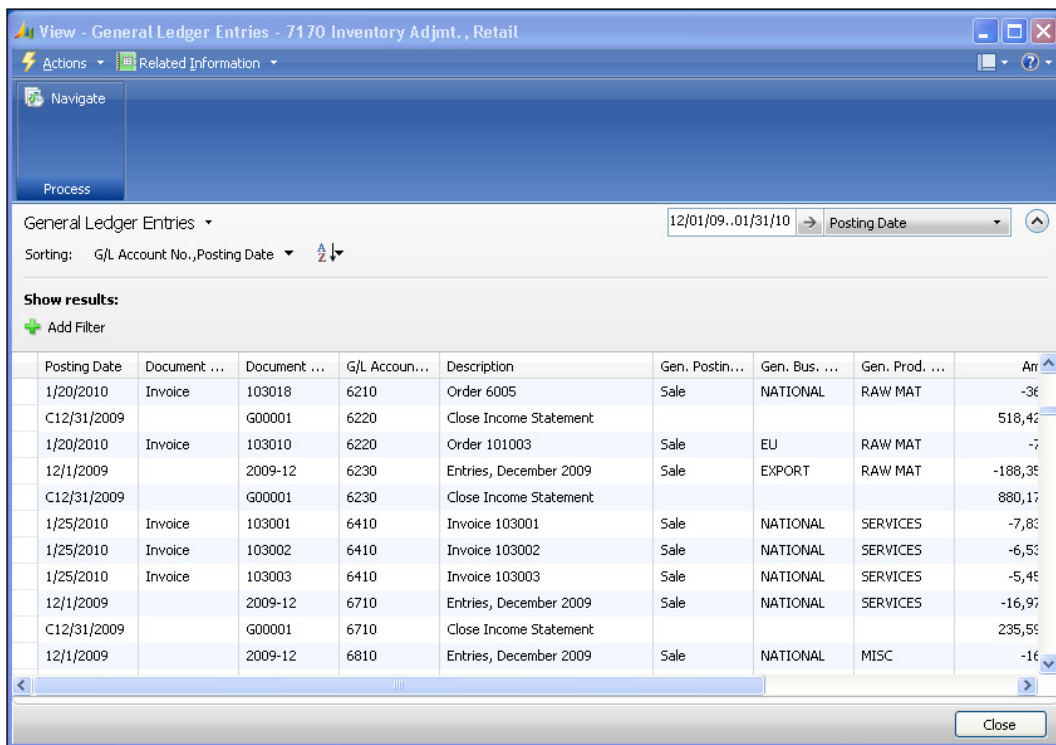
The following are the data types included in Date/Time data:

- **Date:** This contains an integer number, which is interpreted as a date ranging from January 1, 1754 (in SQL Server) to December 31, 9999. A 0D (numeral zero, letter D) represents an undefined date which is interpreted as January 1, 1753 in SQL Server.

A **date constant** can be written as a letter **D** preceded by either six digits in the format MMDDYY or eight digits as MMDDYYYY (where M = month, D = day and Y = year). For example, 011911D or 01192011D both represent January 19, 2011. Later, in **DateFormula**, we will find D interpreted as day, but here the trailing D is interpreted as the date (data type) constant. When the year is expressed as YY rather than YYYY, the century portion (in this case, 20) is supplied based on the century portion of the system date.

NAV also defines a special date called a "Closing" date, which represents the point in time between one day and the next. The purpose of a closing date is to provide a point at the end of a day, after all of the real date- and time-sensitive activity is recorded – the point when accounting "closing" entries can be recorded.

Closing entries are recorded, in effect, at the stroke of midnight between two dates – this is the date of closing of accounting books, designed so that one can include or not include, at the user's option, closing entries in various reports. When sorted by date, the closing date entries will get sorted after all normal entries for a day. For example, the normal date entry for December 31, 2009 would display as **12/31/09** (depending on your date format masking), and the closing date entry would display as **C12/31/09**. All of the C12/31/09 ledger entries would appear after all normal 12/31/09 ledger entries. The following screenshot shows some 2009 closing date entries mixed with normal entries from December 2009 and January 2010:



Posting Date	Document ...	Document ...	G/L Account...	Description	Gen. Postin...	Gen. Bus. ...	Gen. Prod. ...	Am
1/20/2010	Invoice	103018	6210	Order 6005	Sale	NATIONAL	RAW MAT	-36
C12/31/2009		G00001	6220	Close Income Statement				518,4
1/20/2010	Invoice	103010	6220	Order 101003	Sale	EU	RAW MAT	-7
12/1/2009		2009-12	6230	Entries, December 2009	Sale	EXPORT	RAW MAT	-188,3
C12/31/2009		G00001	6230	Close Income Statement				880,1
1/25/2010	Invoice	103001	6410	Invoice 103001	Sale	NATIONAL	SERVICES	-7,8
1/25/2010	Invoice	103002	6410	Invoice 103002	Sale	NATIONAL	SERVICES	-6,5
1/25/2010	Invoice	103003	6410	Invoice 103003	Sale	NATIONAL	SERVICES	-5,4
12/1/2009		2009-12	6710	Entries, December 2009	Sale	NATIONAL	SERVICES	-16,9
C12/31/2009		G00001	6710	Close Income Statement				235,5
12/1/2009		2009-12	6810	Entries, December 2009	Sale	NATIONAL	MISC	-16

- **Time:** This contains an integer number, which is interpreted on a 24-hour clock, in milliseconds, from 00:00:00 to 23:59:59.999. A 0T (numeral zero, letter T) represents an undefined time.
- **DateTime:** This represents a combined Date and Time, stored in **Coordinated Universal Time (UTC)** and always displays local time (that is, the local time on your system). DateTime fields do not support NAV "Closing Date". DateTime is helpful for an application that must support multiple time zones simultaneously. DateTime values can range from January 1, 1754 00:00:00.000 to December 31, 9999 23:59:59.999 (don't test with dates late in 9999 as an intended advance to the year 10000 won't work). An undefined or blank DateTime is stored as January 1, 1753, 00:00:00.000.
- **Duration:** This represents the positive or negative difference between two DateTime values, in milliseconds, stored as a BigInteger.

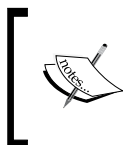
Complex data types

There are a variety of complex data types. Each consists of multiple data elements. For ease of reference, we will categorize them into several groups of similar types.

Data structure

The following data types are in the data structure group:

- **File:** This refers to any standard Windows file outside the NAV database. There is a reasonably complete set of functions to allow creating, deleting, opening, closing, reading, writing, and copying (among other things) data files. For example, you could create your own NAV routines in C/AL to import or export data from a file that had been created by some other application.



When interfacing with external files in the three-tier system, you must carefully plan where the external files are placed for access, because the file read/write processing will occur on the middle tier server system, not on the user's workstation.

- **Record:** This refers to a single line of data within a NAV table. Quite often, multiple instances of a Record variable (that is, table) are defined to support a validation process, allowing access to different records (that is, rows) within the table, within the same function. The working storage variable for a table will be data type Record.

Objects

Page, Form, Report, Dataport, Codeunit, and XMLPort each represent an object of the type Page, Form, Report, Dataport, Codeunit, and XMLPort, respectively. Object data types are used when there is a need for reference to an object or some portion of an object from within another object. Examples are the cases where one object invokes another (for example, calling a Report object from a Form or Page object or from another Report object) or where one object is taking advantage of data validation logic that is coded as a function in a Table object or a Codeunit object.

Automation

The following are Automation data types:

- **OCX:** This allows the definition of a variable that represents and allows access to an ActiveX or OCX custom control. Such a control is typically another external application object, small or large, which you can then invoke from your NAV object.
- **Automation:** This allows the definition of a variable that you may access similarly to an OCX, but is more likely to be a completely independent application. The application must act as an Automation Server and must be registered with the NAV client or server calling it. For example, you can interface from NAV into the various Microsoft Office products (Word, Excel, and so on) by defining them in Automation variables.

Input/Output

The following are the data types in Input/Output:

- **Dialog:** This allows the definition of a simple user interface window without the use of a Page object. Typically, dialog windows are used to communicate processing progress or to allow a brief user response to a go/no-go question, though this latter use could result in bad performance due to locking. There are other user communication tools as well, but they do not use a dialog data item.
- **InStream** and **Outstream:** These are variables that allow reading from and writing to external files, BLOBS, and objects of the Automation and OCX data types.

DateFormula

DateFormula provides the storage of a simple, but clever, set of constructs to support the calculation of runtime-sensitive dates. A DateFormula is a combination of:

- Numeric multipliers (for example, 1, 2, 3, 4, and so on)
- Alpha time units (all must be upper case)
 - D for a day
 - W for a week
 - WD for day of the week, that is day 1 through day 7 (either in the future or in the past, not today), Monday is day 1, Sunday is day 7
 - M for calendar month
 - P for accounting period
 - Y for year
 - CM for current month, CY for current year, CP for current period, CW for current week
- Math symbols
 - + (plus) as in CM + 10D means the Current Month end plus 10 Days or the 10th of next month
 - - (minus) as in -WD3 means the date of the previous Wednesday
- Positional notation (D15 means the 15th of the month and 15D means 15 days)

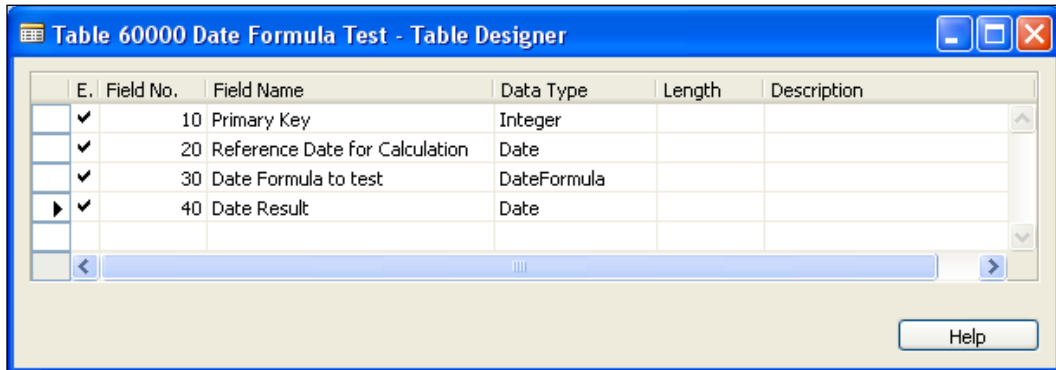
Payment Terms for Invoices make very productive use of DateFormula. All DateFormula results are expressed as a date based on a reference date. The default reference date is the system date, not the Work Date.

Here are some sample DateFormulas and their interpretations (displayed dates are based on the US calendar) with a reference date of July 9, 2010, a Friday:

- CM = the last day of Current Month, 07/31/10
- CM + 10D = the 10th of next month, 08/10/10
- WD6 = the next sixth day of week, 07/10/10
- WD5 = the next fifth day of week, 07/16/10
- CM - M + D = the end of the current month minus one month plus one day, 07/01/10
- CM - 5M = the end of the current month minus five months, 02/28/10

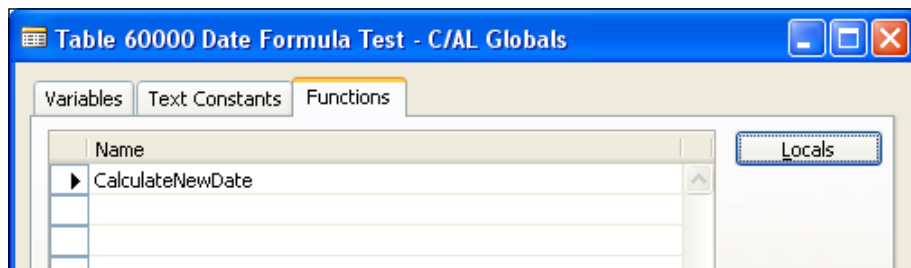
Let us take the opportunity to use the DateFormula data type to learn a few NAV development basics. We will do so by experimenting with some hands-on evaluations of several DateFormula values. We will create a table to calculate dates using DateFormula and Reference Dates.

First, create a table using the **Table Designer** as you did in earlier instances. Go to **Tools | Object Designer | Tables**. Click on the **New** button and define the fields as in the following screenshot. Save it as **Table 60000**, named **Date Formula Test**. After you are done with this test, we will save this table for some later testing.



Now we will add some simple C/AL code to our table so that when we enter or change either the Reference Date or the DateFormula data, we can calculate a new result date.

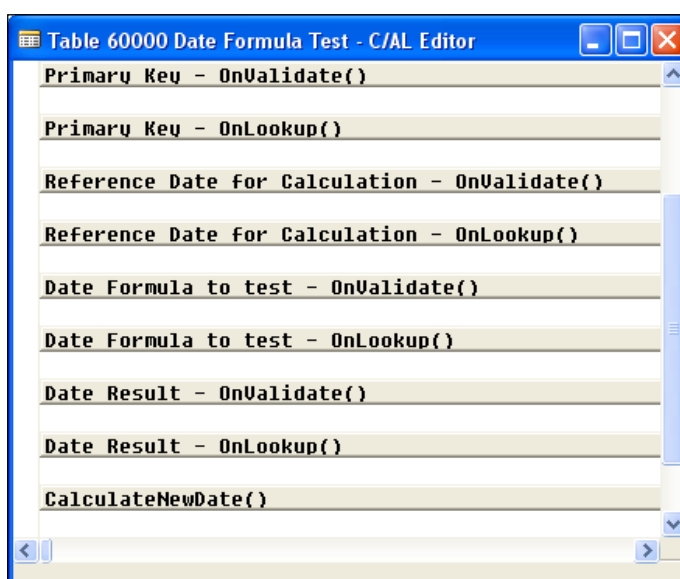
First, access the new table via the **Design** button, then go to the global variables definition form through the **View** menu option, the sub-option **Globals**, and finally choose the **Functions** tab. Type in our new function name as **CalculateNewDate** on the first blank line as shown in the following screenshot and then exit (by means of **Esc** key) from this form back to the list of data fields:



From the **Table Designer** form displaying the list of data fields, either press *F9* or click on the **C/AL Code** icon:



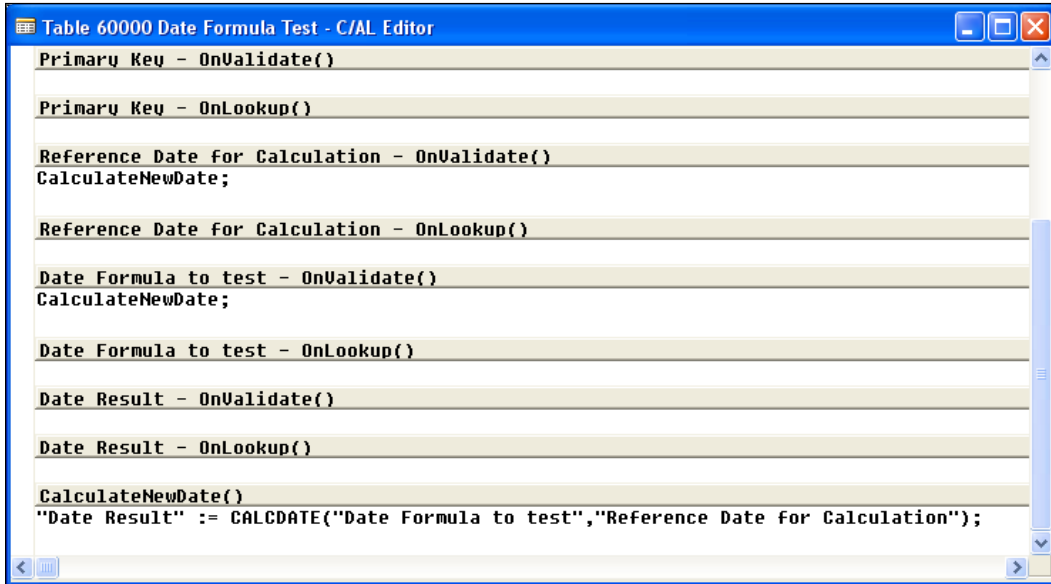
This will take you to the following screen, where you will see all of the field triggers, plus the trigger for the new function that you just defined. The table triggers are not visible, unless we scroll up to show them.



As our goal this time is to focus on experimenting with the `DateFormula`, we will not go much into detail about the logic we are creating. Hopefully, your past experience will allow you to understand the essence of the code.

We are going to create the logic within our new function, `CalculateNewDate()`, to evaluate and store a **Date Result** based on the `DateFormula` and `Reference Date` that we enter into the table.

Just copy the C/AL code exactly as shown in the following screenshot, exit, compile, and save your table.



When you close and save the table, if you get an error message of any type, you probably have not copied the C/AL code exactly as it is shown in the screenshot.

This code will cause the function `CalculateNewDate()` to be called any time an entry is made in either the **Reference Date for Calculation** or the **Date Formula to test** fields. The result will be placed in the **Date Result** field. The use of an integer value in the redundantly named **Primary Key** field allows you to enter several records into the table (by numbering them 1, 2, 3, and so forth) and also allows you to compare the results of date calculations by using several different formulae.

Let us try a few examples. We will access the table via the **Run** button. Enter a **Primary Key** value of 1 (that is, one).

In **Reference Date for Calculation**, enter the letter **T** for today (either uppercase or lowercase), the system date. The same date will appear in the **Date Result** field, because at this point there is no Date Formula entered. Now enter **1D** (numeral 1 followed by the letter **D**, upper case or lower case, C/SIDE will take care of making it upper case) in the **Date Formula to test** field. You will see the **Date Result** field contents are changed to be one day beyond the date in the **Reference Date for Calculation** field.

Let us enter another line. Start with a numeral **2** in the **PrimaryKey** field. Again, enter the letter **T** to insert the system date (that is, Today) in the **Reference Date for Calculation** field and just enter the letter **W** in the **Date Formula to test** field. You will get an error message telling you that your formulas should include a number. Make the system happy and enter **1W**. You will see a date in the **Date Result** field one week beyond your system date.

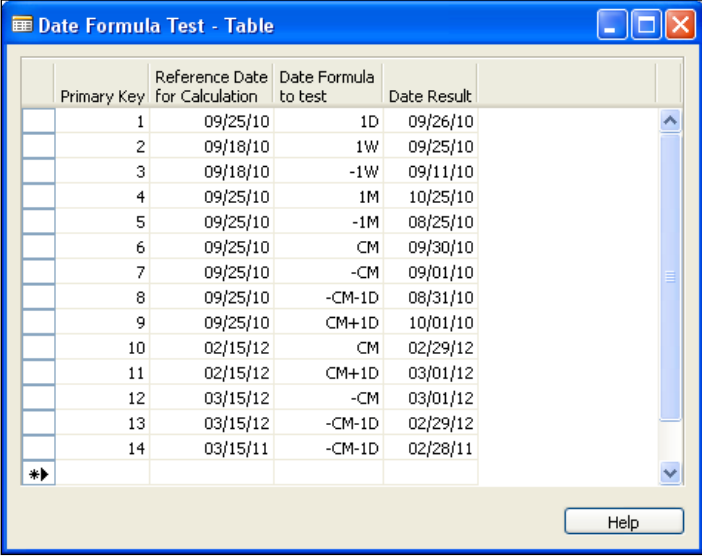
Set the system's Work Date to a date about in the middle of a month. Start another line with the number **3** as the **Primary Key**, followed by a **W** (for Work Date) in the **Reference Date for Calculation** field. Enter **cm** (or CM or cM or Cm, it doesn't matter) in the **Date Formula to test** field. Your result date will be the last day of your Work Date month. Now enter another line using the Work Date, but enter a formula of **-cm** (the same as before, but with a minus sign). This time your result date will be the first day of your Work Date month.

Primary Key	Reference Date for Calculation	Date Formula to test	Date Result	
1	09/25/10	1D	09/26/10	
2	09/25/10	1W	10/02/10	
3	09/18/10	CM	09/30/10	
4	09/18/10	-CM	09/01/10	
*▶				

Help

Enter another line with a new Primary Key. Skip over the **Reference Date for Calculation** field and just enter **1D** in the **Date Formula to test** field. What happens? You get an error message. NAV cannot deal with making a calculation without a Reference Date. If we put this function into production, we might enhance our code to check for a Reference Date before calculating. We could default an empty date to the System Date or the Work Date and avoid this particular error.

The following screenshot shows different sample calculations. Build on these and experiment on your own. You can create a variety of different algebraic formulae and get some very interesting results. One NAV user has due dates on Invoices on 10th of the next month. The Invoices are dated at various times during the month than they are actually printed. But by using the DateFormula of **CM + 10D**, the due date is always the 10th of the next month.



Primary Key	Reference Date For Calculation	Date Formula to test	Date Result
1	09/25/10	1D	09/26/10
2	09/18/10	1W	09/25/10
3	09/18/10	-1W	09/11/10
4	09/25/10	1M	10/25/10
5	09/25/10	-1M	08/25/10
6	09/25/10	CM	09/30/10
7	09/25/10	-CM	09/01/10
8	09/25/10	-CM-1D	08/31/10
9	09/25/10	CM+1D	10/01/10
10	02/15/12	CM	02/29/12
11	02/15/12	CM+1D	03/01/12
12	03/15/12	-CM	03/01/12
13	03/15/12	-CM-1D	02/29/12
14	03/15/11	-CM-1D	02/28/11

Don't forget to test with WD (weekday), P (period), Q (quarter), and Y (year).

While we used DateFormula as the basis for the work we just completed, you've accomplished a lot more than simply learning about that one data type.

- You created a new table just for the purpose of experimenting with a C/AL feature that you might use. This is a technique that comes in handy when you are learning a new feature, trying to decide how it works or how you might use it.
- We put some critical logic in the table. When data is entered in one area, the entry is validated and, if valid, the defined processing is done instantly.
- We created a common routine as a new function. That function is then called from multiple places to which it applies.

- We did our entire test with a table object and a default tabular form that is automatically generated when you **Run** a table. We didn't have to create much of a supporting structure to do our testing. Of course, when you are designing a change to a complicated existing structure, it is likely that you will have a more complicated testing scenario. And when you are developing for the RoleTailored Client, you will do your testing in that environment. Even so, you may start your testing within the Classic Client development environment. One of your goals will always be to simplify your testing scenarios, both to minimize the setup effort and to keep your test narrowly focused on the specific issue.
- We saw how NAV tools make a variety of relative date calculations easy. These are very useful in business applications, many aspects of which are date centered.

References and other

The following data types are used for advanced functionality in NAV, typically supporting some type of interface with an external object.

- **RecordID:** This contains the object number and Primary Key of a table.
- **RecordRef:** This identifies a row in a table, that is, a record. It can be used to obtain information about the table, the record, the filters, and the fields in the record.
- **FieldRef:** This identifies a field in a table and, thereby, allows access to the contents of that field.
- **KeyRef:** This identifies a key in a table and the fields it contains.



RecordRef, FieldRef, and KeyRef are used to support logic which can run on tables that are unknown at design time.

- **Variant:** This defines variables typically used for interfacing with Automation and OCX objects. Variant variables can contain data of a number of other data types.
- **TableFilter:** This defines variables used only by the permissions table related to security functions.
- **Action:** This identifies the return Action from a `Form.RUNMODAL` or `Page.RUNMODAL` function. It has optional values of OK, Cancel, LookupOK, LookupCancel, Yes, No, Close, FormHelp, RunObject, and RunSystem.

- **Transaction Type:** This has optional values of **UpdateNoLocks**, **Update**, **Snapshot**, **Browse**, and **Report** which describe SQL Server basic transaction type options. This relates to the Report property of the same name. The options are designed to allow manual optimization of SQL Server performance for a particular operation design.
- **BLOB:** This can contain either a graphic in the form of a bitmap, specially formatted text, or other developer-defined binary data up to 2 GB in size. The term BLOB stands for **Binary Large Object**. BLOBs can only be included in tables, not used to define working storage Variable.
- **BigText:** This can contain large chunks of text up to 2 GB in size. BigText variables can only be defined in the working storage within an object, but not included in tables. BigText variables cannot be directly displayed or seen in the debugger. There is a group of functions that can be used to handle BigText data (for example, to move it to or from a BLOB, to read or write BigText data, to find a substring, to move data back and forth between BigText and normal Text variables, and so on).



If you wish to handle text strings in a single data element greater than 250 characters in length, you can use a combination of BLOB and BigText variables.

- **GUID:** This is used to assign a unique identifying number to any database object. GUID stands for **Globally Unique Identifier**, a 16-byte binary data type that is used for the unique global identification of records, objects, and so on. The GUID is generated by an algorithm developed by Microsoft.

Data type usage

Some data types can be used to define the data stored in tables or in working storage data definitions (that is, within a Global or Local data definition within an object).

A couple of data types can only be used to define table stored data, but not working storage data. A much larger set of data types can only be used for working storage data definitions.

The list in the following screenshot shows which data types can be used for table (persisted) data and which ones for working storage (variable) data:

Table Data Types	Working Storage Data Types
	Action
	Automation
BigInteger	BigInteger
	BigText
Binary	Binary
BLOB	
Boolean	Boolean
	Char
Code	Code
	Codeunit
	Dataport
Date	Date
DateFormula	DateFormula
DateTime	DateTime
Decimal	Decimal
	Dialog
Duration	Duration
	FieldRef
	File
	Form
GUID	GUID
	Instream
Integer	Integer
	KeyRef
	OCX
Option	Option
	Outstream
	Record
	Page
RecordID	RecordID
	RecordRef
	Report
TableFilter	
Text	Text
Time	Time
	TransactionType
	Variant
	XMLport

FieldClass property options

Each data field has a FieldClass property. We will cover most of the field properties in the next chapter, but FieldClass has as much affect on the content and usage of a data field as does the data type, in some instances maybe even more. For that reason, we will discuss FieldClass options now, as a follow-on to our discussion on data types.

- **Normal:** The FieldClass containing all of the 'normal' data — data that will be typically be stored in the parent table. If the FieldClass is normal, then the field contains just what you would expect, based on the data type and all the descriptions.
- **FlowField:** The FlowField is an important and controlling property of a field. FlowFields do not contain data in the conventional sense. They are virtual field. A FlowField contains the definition of how to calculate the data that it represents at run time. Depending on the CalcFormula Method, this could be a value, a reference lookup, or a Boolean. When the CalcFormula Method is `Sum`, this FieldClass connects a data field to a previously defined `SumIndexField` in a table.

A FlowField value is always 0, unless something happens to cause it to be calculated. If the FlowField is displayed directly on a form/page, then it is calculated automatically on initial display. FlowFields are also automatically calculated when they are the subject of predefined filters as part of the properties of a data item in an object (this will be explained in more detail in the chapters covering Reports and Dataports/XMLports). In all other cases, a FlowField must be forced to calculate using the `C/AL <Record>.CALCFIELDS` function. This is also true if the underlying data is changed after the initial display of a form/page (that is, the FlowField must be recalculated to take the change into account).

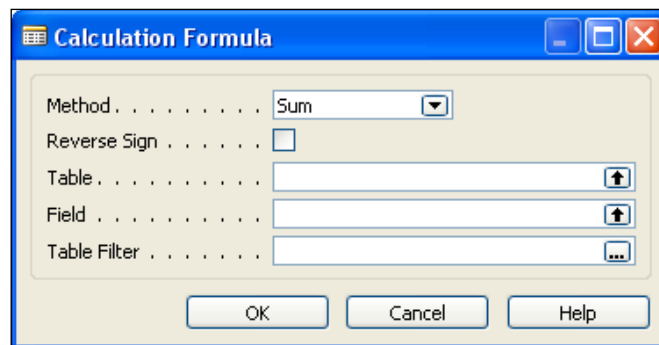


Because a FlowField does not contain actual data, it cannot be used as a field in a key, that is, you cannot sort on a FlowField. You also cannot define a FlowField that is based on another FlowField.

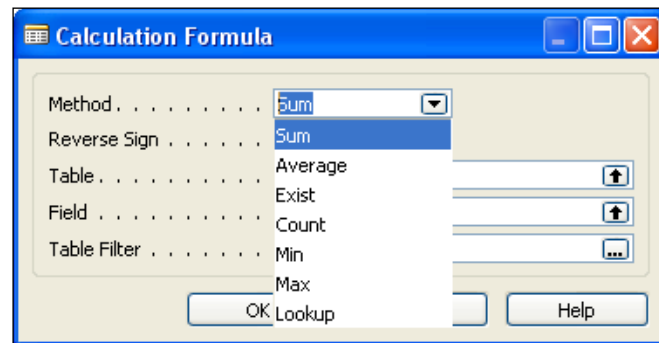
When a data item has its FieldClass set to `FlowField`, another directly associated property becomes available: **CalcFormula** (conversely, the **AltSearchField**, **AutoIncrement**, and **TestTableRelation** properties disappear from view when FieldClass is set to `FlowField`). The `CalcFormula` is the place where you can define the formula for calculating the FlowField. This formula consists of five components as follows:

1. FlowField type (aka Method)
2. Sign control (aka Reverse Sign)
3. Table
4. Field
5. Table Filter

On the CalcFormula property line, there is an ellipsis button displayed. Clicking on that button will bring up a form similar to the following screenshot:



The following screenshot shows seven FlowField types:



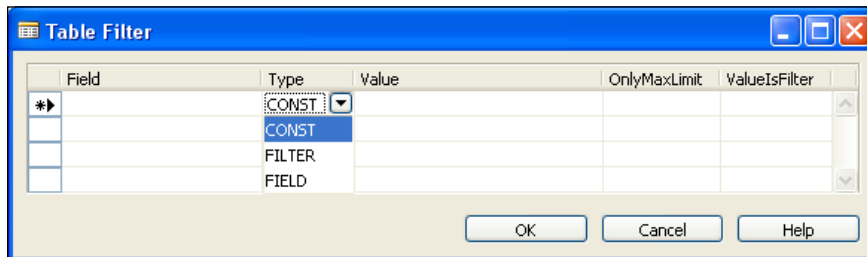
The explanation of the seven FlowFields is given in the following table:

FlowField type	Field data type	Description (in all cases, it applies to the specified set within a specific column in a table—field)
Sum	Decimal	The sum total
Average	Decimal	The average value (the sum divided by the count)
Exist	Boolean	Yes or No, does an entry exist?
Count	Integer	The number of entries that exist
Min	Any	The smallest value of any entry
Max	Any	The largest value of any entry
Lookup	Any	The value of the specified entry

The **Reverse Sign** control allows you to change the displayed sign of the result for FlowField types **Sum** and **Average** only; the underlying data is not changed.

Table and **Field** allow you to define the Table and the Field within that table to which your Calculation Formula will apply. When you make the entries in your **Calculation Formula** screen, there is no validation checking by the compiler that you have chosen an eligible table-field combination. That checking doesn't occur until run time. Therefore, when you are creating a new FlowField, you should test it as soon as you get it defined.

The last, but by no means the least significant, component of the FlowField calculation formula is the **Table Filter**. When you click on the ellipsis in the table filter field, the window shown in the following screenshot will appear:



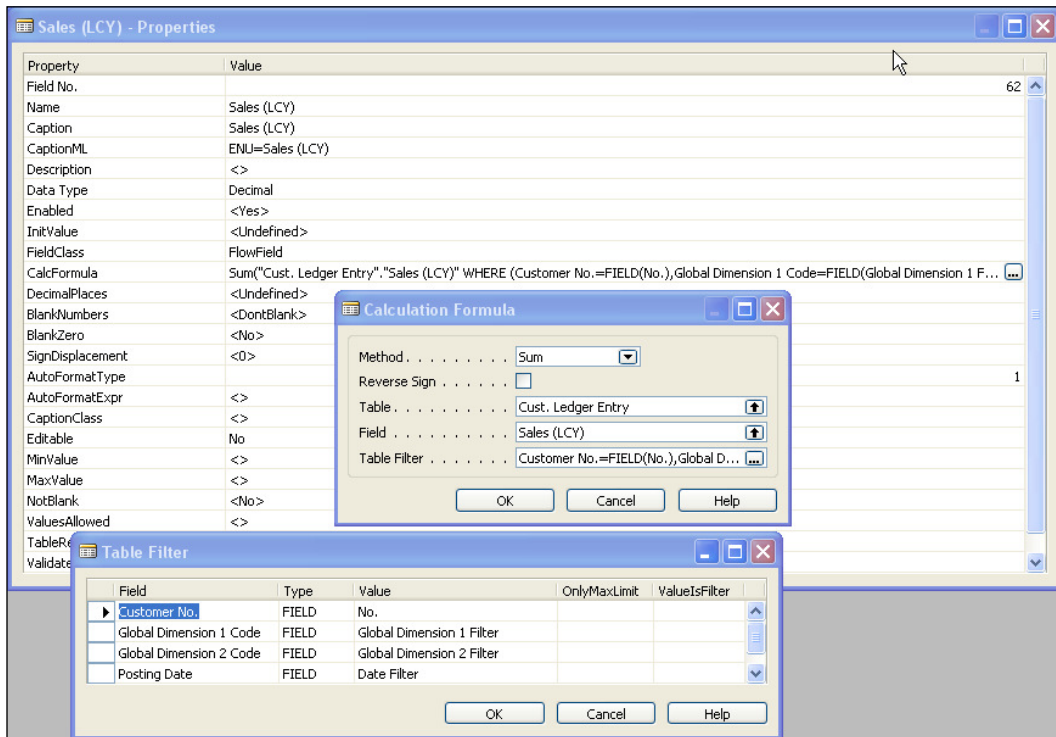
When you click on the **Field** column, you will be invited to select a field from the table that was entered into the **Table** field earlier. The **Type** field choice will determine the type of filter. The **Value** field will have the filter rules you define on this line, which must be consistent with the **Type** choice. The explanation is given in the following table:

Filter type	Value	Filtering action	OnlyMax-Limit	Values-Filter
Const	A constant which will be defined in the Value field	Uses the constant to filter for equally valued entries		
Filter	A filter which will be spelled out as a literal in the Value field	Applies the filter expression from the Value field		
Field	A field from the table within which the FlowField exists	Uses the contents of the specified field to filter for equally valued entries	False	False
		If the specified field is a FlowFilter and the OnlyMaxLimit parameter is True, then the FlowFilter range will be applied on the basis of only having a Max-Limit, that is, having no bottom limit. This is useful for the date filters for Balance Sheet data. (See Balance at Date field in the G/L Account table for an example)	True	False
		Causes the contents of the specified field to be interpreted as a filter (See Balance at Date field in the G/L Account table for an example)	True or False	True

- **FlowFilters:** These do not contain permanent data. They contain filters on a per user basis, with the information stored in that user's instance of the code being executed. A FlowFilter field allows a filter to be entered at a parent record level by the user (for example, G/L Account) and applied (through the use of FlowField formulas, for example) to constrain what child data (for example, G/L Entry records) is selected.

A FlowFilter allows you to provide flexible data selection functions to the users in a way that is simple to understand. The user does not need to have a full understanding of the data structure to apply filtering in intuitive ways, not just to the primary data table, but also to the subordinate data. Based on your C/AL code design, FlowFilters can be used to apply filtering on more than one subordinate table. Of course, it is your responsibility as the developer to make good use of this tool. As with many C/AL capabilities, a good way to learn more is by studying standard code.

A number of good examples on the use of FlowFilters can be found in the **Customer** (Table 18) and **Item** (Table 27) tables. In the **Customer** table, some of the FlowFields using FlowFilters are Balance, Balance (LCY), Net Change, Net Change (LCY), Sales (LCY), and Profit (LCY). The **Sales (LCY)** FlowField FlowFilter usage is shown in the following screenshot:

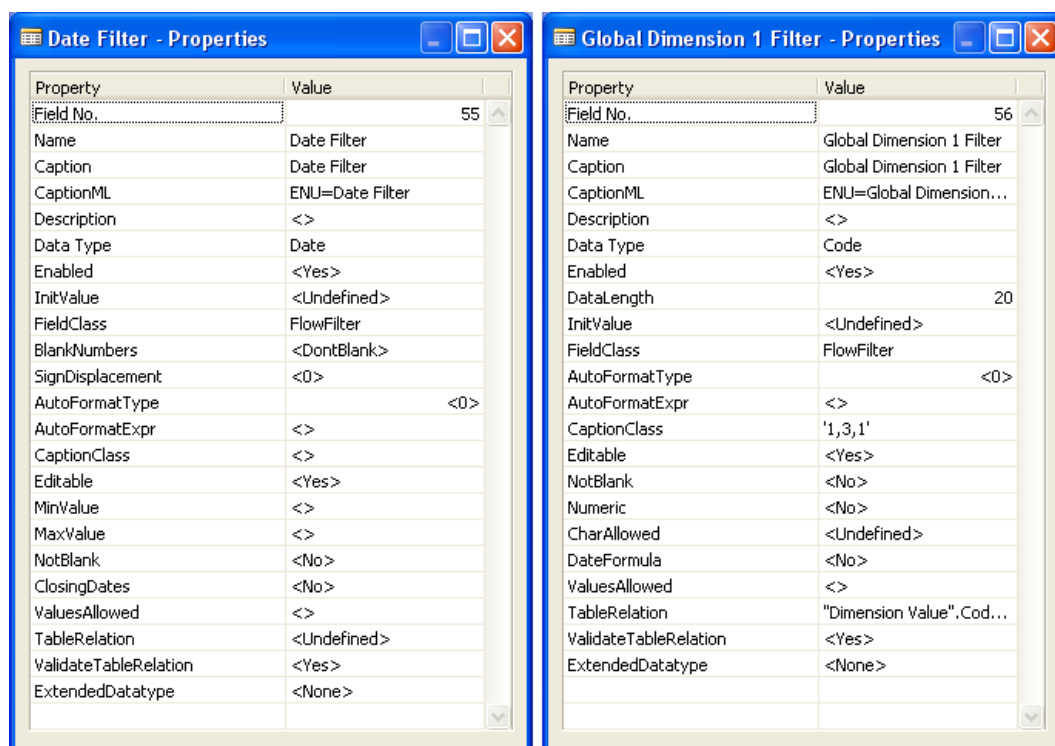


Similarly constructed FlowFields using FlowFilters in the **Item** table include Inventory, Net Invoiced Qty. Net Change, Purchases (Qty.), and a whole host of other fields.

Throughout the standard code, there are a number of FlowFilters that appear in most of the master table definitions. These are the Date Filter and Global Dimension Filters (global dimensions are user-defined codes that facilitate the segregation of accounting data by meaningful business break-outs such as divisions, departments, projects, customer type, and so on). Other FlowFilters that are widely used in the standard code, for example, related to Inventory activity, are Location Filter, Lot No. Filter, Serial No. Filter, and Bin Filter.

The properties defined for FlowFilter fields, such as **Date Filter** in the following screenshot, are similar to those of Normal data fields. Take a look at the **Date Filter** field (a **Date FlowFilter** field) and the **Global Dimension 1 Filter** field (a **Code FlowFilter** field) in the **Customer** table. The **Date Filter** field property looks similar to a Normal **FieldClass** field.

The **Global Dimension 1 Filter** field property values are different than those of the **Date Filter** because of the data type and its attributes rather than the fact that this is a FlowFilter field.



Filtering

As mentioned earlier, filtering is one of the powerful tools within NAV. Filtering is the application of defined limits on the data that is to be considered in a process. Filter structures can be applied in at least three different ways, depending on the design of the process. The first way is for the developer to fully define the filter structure and the value of the filter. This might be done in a report designed to show only information on a selected group of customers; for example, those with an open balance on account. The **Customer** table would be filtered to report only customers who have an outstanding balance greater than zero.

The second way is for the developer to define the filter structure, but allow the user to fill in the specific value to be applied. This approach would be appropriate in an accounting report that was to be tied to specific accounting periods. The user would be allowed to define what period(s) was (were) to be considered for each report run.

The third way is the ad hoc definition of a filter structure and value by the user. This approach is often used for general analysis of ledger data where the developer wants to give the user total flexibility in how they slice and dice the available data.

It is quite common within the standard NAV applications and in the course of enhancements to use a combination of the different filtering types. For example, the report just mentioned that lists only customers with an open Balance on Account (via a developer-defined filter) could also allow the user to define additional filter criteria. Perhaps, the user wants to see only Euro currency-based customers, so would filter on the **Customer Currency Code** field.

Filters are an integral part of FlowFields and FlowFilters, two of the three Field Classes. These flexible and powerful tools allow the NAV designer to create pages, reports, and other processes that can be used under a wide variety of circumstances. In most of the systems, user inquiries (pages/forms and reports) and processes need to be quite specific to different data types and ranges. The NAV C/AL toolset allows you to create relatively generic user inquiries and processes, and then allow the user to apply filtering to fit their specific needs. Note that the user sees FlowFilters referred to as **Limit Totals** onscreen and both terms are in the user Help.

Defining filter syntax and values

Let us go over some common ways in which we can define filter values and syntax. Remember, when you apply a filter, you will only view or process records where the filtered data field satisfies the limits defined by the filter.



Application of filters and ranges may give varying results depending on Windows settings or the SQL Server collation setup.

Filtering on equality and inequality

In general, filters for relative equality or inequality values follow mathematical notation practices.

- Either an equal (=) sign or the absence of that sign filters for data "equal to" the filter value:

Data type - description	Example filters
Integer	=200
Integer	200
Text	Chicago
Text	" (two single quote marks)

- A greater than (>) sign filters for data greater than the filter value:

Data Type - description	Example filters
Integer	>200
Date	>10/06/09 or >10062009 or >10.06.09 or >10-06-09 (NAV is very flexible relative to date entry formatting)
Decimal	>450.50

- A less than (<) sign filters for data less than the filter value:

Data Type - description	Example filters
Integer	<150
Date	<10/07/10

- The equal sign can be combined with the greater than (>=) or less than (<=) signs to filter for data "greater than or equal to" OR "less than or equal to" the filter value:

Data Type - description	Example filters
Integer	<=100
Date	<=1/1/10
Date	>=1/1/09
Text	>= Grade B

- Not equal is represented by the combination of the "less than" (<) symbol plus the "greater than" (>) symbol to filter for data not equal to the filter value:

Data Type - description	Example filters
Integer	<>1
Date	<>12/31/09
Boolean	<>yes (an awkward way of stating "No")

Filtering by ranges

NAV has very flexible range filtering capabilities.

- Ranges are defined by an expression containing two dots in a row (in other words ".."). Ranges are inclusive, that is the maximum and minimum values are included within the range. Ranges have three variations. The first is the from-to version which includes both a bottom end or minimum of the range and a top end or maximum:

Data Type - description	Example filters
Integer	1..10
Date	5/1/09..5/31/09
Text	Jones..Smith
Decimal	100.01..199.99

- The second range variation consists of the range operator (the two dots "..") plus a range maximum. This means "give me all the values from the lowest possible value up to and including the range maximum". This is generally the same as using the less than or equal to (<=) format:

Data Type - description	Example filters
Integer	..10 (Gives the same results as <=10)
Date	..12/31/09
Decimal	..99.99

- The third range variation consists of a lower limit (minimum) value followed by the range operator (".."):

Data Type - description	Example filters
Integer	100.. (Gives the same results as >=100)
Date	1/1/09..
Decimal	100000.00..

Filtering with Boolean operators

Filtering with Boolean operators is powerful, but must be done carefully and tested carefully.

- There are two Boolean operators. The operators are the ampersand sign (&) representing the logical AND operation and the pipe symbol (|) representing the logical OR operation. AND (&) operators are calculated before OR (|) operators.
- The OR operator is used to create a discontinuous set of allowed values:

Data Type - description	Example filters
Integer	5 10 15 20 (This will give you matches on all four of the stated values, but only on those values.)
Date	10/1/09 11/1/09 12/1/09 (This filter will pass through records dated on the first date of the three months)

- The AND operator can generally only be used in combination with other filtering operators:

Data Type - description	Example filters
Integer	(>=100) & (<=1000) (Gives the same result as the range 100..1000)
Date	C12-31-08 & C12-31-09 (data for the closing dates for two years)
Boolean	<>Yes & >12-31-09

Filtering with wildcards

- There are three wildcard characters that can be used within filter constructs. Wild cards only apply to string data. Although some limits on the use of wildcards are defined in the **Help**, the specifics of what constitutes a wildcard are not clearly defined.
- Asterisk (*) represents any character and any number of characters:

Data Type - description	Example Filters
Text	*st* (Includes all data containing the lowercase letters 'st')
Text	st* (Includes only the data starting with the lowercase letters 'st')
Text	*st (Includes only the data ending with the lowercase letters 'st')

- Question mark (?) represents any character, but only one character:

Data Type - description	Example filters
Text	?st? (Includes all data which is four characters long with the middle two characters being the lowercase letters 'st')
Text	????st (Includes all data which is exactly six characters long ending with the lowercase letters 'st')

- At symbol (@) eliminates case sensitivity for the value following it. The @ is often used in combination with the asterisk to make the filter value satisfy a wider range of data:

Data Type - description	Example filters
Text	*@st* (Includes all data containing any of the strings 'st', 'St', 'ST' or 'sT')
Text	@*st* (Gives the same results as the previous example)

Filtering with combinations

Many of these filter constructs can be used in combination. Be sure to thoroughly test your creations before inflicting them on unsuspecting users. It is relatively easy to create a filter, which on initial thought seems logical, but won't work the way you thought it would. In addition, the C/AL compiler routine which interprets filters is not perfect. It can get confused or just fail.

Be very cautious about using combinations that contain wildcards, especially (but not limited to) those expressions containing both wildcards and Boolean operators. Be very cautious about constructing filters based on exclusions. Generally, the limited "inclusive" approach works better. For example, you might want to print a Customer list excluding all Customers for the Salespeople with codes of JR and MD.

You might try to create a filter on Salesperson Code such as **<> (JR & MD)**. The C/SIDE routine that checks filters will not accept that as a valid entry. The same goes for the attempt to put in two separate filter entries as (only one filter string is allowed per data field). See if you can figure out what will work properly before you read on.

Let us assume all our Salesperson Codes are just two characters long. You could create a filter on the Salesperson Code in the page **(..JQ) | (JP..MC) | (ME..)**. This translates to all of the Customers having either a Salesperson Code less than or equal to JQ or (the pipe symbol: |) from JP to MC or greater than or equal to ME. In other words, all the two character codes except JR and MD. While that rather complicated solution works, so does the much simpler and more general **(<>JR) & (<>MD)**.

Experimenting with filters

Now it's time for you to do some creative experimenting with filters. We want to accomplish several things through our experimentation. Our first purpose is to get more comfortable with how filters are entered. Secondly, we want to see the effects of different types of filter structures and combinations. If we had a database with a large volume of data, we could also test the speed of filtering on fields in keys and on fields not in keys. But the amount of data in the Cronus database is small and our computers are very fast, so any speed differences will be difficult to see.

We could experiment on any report that allows filtering. To give us some options for our experimentation, we will use the **Customer/Item** List. This will report which Customer purchased what Items. The **Customer/Item** List can be accessed on the **Departments** menu via **Sales & Marketing | Sales | Reports | Customer | Customer/Item Sales**.

When you initially run the **Customer/Item Sales**, you will see just three data fields listed for entry of filters on the **Customer** table as shown in the following screenshot:

The screenshot shows the 'Customer/Item Sales' window. At the top is a blue title bar with the text 'Customer/Item Sales' and a close button. Below the title bar is a toolbar with a lightning bolt icon and a dropdown menu labeled 'Actions'. The main area is divided into sections. The first section is 'Options' with two checkboxes: 'New Page per Customer:' and 'Print to Excel:'. The second section is 'Customer' with a 'Sorting:' dropdown set to 'No.' and a sort order icon. Below this is the 'Show results:' section, which contains three filter rules: 'Where No. is Enter a value', 'And Search Name is Enter a value', and 'And Customer Posting Group is Enter a value'. There is a green plus icon and 'Add Filter' text below these rules. The 'Limit totals to:' section also has a green plus icon and 'Add Filter' text. At the bottom is a 'Value Entry' dropdown menu. The window ends with 'Print', 'Preview', and 'Cancel' buttons.

There are also three data fields listed for the entry of filters on the **Value Entry** table as shown in the following screenshot (which has both FastTabs expanded so you can see all the predefined filter entry options):

The screenshot shows a software window titled "Customer/Item Sales". It has a blue header bar with a lightning bolt icon and a dropdown menu labeled "Actions". Below the header, there are three main sections: "Options", "Customer", and "Value Entry".

The "Options" section contains two checkboxes: "New Page per Customer:" and "Print to Excel:", both of which are currently unchecked.

The "Customer" section has a "Sorting:" label followed by a dropdown menu set to "No." and a small icon with "A" and "Z" and a downward arrow. Below this, there is a "Show results:" section with three filter entries, each preceded by a red "X" icon and the word "Where" or "And" in blue. The first entry is "No." followed by a dropdown menu, "is", and "Enter a value". The second entry is "Search Name" followed by a dropdown menu, "is", and "Enter a value". The third entry is "Customer Posting Group" followed by a dropdown menu, "is", and "Enter a value". Below these entries is a green plus icon followed by the text "Add Filter".

Below the "Customer" section is a "Limit totals to:" section with a green plus icon followed by the text "Add Filter".

The "Value Entry" section has a "Show results:" section with three filter entries, each preceded by a red "X" icon and the word "Where" or "And" in blue. The first entry is "Item No." followed by a dropdown menu, "is", and "Enter a value". The second entry is "Inventory Posting Group" followed by a dropdown menu, "is", and "Enter a value". The third entry is "Posting Date" followed by a dropdown menu, "is", and "Enter a value". Below these entries is a green plus icon followed by the text "Add Filter".

At the bottom of the window, there are three buttons: "Print", "Preview", and "Cancel".

In each case, these are the fields that the developer determined should be emphasized. If you run the report without any filters at all, using the standard Cronus data, the contents of the first page of the report will resemble the following screenshot:

Print Preview

Customer/Item Sales

1 of 1 100% Find | Next

2/16/2009 2:51 PM
Page 1
ISAAC\Dave

Customer/Item Sales
Period:
CRONUS International Ltd.
All amounts are in LCY

Item No.	Description	Invoiced Quantity	Unit of Measure	Amount	Discount Amount	Profit	Profit %
01445544	Progressive Home Furnishings Phone No.						
1928-S	AMSTERDAM Lamp	14	PCS	498.40	0.00	109.20	21.90
1972-S	MUNICH Swivel Chair,	1	PCS	123.30	0.00	27.20	22.10
1988-W	CALGARY Whiteboard	1	PCS	877.32	97.48	168.72	19.20
	Progressive Home Furnishings			1499.02	97.48	305.12	20.4
10000	The Cannon Group PLC Phone No.						
1920-S	ANTWERP Conferenc	1	PCS	420.40	0.00	92.40	22.00
1964-W	INNSBRUCK Storage	10	PCS	2,920.00	0.00	1,208.00	41.40
1968-S	MEXICO Swivel Chair,	3	PCS	361.40	18.50	63.10	18.00
1996-S	ATLANTA Whiteboard,	7	PCS	6,029.56	317.34	1,079.16	17.90
70011	Glass Door	5	PCS	361.50	0.00	177.00	49.00
	The Cannon Group PLC			10082.86	335.84	2619.66	26
20000	Selangorian Ltd. Phone No.						
1896-S	ATHENS Desk	0	PCS	0.00	0.00	0.00	0.00
1928-S	AMSTERDAM Lamp	5	PCS	172.66	5.34	33.66	19.50
766BC-C	CONTOSO Storage Sy	0	PCS	0.00	0.00	0.00	0.00

If you want to print information only for customers whose names begin with the letter A, your filter will be very simple, similar to the following screenshot:

The screenshot shows a software window titled "Customer/Item Sales". At the top, there is a blue header bar with a lightning bolt icon and the word "Actions". Below this is a section labeled "Options" with two checkboxes: "New Page per Customer:" and "Print to Excel:", both of which are currently unchecked. The main section of the window is titled "Customer" and contains a "Sorting:" dropdown menu set to "No." with a small icon showing a list sorted by the letter 'A'. Below the sorting section, there is a "Show results:" section with three filter conditions, each preceded by a red 'X' icon: "Where No. is Enter a value", "And Search Name is A*", and "And Customer Posting Group is Enter a value". Below these filters are two green plus icons, each followed by the text "Add Filter". At the bottom of the "Customer" section is a "Limit totals to:" section with a green plus icon and the text "Add Filter". The bottom of the window features a "Value Entry" section with a dropdown arrow. At the very bottom of the window are three buttons: "Print", "Preview", and "Cancel".

The resulting report will be similar to the following screenshot, showing only the data for the two customers on file whose names begin with the letter A.

Print Preview

Customer/Item Sales

Period: 2/16/2009 2:56 PM
CRONUS International Ltd. Page 1
ISAAC\Dave

All amounts are in LCY

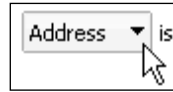
Customer: Search Name: A*

Item No.	Description	Invoiced Quantity	Unit of Measure	Amount	Discount Amount	Profit	Profit %
32656565	Antarcticoopy Phone No.						
1960-S	ROME Guest Chair, gr	7	PCS	875.70	0.00	193.20	22.10
1968-S	MEXICO Swivel Chair,	4	PCS	493.20	0.00	108.80	22.10
1976-W	INNSBRUCK Storage	5	PCS	1,152.46	128.05	399.46	34.70
70011	Glass Door	1	PCS	61.46	10.84	24.55	40.00
	Antarcticoopy			2582.81	138.89	726.01	28.1
49633663	Autohaus Mielberg KG Phone No.						
1896-S	ATHENS Desk	0	PCS	0.00	0.00	0.00	0.00
1906-S	ATHENS Mobile Ped	1	PCS	281.40	0.00	61.90	22.00
1968-S	MEXICO Swivel Chair,	0	PCS	0.00	0.00	0.00	0.00
1972-S	MUNICH Swivel Chair,	0	PCS	0.00	0.00	0.00	0.00
	Autohaus Mielberg KG			281.4	0	61.9	22
Total				2864.21	138.89	787.91	27.5

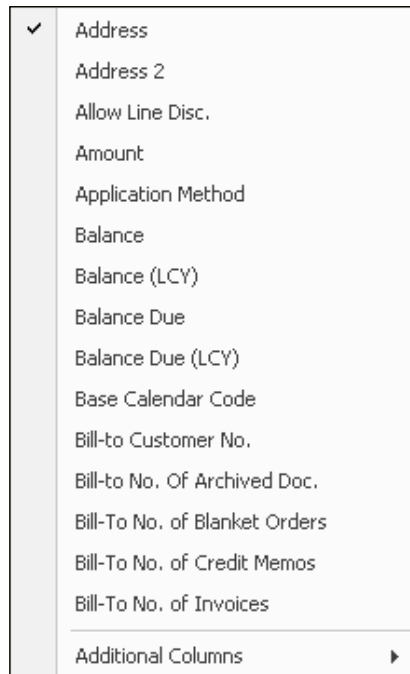
If you want to expand the customer fields to which you can apply filters, you can access the full list of other fields in the customer table. First, click on the **Add Filter** button as shown (actually, you could skip this step):



Then click on the dropdown list symbol next to the field that displays in response to your previous click, unless, of course, that field is the one you wanted (the same dropdown symbol appears next to all filter field options):



Next, you will see an extended list of table fields available for filtering:



Finally, if the total set of fields available is longer than this display allows, there is an **Additional Columns** entry at the end of this list. Clicking on that entry will lead you to a scrollable list of the rest of the fields in the table on which filters can be applied. Notice that, unlike previous versions of NAV, such lists are now in alphabetical order, based on the field names.

Customer/Item Sales

⚡ Actions ▾

Options

New Page per Customer: ☐

Print to Excel: ☐

Customer

Sorting: No. ▾

Show results:

✖ Where Search Name ▾ is A*

✖ And No. ▾ is Enter a value

✖ And Customer Posting Group ▾ is Enter a value

✖ And Address ▾ is Enter a value

+ Add Filter

Limit totals to

+ Add Filter

Address

Address 2

Allow Line Disc.

Amount

Application Method

Balance

Balance (LCY)

Balance Due

Balance Due (LCY)

Base Calendar Code

Bill-to Customer No.

Bill-to No. Of Archived Doc.

Bill-To No. of Blanket Orders

Bill-To No. of Credit Memos

Bill-To No. of Invoices

Additional Columns

Bill-To No. of Orders

Bill-To No. of Pstd. Cr. Memos

Bill-To No. of Pstd. Invoices

Bill-To No. of Pstd. Return R.

Bill-To No. of Pstd. Shipments

Bill-To No. of Quotes

Bill-To No. of Return Orders

Block Payment Tolerance

Blocked

Budgeted Amount

Chain Name

City

Collection Method

Combine Shipments

Comment

Contact

Contract Gain/Loss Amount

Copy Sell-to Addr. to Qty From

Country/Region Code

County

Cr. Memo Amounts

Cr. Memo Amounts (LCY)

Credit Amount

Credit Amount (LCY)

Credit Limit (LCY)

Currency Code

Customer Disc. Group

Customer Posting Group

Customer Price Group

Debit Amount

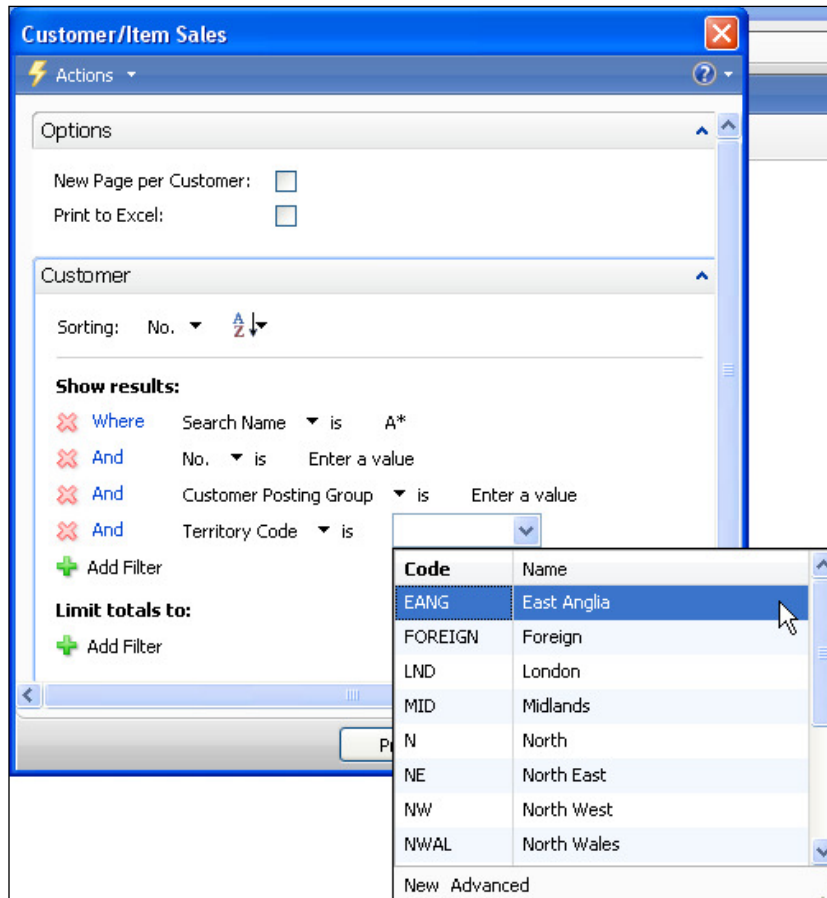
Debit Amount (LCY)

Department Code

E-Mail

Fax No.

From these lists we can choose one or more fields and then enter filters on those fields. If we chose **Territory Code**, for example, then the Request Page would look similar to the following screenshot. And if we clicked on the lookup arrow in the **Filter** column, a screen would pop-up allowing us to choose data items from the related table, in this case, **Territories**.



This particular Request Page has FastTabs for each of the two primary tables in the report. Click on the **Value Entry** FastTab to filter on the item-related data. If we filter on the **Item No.** for Item No's that contain the letter W, the report will be similar to the following screenshot:

Print Preview

Customer/Item Sales

Period: 2/16/2009 4:20 PM
Page 1
ISAAC\Dave

All amounts are in LCY

Value Entry: Item No.: *W*

Item No.	Description	Unit of Measure	Invoiced Quantity	Amount	Discount Amount	Profit	Profit %
01445544	Progressive Home Furnishings Phone No.						
1988-W	CALGARY Whiteboard	1 PCS		877.32	97.48	168.72	19.20
	Progressive Home Furnishings			877.32	97.48	168.72	19.2
10000	The Cannon Group PLC Phone No.						
1964-W	INNSBRUCK Storage	10 PCS		2,920.00	0.00	1,208.00	41.40
	The Cannon Group PLC			2920	0	1208	41.4
30000	John Haddock Insurance Co. Phone No.						
8908-W	Computer - Highline P	3 PCS		342.60	0.00	342.60	100.00
8924-W	Server - Enterprise Pa	1 PCS		346.30	0.00	346.30	100.00
	John Haddock Insurance Co.			688.9	0	688.9	100
31987987	Candoxy Nederland BV Phone No.						
1928-W	ST.MORITZ Storage U	0 PCS		0.00	0.00	0.00	0.00
1952-W	OSLO Storage Unit/Sh	0 PCS		0.00	0.00	0.00	0.00

If we want to see all of the items containing either the letter W or the letter S, our filter would be *W* | *S*. If you made the filter w | s, then you would get only entries equal exactly to W or to S because we didn't use any wildcards.

You should go back over the various types of filters we discussed and try them all. Then you should try some combinations. Get creative! Try some things that may or may not work and see what happens. Explore a variety of reports or list pages in the system by applying filters to see what happens. A good page on which to apply filters is the Customer List (**Sales & Marketing** menu | **Sales** | **Customers**). This is a safe learning experience (you can't hurt anything or anyone).

Accessing filter controls

The NAV User Interface has two very different approaches to setting up filtering; one for the Classic Client and the other for the RoleTailored Client. Because you will be developing and doing some basic testing in the Classic Client, we will cover that approach. Because you will be targeting most of your development for use in the RoleTailored Client, you need to be totally comfortable filtering there.

Classic Client filter access

There are four buttons at the top of the screen that relate to filtering, plus one for choosing the active key. Depending on the display on your system, they will look like those in the following screenshot:



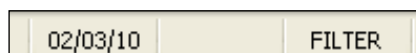
From left to right, they are as follows:

- **Field Filter** (*F7*): Highlight a field, press *F7* (or select **View | Field Filter**), and the data in that field will appear in a display ready for you to define a filter on that data field. You can edit the filter in any way before you click on **OK**.
- **Table Filter** (*Ctrl+F7*): Press the *Ctrl* key and *F7* simultaneously (or select **View | Table Filter**). You will be presented with a page that allows you to choose any number of fields in the left column and, in the right column, enter filters to apply to those fields. Each of these individual filters is a Field Filter—the same as would have been applied using the Field Filter option just described. The filters for the individual fields are "ANDed" together (that is, they all apply simultaneously). If you invoke the Table Filter page when any Field Filters are already applied, they will be displayed.
- **Flow Filter** (*Shift+F7*): Press the *Shift* key and *F7* simultaneously (or select **View | Flow Filter**). You will be presented with a page that allows you to choose any number of fields in the left column and in the right column, enter filters to use with those fields. On initial display, it will show all the Flow Filter fields available. For any Flow Filter field, you can enter a filter, which will then be applied to the underlying data for FlowFields whose definition includes a constraint by that particular Flow Filter field.

You can also use this page to enter Field Filters, but you will not be able to see the field filters that are already in effect via this page. To remove Flow Filters, you must call up this page and manually remove the filters, by deleting the filter lines or at least the filter values.

- **Show All** (*Shift+Ctrl+F7*): This will remove all Field Filters, but will not remove any Flow Filters.
- **Sort** (*Shift+F8*): This allows you (or your user) to choose which key is active on a displayed data list (unless the underlying C/AL code overrules). By properly choosing a key that contains the field on which you wish to filter, you can significantly affect the speed of the filtering process. Of course, this is true for filtering processes coded in C/AL as well.

When you are viewing a form and want to check if the filters are in effect, check the bottom of the screen for the word **FILTER** as shown in the next screenshot:

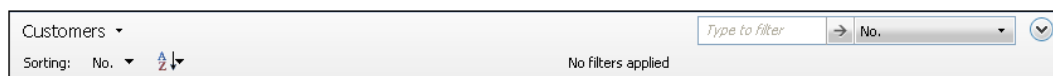


In a page, refer to the Filter Pane for the current filter status.

RoleTailored Client filter access

The basic logic of filtering within the RoleTailored Client is the same as that in the Classic Client. However, the method of accessing fields to use in filtering is quite different. The appearance of the filter definition page segments is completely different.

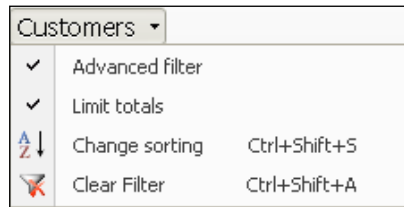
When a page such as the Customer List is opened, the filter section at the top of the page looks like the following screenshot. On the upper right corner is a place to enter single-field filters. This is the **Type to** (aka Quick) filter, essentially equivalent to the Field Filter in the Classic Client. The fields shown as available for filtering are the same as the visible columns showing in the List. On the left is a Sorting field which allows the choice of keys followed by a control to allow the choice of Ascending or Descending for the sort.



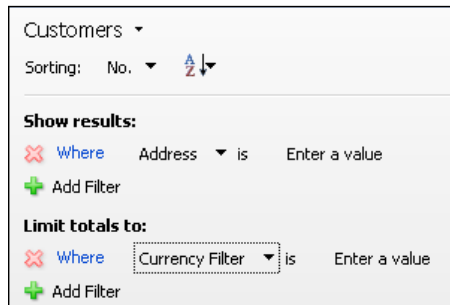
If you click on the chevron button in the far right-upper corner to expand the Filter Pane, the result will look similar to the following screenshot. The filter display now includes an additional filtering definition field, in this case identified as allowing entry of **Limit totals to** filters.



If you go to the Filter Pane header line (where it has the name of the Page) and click on the drop-down symbol, you will see a set of selection options (the filter menu) like the following screenshot shows. The Advanced filter provides for the entry of multiple field filters (essentially the same as the Classic Client Table Filter). The Limit Totals filter provides for the entry of FlowFilter related constraints (essentially the same as the Classic Client Flow Filter entry). This is also one place you can clear filters of all types (you can simply enter *Ctrl+Shift+A* as indicated in the filter menu). Choosing the **Change sorting** option simply puts focus on the key field in the Sorting part.



If, as shown by the checkmarks in the preceding screenshot, you have selected both **Advanced filter** and **Limit totals**, your Filter Pane will look somewhat like the following screenshot. It will display both the **Show results** filter section (Advanced filter) and the **Limit totals to** filter section (Limit totals filter). If filters have been entered, they will show here.



For all practical purposes, there are no limits (other than logical ones) to the number or combinations of filters that you can apply to a particular list display in NAV.

Find-As-You-Type filtering

Field Groups (described in Chapter 2, *Tables*) also provide a form of filtering, known as **Find-as-you-type** filtering. Find-as-you-type (aka filter-as-you-type) works for user entries into fields which have a TableRelation defined.

The find-as-you-type capability is not enabled by default, so it must be enabled by the system administrator adding the following line into the `customsettings.config` file:

```
<add key="FilterAsYouTypeAutomaticLookupEnabled" value="true"></add>
```

See the C/SIDE Help for additional information in the **Configuring the Role Tailored Client** Help.

When enabled, find-as-you-type operates by displaying a drop down list filtered by what characters the user has typed thus far. The fields displayed are determined either by default or by the Field Groups defined for the referenced table. The filtered (aka search) field can be selected from the displayed columns by using the arrows keys to scroll left or right through the columns in the drop down list. When the user sees the entry they want to use, they can highlight and select it.

Summary

In this chapter, we focused on the basic building blocks of NAV data structure, fields, and their attributes. We reviewed the types of data fields, properties, and trigger elements for each type of field. Then, we walked through a number of examples to illustrate most of these elements, though we have postponed exploring triggers until we had enough knowledge of C/AL coding techniques to make that worthwhile.

The data type and FieldClass determine what kind of data can be stored in a field. When you combine the table structure with properly designed fields, the essence of your application system design is defined. In this chapter, we covered the broad range of data type options as well as the Field Classes.

We also considered some examples of different types and classes and discussed how they are used in an application. We dug into the date calculation tool that gives C/AL an edge in business applications. We also discussed filtering in some detail, how filtering is considered as we design our database structure, and how the users will access data. Finally, more of our NAV application was constructed with some features worth emulating in your own future designs.

In the next chapter, we will look at Pages in more detail and see how we can design Pages to take advantage of the data structures we have now put in place.

Review questions

1. A C/AL Name can be up to 50 characters long. True or False?
2. A C/AL Caption is used to support the multi-language feature of NAV. True or False?
3. The Table Relation property defines the reference of a data field to a table. The related table data field must be: (choose one)
 - a. In any key in the related table
 - b. Defined in the related table, but not in a key
 - c. In the Primary Key in the related table
 - d. The first field in the primary key in the related table
4. The new ExtendedDataType property supports designation of any one of three of the following data types, displaying an appropriate action icon. Choose three:
 - a. Email address
 - b. Website URL
 - c. GPS location
 - d. Telephone number
 - e. Country
5. Field numbers can readily be changed at any time. True or False?
6. Data can be easily moved from one data type to another: (choose one)
 - a. Never
 - b. Sometimes
 - c. Always
7. Which of the following are complex data types? Choose three.
 - a. Records
 - b. Strings of text
 - c. DateFormula
 - d. DateTime data
 - e. Objects

8. Every table must have a Primary Key. A Primary Key entry can be defined as unique or duplicates allowed, based on a table property. True or False?
9. FlowFilter data is not stored in the database. True or False?
10. Filters should not use wildcard characters because the results are unpredictable. True or False (False – wildcards are a valuable tool).
11. Which three of the following are legal filter expressions?
 - a. Chicago AND Berlin OR London
 - b. (=1) AND (>=10)
 - c. (JQ) | (JP..MC) | (ME..)
 - d. a*..A*
 - e. 10-1-10..1/31/11

4

Pages—Tools for Data Display

We must welcome the future, remembering that soon it will be the past; and we must respect the past, remembering that it was once all that was humanly possible
— George Santayana

Pages are NAV 2009's new method of structuring information for presentation to its users. The previous versions of NAV used forms for screen displays. Forms were designed by first positioning all the controls in specific locations on the screen, and then providing the user with a very modest amount of flexibility for personalizing the layout. Pages are designed by defining the order of presentation of the elements within a portion of the display, allowing the page rendering routines to handle most of the control-positioning decisions. Once a page is rendered ("painted" on the display), the user has a great deal of flexibility for personalization in order to define what is displayed and how it is organized in the display. As pages are rendered external to applications, rendering routines can be created for display formats other than the common video display. Display targets can be browser-based, SharePoint, mobile devices, **Windows Presentation Foundation (WPF)**, and so on. Pages are not compatible with the Classic Client.

There are several significant implications of the new user interface of Dynamics NAV 2009. In the past, the relative rigidity of the interface resulted in screen designs that were tied to the underlying data structures. This was true whether or not it resulted in a good design from the user's point of view. Part of that was simply due to the tendencies of the programmer/designer. Now, because of the flexibility, tailorability, variety of components, and the page/display rendering tools used, pages allow for a much more task-oriented design specific to each user's personal task set.

If the user interface brings attention to itself rather than the data, or makes it harder for the user to see critical patterns and trends in the data, then we don't have a good design. Page technology provides a much wider variety of ways to present data to fit the user's needs. As always, the designer/developer has the responsibility of using the tools to their best effect.

In this chapter, we will explore the various types of pages that NAV offers you. We will review many of the options for formatting, data access, and tailoring your pages. We will also learn about the Page Designer tools and the inner structures of pages.

What is a page?

Pages serve the purpose of both input and output. Pages are views of data or process information designed only for on-screen display. Pages can also be user data-entry vehicles. There are several types of pages, including Card, Card Part, List, List Part, List+, Role Center, Departments, Document, Journal/Worksheet, and NavigatePage (also used to create Wizard pages). Cards, Lists, List+, and Documents can be used both for inquiry and data entry.

Controls

Controls are the objects that display information on pages. This information can be data from the database, static material, pictures, or the results of a C/AL expression. Container controls, such as **Groups** and **FastTabs**, can contain other controls. Group controls make it easy for the developer to handle a set of contained controls as a group. A FastTabs control does the same, but it also makes it easy for the user to consider a set of controls as a group. The user can make all the controls on a FastTab visible or invisible by expanding or collapsing the FastTab. The user also has the option to show or not to show a particular FastTab as a part of the page customizing capability.

Bound and unbound

Pages can be created as **bound** (tightly associated with a specific table), or **unbound** (not associated with a table). Typically, a card or a list page will be bound, but role center pages will be unbound. Other instances of unbound pages are rare. An example of an unbound page is Page 476 – Copy Tax Setup.

Controls on an unbound page are generally also unbound. When a page is bound to a table, it is easy to tie the controls on the page to fields in the table. A bound page can also have unbound controls, that is, controls displaying computational results or values entered into working storage variables. Either category of page—bound or unbound—can have bound controls (that is, controls referring to tables other than the one to which the page is bound). Unbound pages are generally used for displaying information or processing status.

Pages—a stroll through the gallery

When you look at a page as a user with a developer's knowledge, the page type will be obvious most of the time. The specific layout and features of the page object available to you as a developer will offer many choices. Some pages require many design decisions, some require only a few. C/SIDE allows you to create pages with vastly different "look and feel" attributes. The standard NAV application only uses a few of the possibilities, and closely follows a set of **Graphical User Interface (GUI)** guidelines that provide consistency throughout the system. Those guidelines are described in an interactive document named Microsoft Dynamics NAV 2009 User Experience Guidelines (UX Guide for short). You should obtain a copy of this guide and study it.

Good design practice dictates that enhancements should integrate seamlessly unless there is an overwhelming justification for being different. In general, your new pages should have the same look and feel as the pages in the out of the box product. When you add changes to pages, to the extent the new functionality allows, the changes should have the same look and feel as the original page. This consistency makes support, maintenance, and training more efficient.

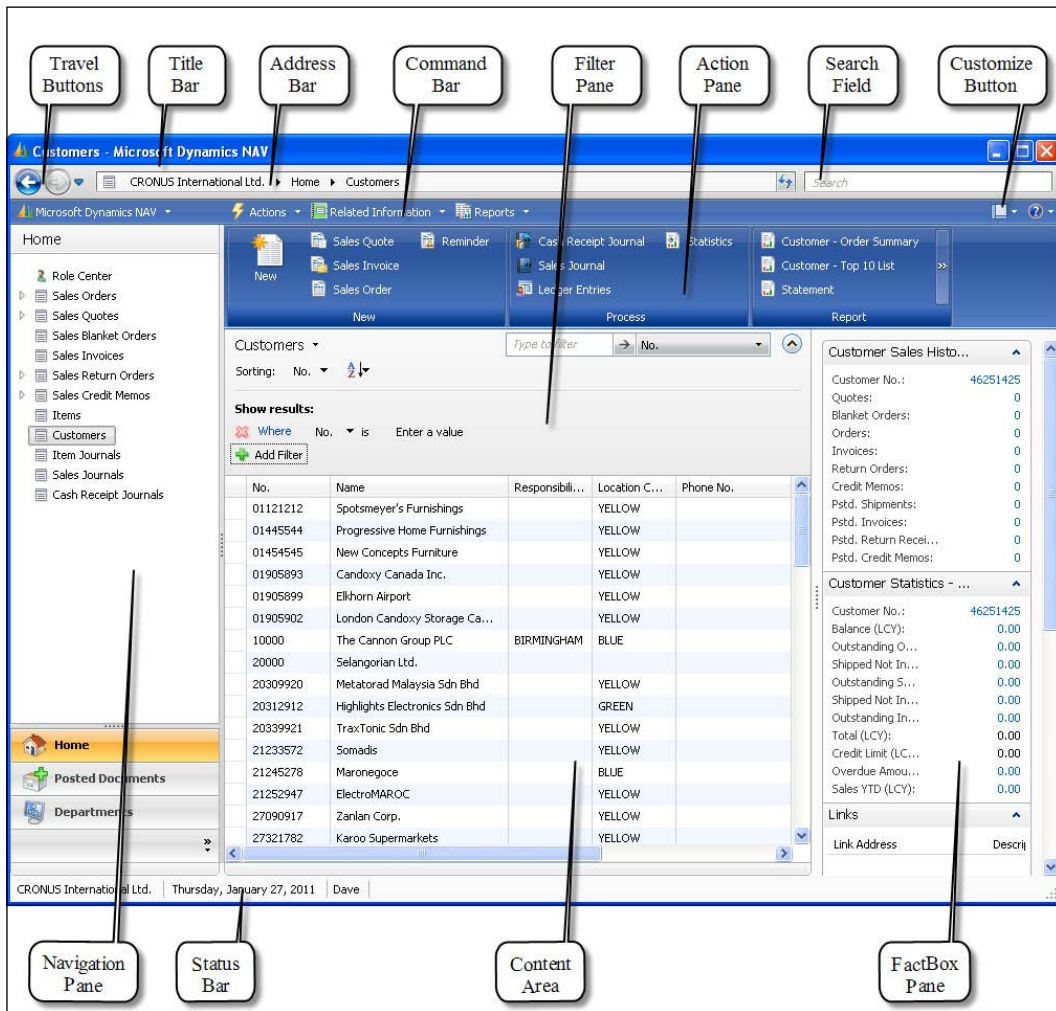
There will be instances where you will need to provide a significantly different page layout in order to address a special requirement. Perhaps, you need to provide two or more tabular displays on the same page. Maybe you need to use a special symbol to warn of a critical situation, or you need to create a screen layout for a display device that is significantly different from a standard desktop video display. In such cases, remember that the look and feel of the basic NAV pages has withstood the test of time for both usability and (reasonably) good taste. Even when you are going to be different, continue to be guided by the environment and context in which your new work will operate.

Before we begin reviewing the page types available in the NAV 2009 **RoleTailored Client** (familarly known as the **RTC**), let's look at the general screen layout of the RTC. We'll identify the components that surround the core page structure and then review each page type that presents content to the user.

The general layout shown in the following screenshot includes a list page at its core.

A sample RoleTailored Client page

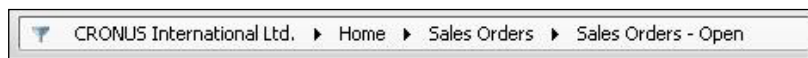
Let's take a look at what makes up a typical Role Tailored Client page.



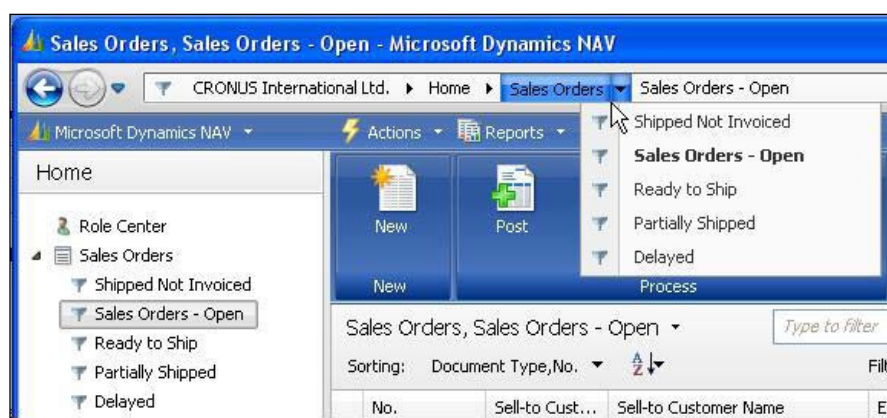
In the RTC, travel buttons serve the same purpose that they serve in Internet Explorer – to move backward or forward through the previously displayed pages.

The title bar displays the information defined in the applicable page properties.

The address bar displays the navigation path that led to the current display. It can be seen in the following screenshot:



If you click on one of the right-facing arrowheads in the address bar, the appropriate child menu options will be displayed in a drop-down list (see the following screenshot). In this instance, you can see that the list of options subordinate to **Sales Orders** is displayed both in the drop-down menu from the address bar, and in the detailed list of options in the navigation pane.

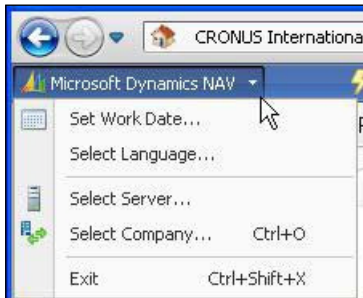


When you click on the address bar, the format of the display changes to a standard path format. The following screenshot shows an example of how the image changed after clicking on the address bar. That path can be copied, pasted, and thus reused in the same fashion as any Windows Explorer address path.



The command bar provides access to a standard set of menu options, which varies slightly based on what is in the content area.

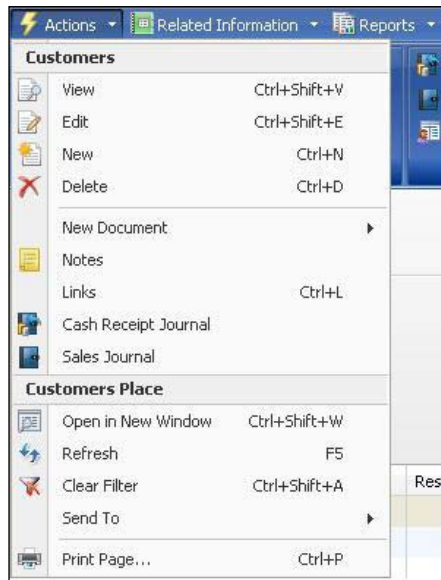
The left-most menu in the command bar is accessed by clicking on the button labeled **Microsoft Dynamics NAV**. It provides access (as shown in the next screenshot) to some basic administration functions, as shown in the next screenshot.



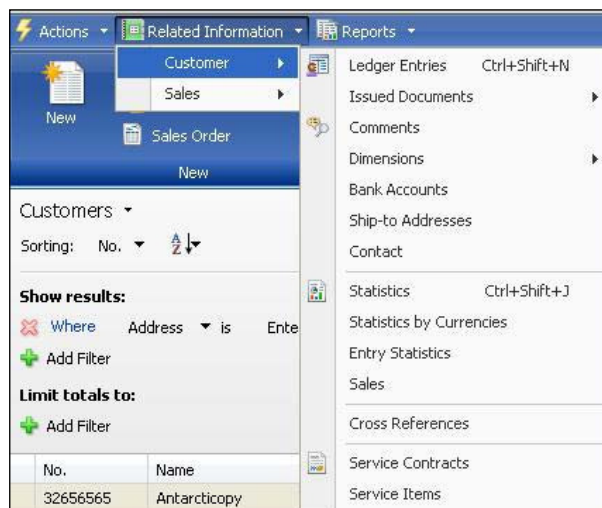
The center portion of the command bar can have one, two, or three menu buttons, depending on what the content area contains. The option choices in these menus also depend on the content area.



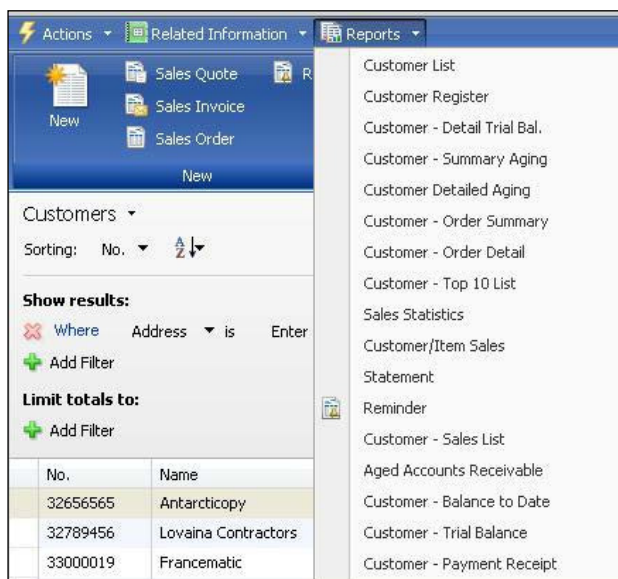
In the three screenshots that follow, you can see the menu options for the three menu buttons of the preceding screenshot, as they appear in the Customer List. The next image shows the Actions menu list:



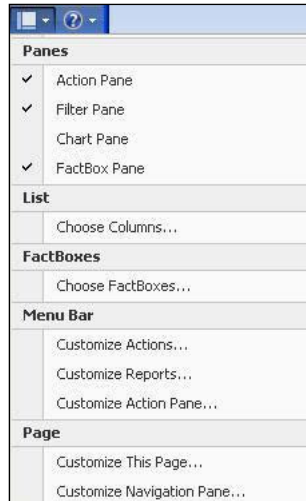
The following screenshot shows the options under **Related Information**:



The following screenshot shows the menu options under **Reports** of the **Customer List** page:



The right end of the command bar contains the **Customize** and **Help** menu buttons. The contents of the **Customize** menu are also dependent on what is in the content area. An example of the **Customize** menu is shown in the following screenshot:



The **Filter Pane** is where the user can enter the various types of data filters and the sort key choice to be applied to the page display. Filters can be either data (row) filters or flow filters. Data filters are displayed in the **Show results - Where** section, and flow filters are displayed in the **Limit totals to** section of the **Filter Pane**.

The **Action Pane** contains the commands that are most frequently used by the user. It provides the key section of role tailoring for the RoleTailored Client. These same commands will be duplicated in other menu locations, but are in the **Action Pane** for quick and easy access.

The **Navigation Pane** is similar to those in other Microsoft products, but with specific attributes consistent with the design philosophy of the RoleTailored Client. The standard **Navigation Pane** contains menu options based on the active Role Center (tied to the user's login). It also contains activity buttons, at least the **Home** and **Department** buttons.

The status bar shows the name of the active company, the work date, and the current user ID. If you click on the company name, you can change companies. If you click on the work date, you can change it.

The content area is the focus of this display. It may be a list page, or a card page, or a transaction-oriented page. It is the content.

The **FactBox Pane** appears on list and card structured pages. FactBoxes provide no-click access to related information about the data in focus.

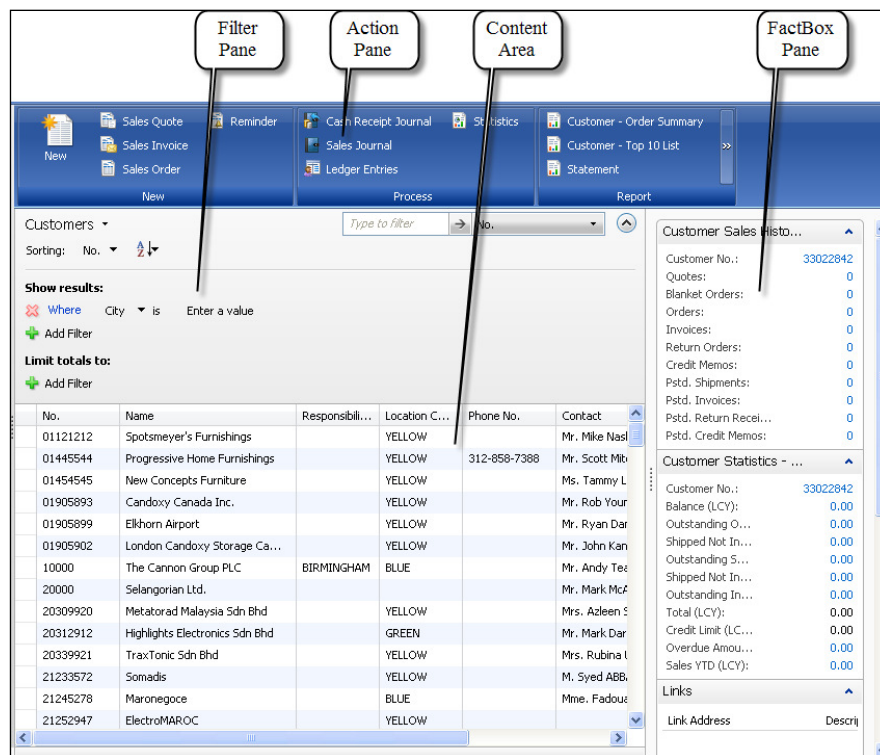
Types of pages

Let's briefly review the types of pages that we typically use in an application. Then we will create several examples for our ICAN system. From an application design point of view, we need to consider which page type to use under what circumstances.

List page

List pages display a list of any number of records (rows) at one time, one line per record, with each displayed data field shown as a column. For Reference table maintenance and inquiry, List pages are used. When a List page is initially selected, typically it is not editable. When you double-click on an entry in the viewing List, an editable Card page is generally displayed. In some cases, the List page itself becomes edit enabled. List pages are also used to show a list of master records in order to allow the user to compare records, or to easily choose one master record on which to focus. List pages may also contain a Filter pane. Specific List pages, such as Ledger Entries, allow editing of some fields (such as Invoice Due Dates). The menu options in the navigation pane target list pages (that is, not card pages, reports, and so on.).

The following screenshot shows a typical list page – the Customer list.



Card page

Card pages display and allow updating of a single record. A Card page is generally used for master tables and setup data. Complex cards can contain a number of FastTabs and may include display data from subordinate tables. A Card page should not include a list-formatted display (such as the one shown in a List page or the ListPart subpage shown in a Document page) on any of its FastTabs.

The screenshot shows a software window titled 'Edit - Customer Card - 10000 - The Cannon Group PLC'. The window has a menu bar with 'Actions', 'Related Information', and 'Reports'. Below the menu is a toolbar with icons for 'Sales Invoice', 'Sales Order', 'Reminder', 'Apply Template', 'Cash Receipt Journal', 'Sales Journal', 'Statistics', and 'Customer - Balance to Date'. The main area is divided into several sections: 'General' (containing fields for No., Name, Address, Post Code, City, Country/Region Code, Phone No., Primary Contact No., Contact, Search Name, Balance (LCY), Credit Limit (LCY), Salesperson Code, Responsibility Center, Service Zone Code, Blocked, and Last Date Modified), 'Communication' (containing fields for Phone No., Fax No., E-Mail, Home Page, and I/C Partner Code), 'Invoicing' (with tabs for NATIONAL and DOMESTIC), 'Payments' (with tabs for 1M(8D) and DOMESTIC), and 'Shipping' (with tabs for BLUE and DHL). On the right side, there is a 'Customer Sales History' panel showing a list of transactions with columns for Customer No., Quotes, Blanket Orders, Orders, Invoices, Return Orders, Credit Memos, Pstd. Shipments, Pstd. Invoices, Pstd. Return Re..., and Pstd. Credit Me... The window ends with an 'OK' button at the bottom right.

Document page

Document (task) pages have at least two FastTabs in a header/detail format. The FastTab at the top contains the header fields, followed by a FastTab containing multiple records in a list style format; for example, the format used in Sales Orders and Invoices, Purchase Orders and Invoices, Production Orders, and so on. This page type is appropriate whenever you have a parent record tied to a subordinate or child set of data in a one-to-many relationship.

FastTab

FastTabs are the collapsible/expandable replacements for traditional forms tabs.

FastTabs typically represent subject areas on a Card page or a Document page (such as a Sales Order). Important fields can be promoted to display on a FastTab when the tab is collapsed, allowing the user to see the needed data with minimal effort. A Sales Order example is shown in the following screenshot, with some FastTabs collapsed and some expanded. On several of the collapsed FastTabs, some promoted key fields are displayed. Such field displays disappear from the FastTab line when the FastTab is expanded.

Fast Tabs

101023 - John Haddock Insurance Co.

General 101023 | 30000 | 2/23/2011 | Open

Type	Item	Description	Location C...	Quantity	Reserved Qua...	Unit of Mea...	Unit Price
Item	1920-5	ANTWERP Conference Table	GREEN	4		PCS	
Item	1936-5	BERLIN Guest Chair, yellow	GREEN	23		PCS	

Invoicing

Bill-to Customer No.: 30000 Due Date: 2/28/2011

Bill-to Name: John Haddock Insurance Co. Payment Discount %: 0

Bill-to City: Manchester Pmt. Discount Date: 2/23/2011

Department Code: SALES Payment Method Code:

Project Code: Prices Including VAT: ☐

Payment Terms Code: CM VAT Bus. Posting Group: NATIONAL

Show more fields

Shipping MO2 4RT | 2/23/2011 | Partial

Foreign Trade

E-Commerce

Prepayment 0 | 2/28/2011

Customer Sales History

Customer No.: 30000

Quotes: 0

Blanket Orders: 0

Orders: 5

Invoices: 0

Return Orders: 0

Credit Memos: 0

Pstd. Shipments: 5

Pstd. Invoices: 2

Pstd. Return Recel...: 0

Pstd. Credit Memos: 0

Sales Line Details

Item No.: 1920-5

Availability: 58

Substitutions: 0

Sales Prices: 0

Sales Line Discounts: 1

Notes

Click here to create a new note.

OK

Fast Tabs

A Document page may have additional FastTabs with data fields and a FactBox pane. The following screenshot shows a Production Order document page, which has the requisite FastTab for header information (the **General** FastTab), a detailed FastTab (**Lines**) where multiple records can be displayed in list format, followed by two more FastTabs containing additional data (**Schedule** and **Posting**):

Edit - Firm Planned Prod. Order - 1010005 - Bicycle

Actions | Related Information | Reports

Process | Reports

1010005 · Bicycle

General

No.: 1010005 Search Description: BICYCLE

Description: Bicycle Quantity: 16

Description 2:

Source Type: Item Due Date: 1/30/2010

Source No.: 1000 Assigned User ID:

Last Date Modified:

Lines

Item No.	Due Date	Description	Starting Date-Time	Ending Date-Time	Quantity
1000	1/30/2010	Bicycle	1/26/2010 3:04 AM	1/29/2010 9:00 AM	16

Schedule

Starting Time: 10:04:00 AM Ending Time: 4:00:00 PM

Starting Date: 1/26/2010 Ending Date: 1/29/2010

Posting FINISHED

List+ page

A List+ page is similar to a Document page, as it will have at least one FastTab with fields and one FastTab at the bottom with a list page (grid) format. But a List+ page may have more than one FastTab with fields and one or more FastTabs with a list page format, while a Document page can only have a single list page style grid.

Journal/Worksheet page

Journal/Worksheet pages are widely used as transaction-entry pages. This page format consists of a list style section (such as multiple records or line by line) in the content area, followed by a section containing either additional line detail fields or totals. Data is entered into a Journal/Worksheet either manually or by a batch process.

The following screenshot shows a **Sales Journal** page.

Batch Name: DEFAULT

Posting Date	Document	Document	Account Type	Account No.	Description	Gen. Postin...	Gen. Bus. ...	Gen. Prod. ...	Amount	Ba
1/28/2010	Invoice	G01001	Customer	10000	The Cannon Group PLC				1,400.00	G/
1/28/2010	Invoice	G01001	Customer	20000	Selangorian Ltd.				650.00	G/
1/28/2010	Invoice	G01001	Customer	01454545	New Concepts Furniture				127.50	G/

Account Name: Selangorian Ltd. Bal. Account Name: Balance: 2,050.00 Total Balance: 2,121.09

OK

All of the page types discussed so far are bound pages associated with tables. These bound pages display the data from those tables. Such pages, when properly designed, are one of the best ways for the easy and efficient use of a NAV application.

Confirmation (Dialog) page

This is a simple display page embedded in a process; it is used to communicate with a user/operator. A sample Dialog page is as follows:

Details	
No.:	01454545
Name:	New Concepts Furniture
Balance (LCY):	222,241.32
Outstanding Amt. (LCY):	702.82
Shipped/Ret. Rcd. Not Invd. (LCY):	0.00
Current Amount (LCY):	72.48
Total Amount (LCY):	223,016.62
Credit Limit (LCY):	0.00
Overdue Amounts (LCY) as of 02/03/10:	222,241.32

Request page

This is a relatively simple page consisting of several tabs which allow information to be entered to control the execution of a report object. A sample Request page for the Customer/Item Sales report is shown in the following screenshot:

Options

New Page per Customer: ☒

Print to Excel: ☐

Customer

Value Entry

Show results:

Where Item No. is *W*

And Inventory Posting Group is Enter a value

And Posting Date is Enter a value

+ Add Filter

Navigate page

One of the early unique features of NAV (then Navision), the Navigate function is implemented using the NavigatePage page type.




This is a somewhat confusing set of terminology. The page that implements the Navigate function is internally identified and structured as a NavigatePage page type. The NavigatePage page type is also used to implement Wizards.

The **Navigate** page allows the user to view the summary of the number and type of posted entries having the same document number or posting date as a related entry or user-entered value. Navigate is a terrific tool for tracking down related entries. It can be productively used by a user, an auditor, or even a developer. A sample Navigate page is shown in the following screenshot:

Table Name	No. of Rec...
Posted Sales Invoice	1
G/L Entry	5
VAT Entry	2
Cust. Ledger Entry	1
Detailed Cust. Ledg. Entry	1
Value Entry	10

An unusual page feature of the Navigate page is the use of standard tabs (at the bottom of the page display) rather than FastTabs.

The NavigatePage page type is also used as the basis for Wizard pages within NAV. Some Wizard page examples are pages: 5077 – Create Interaction, 5097 – Create To-do, and 5126 – Create Opportunity. A Wizard page (Page 5097 – **Create Interaction**) is shown in the following screenshot:

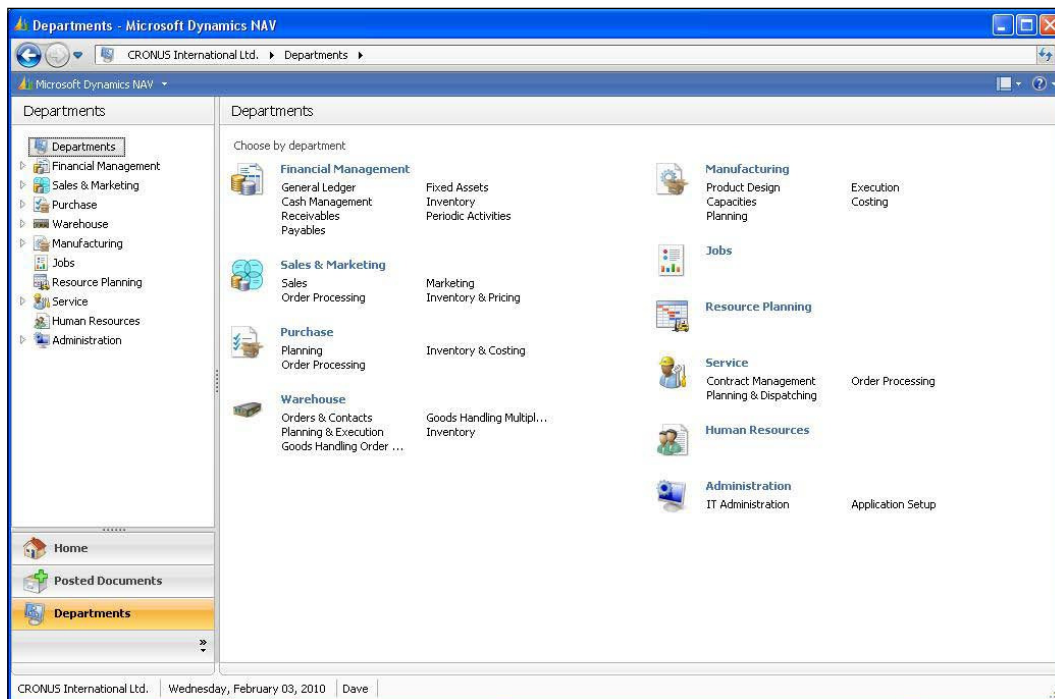


The screenshot shows a Windows-style window titled "Create Interaction - CT200081 Greg Chapman". The window has a blue title bar and a standard Windows interface. Inside the window, there is a "Actions" menu at the top left. Below the menu, a text box explains the wizard's purpose: "This wizard helps you to create interactions and record information regarding their cost, duration and connection to a campaign." The main area contains several input fields: "Who are you interacting with?" with a text box containing "Greg Chapman" and a search icon; "What is the type of interaction?" with a dropdown menu showing "BUS"; "Language Code:" with a dropdown menu; "Who is the salesperson responsible?" with a dropdown menu showing "BD"; and "Describe your interaction.:" with a text box containing "Data collection meeting". At the bottom of the window, there are five buttons: "< Back", "Next >", "Finish", "Contact Search", and "Close".

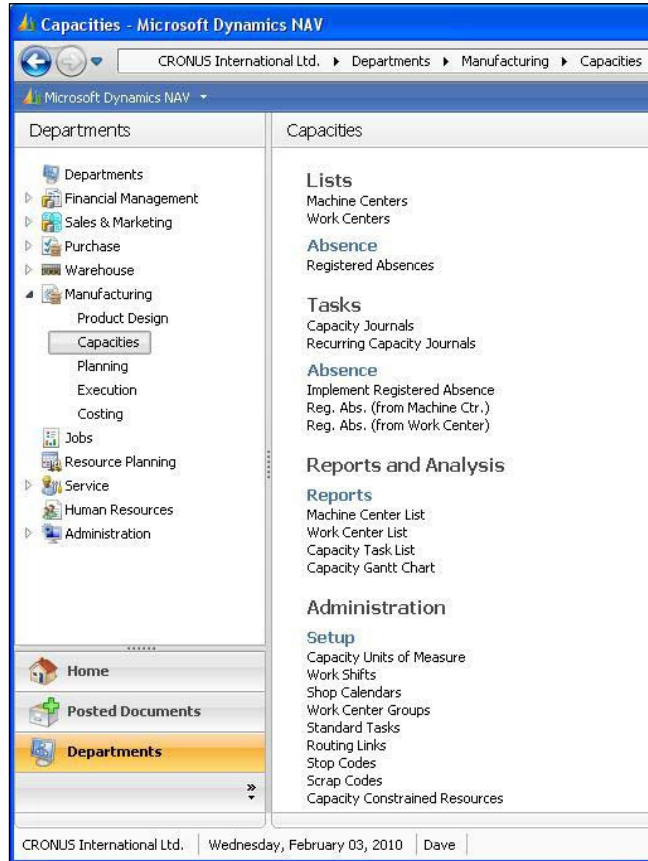
Departments page

The Departments page is a one-of-a-kind, system-generated page. You will never create a Departments page, but you will need to add most new pages that you create to the Departments page. The look and feel of the Departments page cannot be restructured by the developer or personalized by the administrator or user (though individual entries can be added, changed, deleted, or made invisible). But the visible contents will be controlled based on the permissions of the user.

The Departments page is sort of a "site map" to the NAV system for the user. When you add new pages to the Departments page, NAV UX design guidelines encourage the entry of duplicate links within whichever sections the user might consider looking for that page. An example of the **Departments** page is as shown in the following screenshot:



Clicking on **Manufacturing | Capacities** in the **Departments** primary menu results in the following Departments submenu. A brief study of this menu and others like it reinforces the guideline advice encouraging duplicate entries wherever it appears that they might be useful.

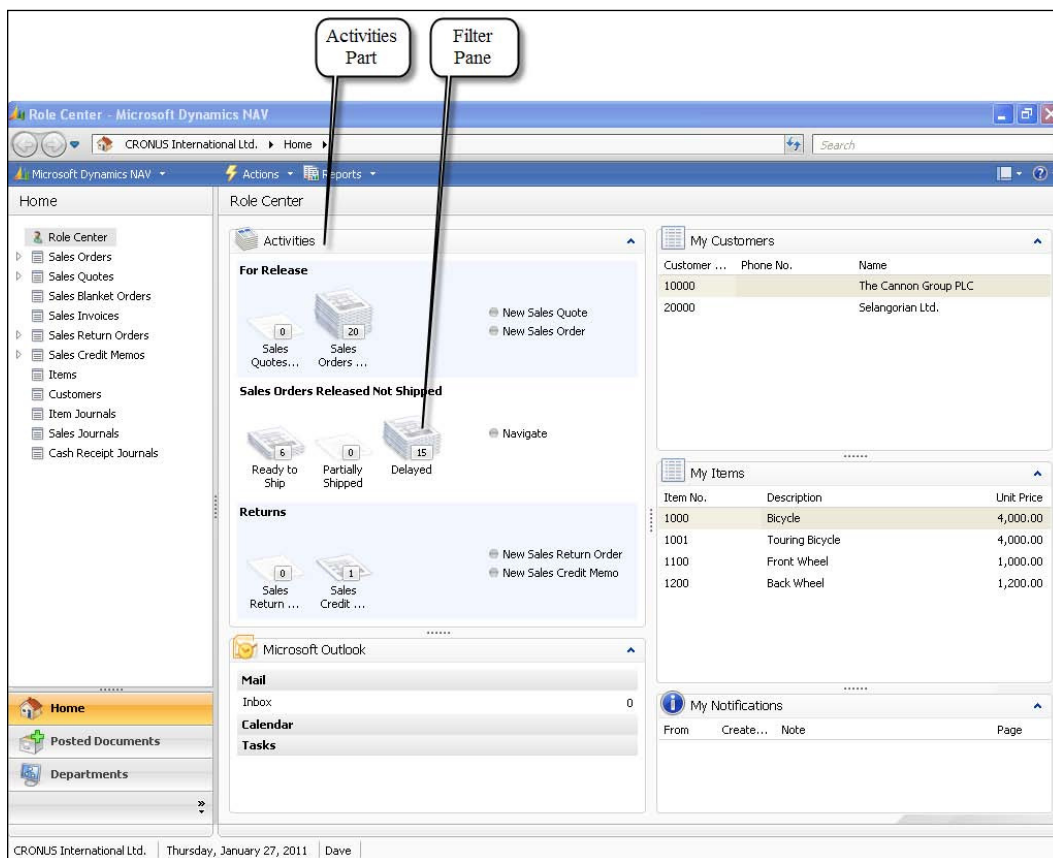


Role Center page

Each user's assigned Role Center page is the location in which they land when first entering NAV 2009. Role Center pages can be modified by the developer or personalized by the administrator, super user, or user. The purpose of a Role Center page is to provide a task-oriented home base which provides only the information that the user typically needs in order to do his/her job. All common tasks should be one or, at most, two clicks away. Twenty-one Role Center pages for roles such as Bookkeeper, Sales Manager, and Production Planner are part of the standard system distribution.

Central to the Role Center page is the **Activities** part. The **Activities** part provides the user a visual overview of their primary tasks. Central to the **Activities** part are the cues. Each cue represents a filtered list of documents in a particular state, thus indicating the amount of work to be done.

The following screenshot shows a Role Center page for an order processor:



Page parts

A number of the pages we have reviewed are made up of multiple panes with each pane including special purpose parts. In order to complete our review of pages, let's look at a number of available component page part building blocks.



Several types of page parts compute the displayed data on the fly, taking advantage of flow fields. As a developer, you need to be careful about overuse of such resource-intensive displays. Too much of this good thing may cause performance problems.

FactBoxes

FactBoxes are the components that appear within the FactBox pane. A variety of such components are available as part of the standard product. FactBoxes can be Card Parts, List Parts, Chart Parts, Notes, Outlook, and Record Links among others. Notes, Outlook, and Record Links are System Parts and cannot be modified. All the others can be enhanced from the standard instances, or new ones may be created from scratch.

Card parts and List parts

Card parts are used for FactBoxes that don't require a list. Card parts display fields or perhaps a picture control. An example of the **Customer Statistics** FactBox card part is as shown in the following screenshot:

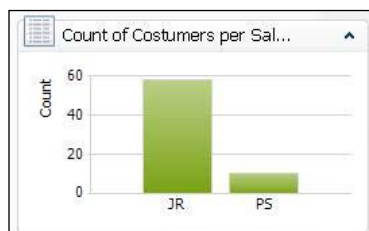
Customer Statistics - ...	
Customer No.:	30000
Balance (LCY):	349,615.40
Outstanding ...	9,502.16
Shipped Not ...	1,996.90
Outstanding ...	10.65
Shipped Not ...	1,996.90
Outstanding ...	0.00
Total (LCY):	361,114.46
Credit Limit (...)	0.00
Overdue Am...	349,615.40
Sales YTD (L...	6,142.90

List parts are used for FactBoxes that require a list. A list is defined as columns of repeated data. The UX Guidelines indicate that a list part should be limited to two columns. Even the standard product doesn't always follow the guidelines. An example of the three-column **My Customers** FactBox list part (Page 9150) is as follows:

My Customers		
Custo...	Phone No.	Name
No.		
10000		The Cannon Group PLC
20000		Selangorian Ltd.
30000		John Haddock Insura...
50000		Guildford Water Depa...

Chart pane

A **Chart** pane displays data from a list in graphic form. The list is generally not displayed. Some **Chart** panes require range parameters, others do not (they default to a defined data range). Most of the charts are two-dimensional, but a sampling of three-dimensional, dynamic charts is distributed with the system. A sample chart pane is shown in the following screenshot, which shows a factbox containing a chartpart:



Page names

Card pages are named in a similar manner as the table with which they are associated, plus the word "Card". Examples include Customer table and Customer Card, Item table and Item Card, Vendor table and Vendor Card, and so on.

The single-record Setup tables that are used for unique setup and control information throughout NAV are named after their functional area plus the word "Setup". The associated Card page should also be (and generally is) named similarly to the table. For example, General Ledger Setup table and General Ledger Setup page, Manufacturing Setup table and Manufacturing Setup page, and so on.

Journal entry (worksheet) pages are given names tied to their purpose, plus the word "Journal". In the standard product, several Journal pages for different purposes may be associated with the same table. For example, the Sales Journal, Cash Receipts Journal, Purchases Journal, and Payments Journal are all associated to the General Journal Line table (that is different pages, but same table).









List pages are named similarly to the table with which they are associated. The List pages which are simple non-editable lists have the word "list" associated with the table name. Examples are Customer List, Item List, and Vendor List. In each of these instances, the table also has an associated card page. Where the table has no associated card page, the list pages are named after the tables, but in the plural format. Examples include Customer Ledger Entry table and Customer Ledger Entries, Item Ledger Entry table and Item Ledger Entries, BOM Ledger Entry table and BOM Ledger Entries, Country table and Countries page, Resource Cost table and Resource Costs page.

If there is a Header and Line table associated with a data category such as Sales Orders, the related page and subpage ideally should be named to maintain the relationship between the tables and the pages. However, in some cases, it is better to tie the page names directly to the function they perform rather than the underlying tables. An example is the two pages making up the display called by the Sales Order menu entry – the Sales Order page is tied to the Sales Header table, and the Sales Order Subform page is tied to the Sales Line table. The same tables are involved for the Sales Invoice page and Sales Invoice Subform page. The occasional lapse of NAV 2009 object terminology into using "form", as in Sales Invoice Subform, rather than always using "page" appears to be the inevitable consequence of supporting the option of having both forms and pages operational.

Sometimes, while naming pages, you will have a conflict between naming based on the associated tables and naming based on the use of the data. For example, the menu entry Contacts invokes a Main page/Subpage named Contact Card and Contact Card Subform. The respective tables are the Contact table and the Contact Profile Answer table. The context usage should take precedence in the page naming.

Accessing the Page Designer

The **Page Designer** is accessed through **Tools | Object Designer | Page**. The Page Designer can be opened either with a new page using the **New** button or on an existing page using the **Design** button (more detail on this process will follow shortly). Once the Page Designer is open, a row of control icons appears at the top of your screen. The following table explains the icons:

	Find
	Toolbox
	Properties (Shift + F4)
	Color
	Font
	Field Menu
	C/AL Symbol Menu (F5)
	C/AL Code (F9)



These are the same icon controls that apply to the Form Designer. However, **Find**, **Toolbox**, **Color**, and **Font** tools do not apply to Page Designer work.

At various points during the creation and maintenance of a page, you can use the other icons (**Properties**, **Field Menu**, **C/AL Symbol Menu**, and **C/AL Code**), their keystroke shortcuts, or in some cases, their right-click menu shortcuts, in order to access these Page Designer functions.

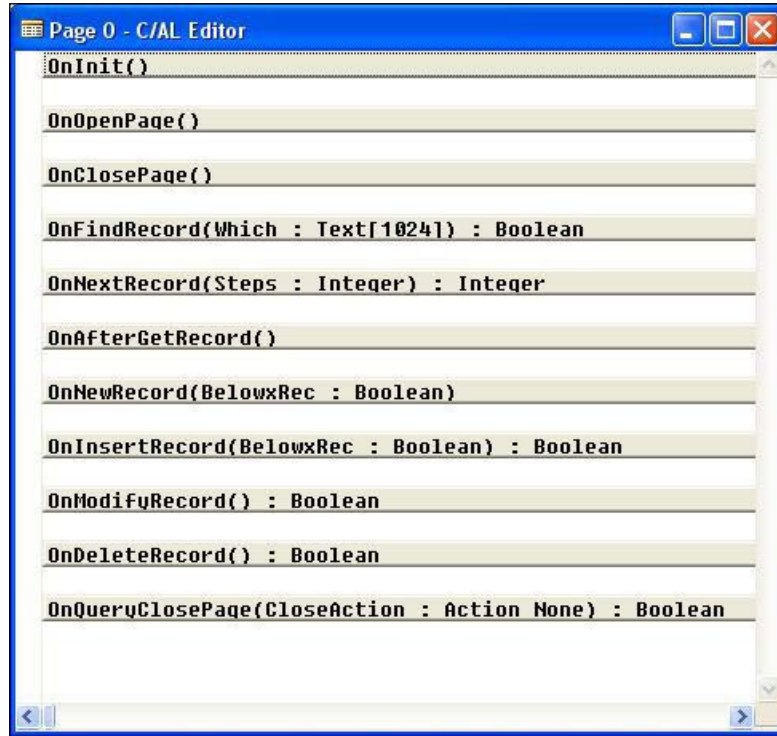
What makes up a page?

All pages are made up of certain common components. How are these components assembled to create the different page types? The basic elements of a page object are the page triggers and properties, plus the controls with their control triggers and properties.

The following screenshot shows the page triggers. We will not spend much time on page triggers because it is generally bad practice to insert any C/AL code in pages. You probably wonder, "Why triggers exist if we shouldn't use them?". The answer is that they exist for use in the cases where page-resident code is absolutely needed to accomplish some display or processing logic. As NAV has changed over the years, the need to use page-resident code has reduced significantly. The only time you should insert code in a page is if:

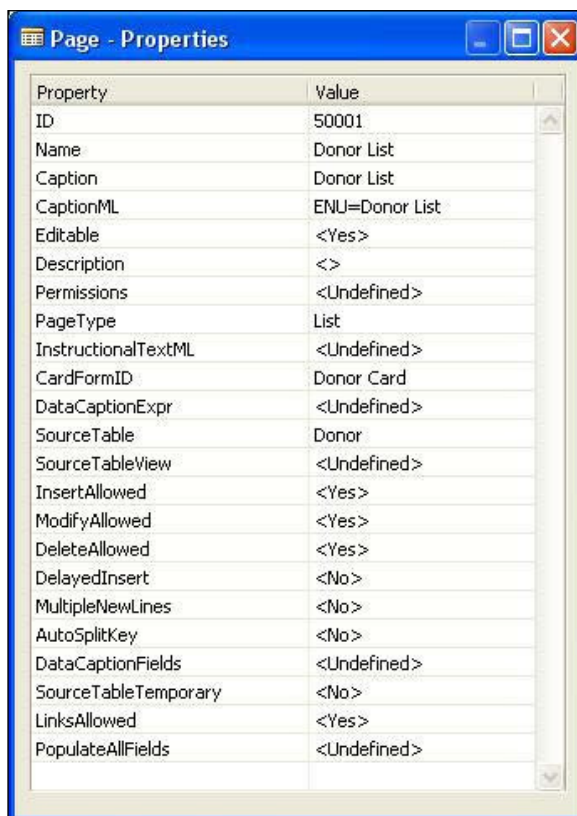
- You can find an example of the same thing in standard code
- There just isn't any other rational way to meet the user interface goal

Even the first example may not be adequate justification because some standard code in pages is there for historical compatibility or upgrade reasons. The correct approach is to put your logic within the tables or, if absolutely necessary, call well-organized functions defined in a Codeunit.



Page properties

The following screenshot shows the **Page - Properties** screen. We will step through the list briefly, but you should actually go to the **Page - Properties** screen and invoke "help" for each field.



In order to illustrate page properties, we will look at the properties of the Donor List page that we created earlier. Many of the properties are still in their default condition. The following are the properties with which we are most likely to be concerned:

- **ID:** The unique object number of the page.
- **Name:** The unique name by which this page is referenced in C/AL code.
- **Caption** and **CaptionML:** The page name to be displayed, depending on the language option in use.
- **Editable:** This determines whether or not the controls in the page can be edited (assuming the table's Editable properties are also set to **Yes**). If this property is set to **Yes**, the control passes on to the **Editable** property on the individual controls.

- **Description:** For internal documentation only.
- **Permissions:** This is used to instruct the system to allow the users of this page to have certain levels of access (r=read, i=insert, m=modify, d=delete) to the TableData in the specified table objects. For example, users of **page 499 – Available – Sales Lines** are allowed to only read or modify the data in the Sales Line table. Anytime you are defining special permissions be careful to test with an end-user license.
- **PageType:** Specifies how this page will be displayed, using one of the available nine page types (Card, List, RoleCenter, CardPart, ListPart, NavigatePage, Document, Worksheet, ListPlus, ConfirmationDialog).
- **CardFormID:** The ID of the Card page that should be launched when the user double-clicks on an entry in this list. This is only used on List pages.
- **SourceTable:** The name of the table to which the page is bound. It must be filled in if this is a bound page.
- **SourceTableView:** This can be utilized automatically and without exception in order to apply certain filters or open the page with a default key other than the Primary Key.
- **DelayedInsert:** This delays the insertion of a new record until the user moves focus away from the new line being entered. If this value is no, then a new record will automatically be inserted into the table as soon as the primary key fields have been completed. This property is generally set to yes when AutoSplitKey (see below) is set to yes. It allows complex new data records to be entered with all of the necessary fields completed.
- **MultipleNewLines:** When set to yes, it allows the insertion of multiple new lines between existing records. However, it defaults to no, which prevents users from inserting new lines between records.
- **AutoSplitKey:** This allows for the automatic assignment of a primary key, provided the last field in the primary key is an integer (there are exceptions to this, but we won't worry about them in this book). This feature enables each new entry to be assigned a key so it will remain sequenced in the table following the record appearing above it. Note that AutoSplitKey and DelayedInsert are generally used jointly.

On a new entry at the end of a list of entries, the trailing integer portion of the primary key, often named `Line No.`, is automatically incremented by 10,000 (the increment value cannot be changed). When an entry is inserted between two previously existing entries, their current key-terminating integer values are summed and divided by two (hence the term `AutoSplitKey`) with the resultant key value being used for the new entry. As 10,000 can only be divided by two and rounded to a non-zero integer result 13 times, only 13 new entries can be inserted between two previously recorded entries by the `AutoSplitKey` function.

- **SourceTableTemporary:** Allows use of a temporary table as the `SourceTable` for the page. This can be very useful in an instance where there is a need to display data based on the structure of a table, but not using the data as it persists in the database. Examples of such an application are Page 634 – Chart of Accounts Overview and Page 6510 – Item Tracking Lines. Note that the temporary instance of the source table is empty when the page opens up, so your code must populate it.

Types of page controls

Next we will discuss some of the different controls that can appear on pages. Controls on pages serve a variety of purposes. Some controls are containers for constants, some for data, some define the organization of other controls. Some controls define actions to be performed.

One of the attributes that makes designing NAV 2009 pages different from previous versions of NAV is that you can define what information is displayed and the sequence order of the elements, but not the detailed screen layout. As a consequence we don't work in a WYSIWYG design environment, but in a Page Designer worksheet environment. A benefit of this is the ability to have the definition work with display targets other than standard workstations, leaving the actual rendering to be target specific.

The following screenshot from the **Page Designer** shows some of the controls on the Customer Card (Page 21). The **Name** is the internal reference name of the object. The **Caption** is what will appear on screen. In this case, all captions are defaulted to what was defined in the table. **Type** and **SubType** define how this control is interpreted by the Role-Tailored Client. Finally, **SourceExpr** defines the source of the data and all table fields.



To a large extent, the structure defined by controls is shown by the indenting of lines. In the preceding screenshot, the primary structure is based on the container control. All of the other controls you can see are indented within the container. The next level of structure is the **General** group control. Indented under this control are a set of data field controls. Following them are the **Communication** group and the **Invoicing** group. The corresponding screen display matching the preceding screenshot is as follows:

Edit - Customer Card - 10000 - The Cannon Group PLC

Actions ▾ Related Information ▾ Reports ▾

Sales Invoice Apply Template Statistics Customer - Balance to Date
Sales Order Cash Receipt Journal
Reminder Sales Journal

New Process Reports

10000 - The Cannon Group PLC

General

No.: 10000
Name: The Cannon Group PLC
Address: 192 Market Square
Address 2:
Post Code: B27 4KT
City: Birmingham
Country/Region Code: GB
Phone No.:
Primary Contact No.:
Contact: Mr. Andy Teal
Search Name: THE CANNON GROUP PLC
Balance (LCY): 168,889.91
Credit Limit (LCY): 0.00
Salesperson Code: P5
Responsibility Center: BIRMINGHAM
Service Zone Code: M
Blocked:
Last Date Modified: 11/5/2008

Communication

Phone No.:
Fax No.:
E-Mail: the.cannon.group.plc@cronu...
Home Page:
IC Partner Code:

Invoicing NATIONAL DOMESTIC
Payments 1M(8D) DOMESTIC 1.5 DOM.
Shipping BLUE Partial EXW DHL
Foreign Trade

Customer Sales Histo...

Customer No.: 10000
Quotes: 0
Blanket Orders: 0
Orders: 3
Invoices: 0
Return Orders: 0
Credit Memos: 0
Pstd. Shipments: 6
Pstd. Invoices: 4
Pstd. Return Re...: 1
Pstd. Credit Me...: 1

Customer Statistics - ...

Customer No.: 10000
Balance (LCY): 168,889.91
Outstanding ...: 1,612.50
Shipped Not ...: 0.00
Outstanding ...: 6.63
Shipped Not ...: 0.00
Outstanding ...: 0.00
Total (LCY): 170,502.41
Credit Limit (...): 0.00
Overdue Am...: 147,811.14
Sales YTD (L...): 17,521.36

Links
Notes

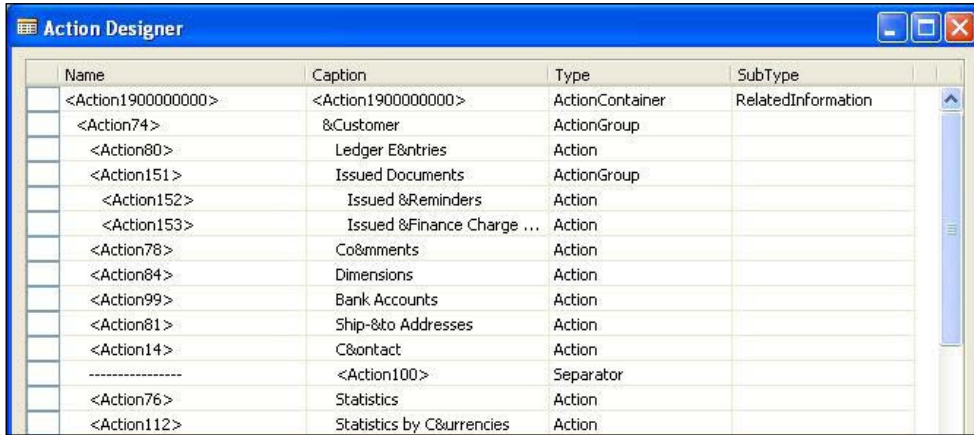
OK

As you can see, the **General** group control defines the General FastTab. The **Communication** group control that followed defines the next FastTab, and so on.

Another group of controls are those that provide action instructions. In NAV 2009, action menus and icons can be found in several locations. First, there is the ubiquitous lightning bolt **Actions** menu and its companion menus in the command bar, as shown in the following screenshot:



The **Action Designer**, where these actions are defined, is accessed from the **Page Designer** form, by focusing on the first blank line below the controls, then clicking on **View** and selecting **Actions**. If you do that for the Customer Card page, you will see a list of actions in the Action Designer, as shown in the following screenshot:

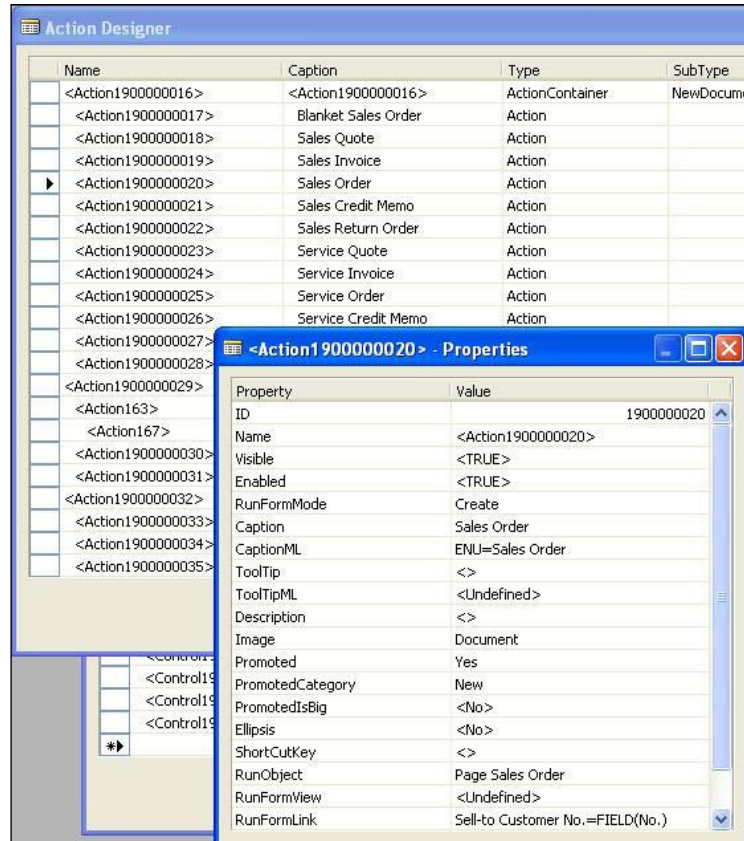


The screenshot shows the 'Action Designer' window with a table of actions. The table has four columns: Name, Caption, Type, and SubType. The actions listed include various system-generated actions and user-defined actions for customer management.

Name	Caption	Type	SubType
<Action1900000000>	<Action1900000000>	ActionContainer	RelatedInformation
<Action74>	&Customer	ActionGroup	
<Action80>	Ledger E&ntries	Action	
<Action151>	Issued Documents	ActionGroup	
<Action152>	Issued &Reminders	Action	
<Action153>	Issued &Finance Charge ...	Action	
<Action78>	Co&mmments	Action	
<Action84>	Dimensions	Action	
<Action99>	Bank Accounts	Action	
<Action81>	Ship-&to Addresses	Action	
<Action14>	C&ontact	Action	
-----	<Action100>	Separator	
<Action76>	Statistics	Action	
<Action112>	Statistics by C&urrencies	Action	

Second, the Action Pane contains access to most frequently used tasks, based on the user profile and sits atop many pages. Following is an Action Pane screenshot:

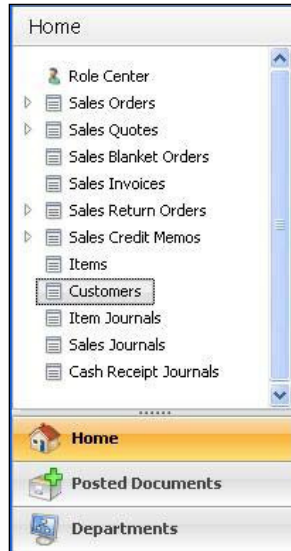




Actions that are already included in the command bar actions list can be promoted to the Action Pane by changing two settings.

First, set the **Promoted** property to **Yes**. Second, set the **PromotedCategory** property to define where the action is to be displayed. See the preceding screenshot for example settings.

Third, is the Navigation Pane on the left side of the RTC display. It is a container of action controls. The Navigation Pane is actually part of the Role Center. The following screenshot displays the action (menu) items of the **Home** activity button for this Role Center. The action entries are defined using the Action Designer within each Role Center page.



The **Home** and **Departments** activity buttons are always present. The **Departments** button provides access to all the entries from the MenuSuite (that is, all the assigned permissions allow).

Fourth and last, the Role Center page is dedicated to providing a home for conveniently located action controls.

It should be obvious that one of the key design criteria for the NAV 2009 RoleTailored Client was to make it easy for a user to have access to the actions they need, in order to get their job done. Our job as developers is to take full advantage of all of these options, in order to make life easier for the user. In general, it's better to go overboard in providing access to useful capabilities, than to be too conservative. "Conservative" means making the user search for the right tool or use several steps in order to get to it. The challenge is to not clutter up the first-level display with too many things, but still have all of the important user tools only one click away.

Inheritance

One of the attributes of an object oriented system is the inheritance of properties. While NAV is more properly described as object based rather than object oriented, the properties that affect data validation are inherited. In addition, properties such as decimal formatting are also inherited. If the property is explicitly defined in the table, it cannot be less restrictively defined elsewhere.

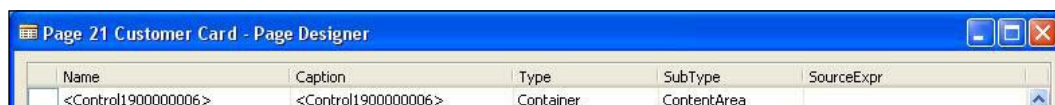
Controls that are bound to a table field will inherit the settings of the properties that are common to both. This basic concept applies on inheritance of data properties – beginning from fields in tables to pages and reports, and then from pages and reports to controls within pages and reports. Inherited property settings that involve data validation cannot be overridden, but all of the others can be changed. This is another instance where it is generally best to define the properties in the table, for consistency and ease of maintenance, rather than defining them for each instance of use in a page or a report.

Page control details

There are five primary types of page controls – **Container**, **Group**, **Field**, **Part**, and **Action**. Container, Group, and Field controls are used in the body, the content area of data displaying pages. Part controls are used to define FactBoxes. Finally, Action controls are used to define menus, icons, and buttons – anything within the RoleTailored Client that allows the user to invoke an action. We'll discuss the content area controls first.

Container controls

Container controls can be of subtypes such as **ContentArea**, **FactBoxArea**, or **RoleCenterArea**. Container controls define the root-level primary structures within a page. All of the page types start with a Container control. The RoleCenterArea Container control can only be used on a RoleCenter pagetype. A page can only have one content area and one Factbox Area.



The screenshot shows a window titled "Page 21 Customer Card - Page Designer". Inside the window is a table with the following data:

Name	Caption	Type	SubType	SourceExpr
<Control1900000006>	<Control1900000006>	Container	ContentArea	

Group controls

Group controls provide the next level of structure within a page. Group controls are the home for fields. Almost every page has at least one Group control.



Several of the Group control properties are particularly significant because of their effect on all the fields within the group.

- **Visible:** It can be TRUE or FALSE, and defaults to TRUE. The Visible property can be assigned a Boolean expression, which can be evaluated during processing. This allows for the possibility of turning on or turning off the visibility of a group of fields during processing, based on some variable condition.
- **Editable:** It can be TRUE or FALSE, and defaults to TRUE. The Editable property can be assigned a Boolean expression which can be evaluated during processing in the same manner as the Visible property.
- **Enabled:** It can be TRUE or FALSE, defaults to empty, which acts as TRUE. The Enabled property can be assigned a Boolean expression, which can be evaluated during processing in the same manner as the Visible and Editable properties.
- **GroupType:** It will be one of the four choices – **Group**, **Repeater**, **Cues**, or **FixedLayout**. The GroupType property is also visible on the Page Designer screen in the column headed **SubType** (see the preceding screenshot).
 - **Group** is used in Card type pages as the structure for fields, which are then displayed in the sequence they hold within the group
 - **Repeater** is used in List type pages as the structure for fields, which are then displayed as repeated rows of those fields
 - **CueGroup** is used for Role Center pages as the structure for the actions that are the primary focus of a user's work day. Cue groups are found in page parts, typically having the word "Activities" in their name. The page parts are included in RoleCenter page definitions. The following screenshot shows a Cue group defined in the Page Designer:

Page 9060 SO Processor Activities - Page Designer				
Name	Caption	Type	SubType	SourceExpr
<Control1900000001>	<Control1900000001>	Container	ContentArea	
<Control1>	For Release	Group	CueGroup	
<Sales Quotes - Open>	<Sales Quotes - Open>	Field		"Sales Quotes - Open"
<Sales Orders - Open>	<Sales Orders - Open>	Field		"Sales Orders - Open"

The above cue group is displayed in the RTC as follows:



- **FixedLayout** is used at the bottom of certain List pages, following a Repeater group. The FixedLayout group typically contains total or additional line-related detail fields. Many of the Journal pages, such as **Page 39 - General Journal**, **Page 40 - Item Journal**, and **Page 201 - Job Journal** have FixedLayout groups. The **Item Journal** FixedLayout group only shows the item description, which is also available in a column, but it could easily display a field that was not otherwise visible. A FixedLayout group can also show a lookup or calculated value like many of the Statistics pages do (for example, **Page 151 - Customer Statistics**, **Page 152 - Vendor Statistics**).

- **IndentationColumnName** and **IndentationControls**: These allow a group to be defined in which fields will be indented, as shown in the following screenshot of the chart of the Accounts page. Examples of pages that utilize the indentation properties include **Page 16 – Chart of Accounts** and **Page 18 – G/L Account List**.

Chart of Accounts ▾				
Sorting: No. ▾		No filters applied		
No.	Name	Income/Bal...	Account Type	
1000	BALANCE SHEET	Balance Sh...	Heading	
1002	ASSETS	Balance Sh...	Begin-Total	
1003	Fixed Assets	Balance Sh...	Begin-Total	
1005	Tangible Fixed Assets	Balance Sh...	Begin-Total	
1100	Land and Buildings	Balance Sh...	Begin-Total	
1110	Land and Buildings	Balance Sh...	Posting	
1120	Increases during the Year	Balance Sh...	Posting	
1130	Decreases during the Year	Balance Sh...	Posting	
1140	Accum. Depreciation, Buildings	Balance Sh...	Posting	
1190	Land and Buildings, Total	Balance Sh...	End-Total	

- **FreezeColumnID**: It freezes the identified column and all of the columns to the left of it, so that they remain in a fixed position while the columns to the right can scroll horizontally. This is like freezing a pane in an Excel worksheet. A similar effect can be applied by a user tailoring their display.
- **ShowAsTree**: It works in concert with the indentation properties. ShowAsTree allows an indentation to be expanded or collapsed dynamically by the user for easier viewing. Examples are **Page 583 - XBRL Taxonomy Lines**, **Page 634 - Chart of Accounts Overview**, and **Page 5522 - Order Planning**.

Field controls

All of the field controls appear in a common format in the Page Designer. The **SubType** column is not used for field controls. The **SourceExpr** column identifies the data field within the associated table.

Page 21 Customer Card - Page Designer					
Name	Caption	Type	SubType	SourceExpr	
<Control1900000006>	<Control1900000006>	Container	ContentArea		
<Control1>	General	Group	Group		
<No.>	<No.>	Field		"No."	
<Name>	<Name>	Field		Name	

All the field control properties are listed for each field, whether or not they apply. It may seem quite obvious, but properties only apply to the data type for which they make sense. For example, the **DecimalPlaces** property only applies to fields where the data type is decimal.

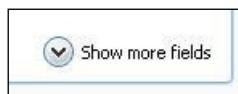
The screenshot shows a window titled "<Name> - Properties" with a list of properties and their corresponding values. The properties are listed in a table with two columns: "Property" and "Value".

Property	Value
ID	1000000003
Name	<Name>
Visible	<TRUE>
Enabled	<TRUE>
Editable	<TRUE>
HideValue	<FALSE>
Caption	<Name>
CaptionML	<ENU=Name>
MultiLine	<No>
ToolTip	<>
ToolTipML	<Undefined>
Description	<>
OptionCaption	<Undefined>
OptionCaptionML	<Undefined>
DecimalPlaces	<Undefined>
Title	<No>
MinValue	<>
MaxValue	<>
NotBlank	<No>
CharAllowed	<Undefined>
ValuesAllowed	<>
BlankNumbers	<DontBlank>
BlankZero	<No>
AutoFormatType	<0>
AutoFormatExpr	<>
SourceExpr	Name
TableRelation	<Undefined>
Importance	<Standard>
CaptionClass	<>
DrillDownFormID	<Undefined>
LookupFormID	<Undefined>
Lookup	<Undefined>
DrillDown	<Undefined>
AssistEdit	<Undefined>
ClosingDates	<No>
Numeric	<No>
DateFormula	<No>
ExtendedDatatype	<None>

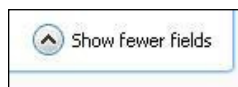
We'll review some of the field control properties which are more frequently used to control some aspect of a particular field, or are more significant in terms of their effect:

- **Visible, Enabled, and Editable:** They have the same functionality as the identically named group controls, but these only apply to individual fields. If the group control is set to **FALSE**, either statically (in the control definition within the page) or dynamically by an expression evaluated during processing, the Group control's **FALSE** condition will take precedence over the equivalent Field control setting. Precedence applies in the same way at the next, higher levels of identically named properties at the Page level, and then at the table level. For example, if a data field is set to **Non-Editable** in the table, that setting will take precedence (overrides) over other settings in a page, control group, or control.
- **HideValue:** It allows the value of a field to be optionally displayed or hidden, based on an expression that evaluates to **TRUE** or **FALSE**.
- **MultiLine:** This should be set to **TRUE** in order to allow the field to display multiple lines of text.
- **OptionCaption** and **OptionCaptionML:** They set the text string options that are displayed to the user. For display purposes, the captions that are set as page field properties will override those defined in the equivalent table field property. The default captions are those defined in the table.
- **DecimalPlaces:** It applies to decimal fields only. If the number of decimal places defined in the page is smaller than that defined in the table, the display is rounded accordingly. If the field definition is the smaller number, it controls the display.
- **Importance:** This controls the display of a field. In the current implementation, this property only applies to Card and CardPart pages. Importance can be set to **Standard** (the default), **Promoted**, or **Additional**:
 - **Standard:** It is the normal display. Implementations of the rendering routines for future targets may utilize this differently.
 - **Promoted:** If the property is set to **Promoted** and the page is on a collapsed FastTab, then the field will be displayed on the FastTab line. If the page is expanded, the field will display normally.

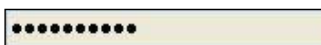
- **Additional:** If the property is set to **Additional** and the FastTab is collapsed, there is no effect on the display. If the FastTab is expanded, then the user can determine whether or not the field is displayed by clicking on the **Show More Fields** or **Show Fewer Fields** display control in the lower-right corner of the page (see the following screenshots).



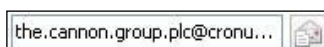
Clicking on the button shown in the preceding screenshot will display the additional field(s), and change the button's caption to that shown in the following screenshot:



- **ExtendedDatatype:** It allows a text field to be categorized as a special data type. The default value is **None**. If **ExtendedDatatype** is selected, it can be any one of the following:
 - **Phone No.**
 - **URL**
 - **E-Mail**
 - **Ratio:** For a processing progress bar display
 - **Masked:** Fills the field as shown in the following screenshot, in order to mask the actual entry. The number of masking characters displayed is independent of the actual field contents. The contents of a masked field cannot be copied.



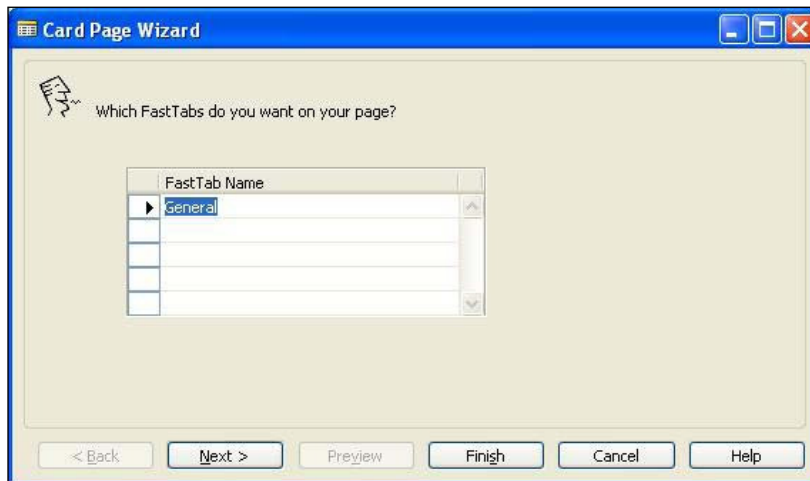
If ExtendedDatatype is Phone No., URL, or E-Mail, an active icon is displayed on the page following the text field providing access to call the phone number, access the URL in a browser, or invoke the email client (see the following screenshot for an example of an email field). Setting ExtendedDatatype will also define the validation that will automatically be applied to the field.



Using page controls in a Card page

The best way to get familiar with properties of the various controls is to work with them. In Chapter 1, *A Short Tour through NAV 2009*, we created basic card and list pages for our Donor table as it existed at that time. Since then, our Donor table has been enhanced considerably. Therefore, we'll begin experimenting by creating a new replacement Donor card.

First, we'll create a card layout that simply contains all the fields in the Donor table. Just as we did in Chapter 1, open the **Object Designer**, click on **Page** and then click on **New**. The Page Wizard's screen will appear. Enter the name (**Donor**) or table number (50000). Choose the **Create a page using a wizard:** option, then choose **Card**. Click on **OK** and you are presented with the **Card Page Wizard** ready for you to lay out a new Card page. As you can see in the following screenshot, the Wizard defaults to a single FastTab of **General**. You can change the name of the FastTab and add new ones, but the Wizard requires you to have at least one FastTab defined.



We're going to design our Donor Card with four FastTabs, modeled after the Customer and Vendor Card pages. Our FastTabs are **General**, **Communications**, **Status**, and **Activity**, as shown in the next screenshot:



Now, click on the **Next** button and you will see the following screen:



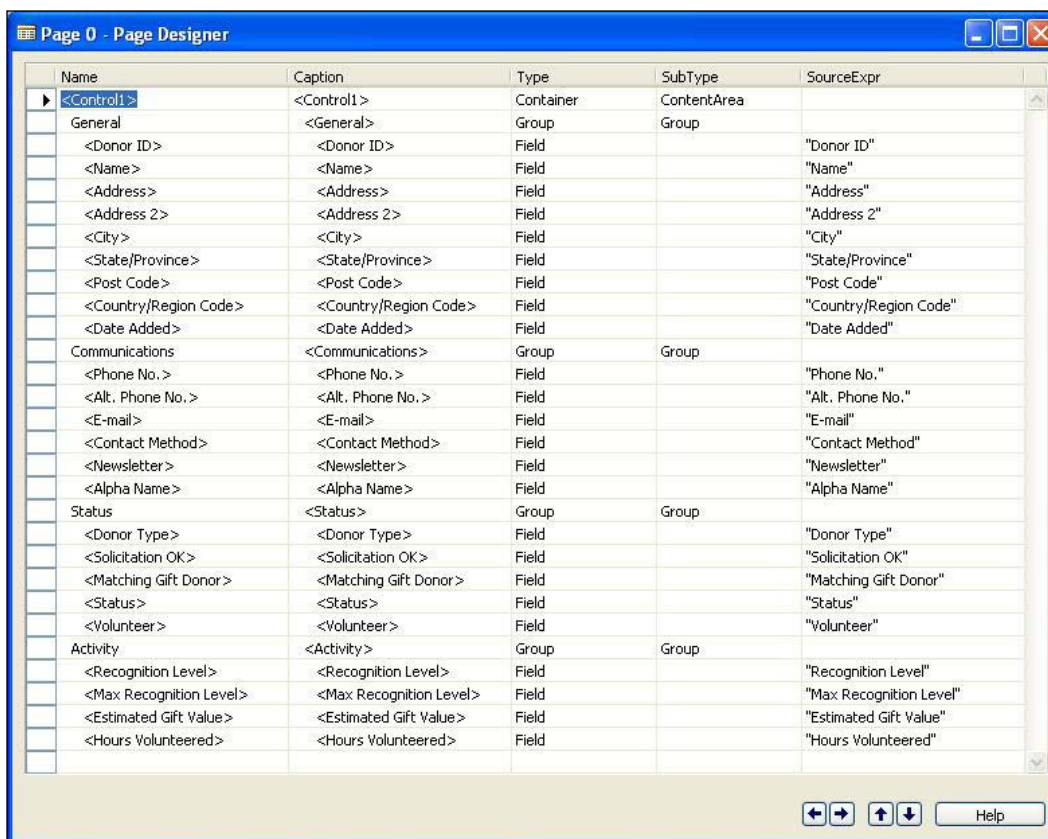
The Donor table fields are listed in the left display, available for selection and placement in the right display panel. There is a separate right side display for each tab of the page we are designing. The next two screenshots show the Wizard display for the **General** and **Activity** tabs after we have selected fields for those.



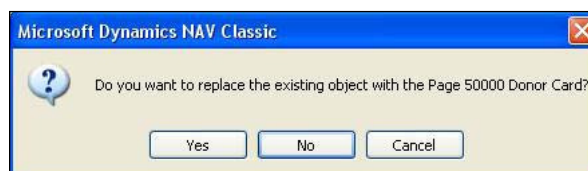
The order in which the fields are listed in the Field Order display is the order in which they will appear in the generated page (split into two equal-sized columns, sequenced top to bottom in the first column, then top to bottom in the second column).



Once we have completed selecting fields and assigning them to tabs, we can click on the **Finish** button and generate the page code. A screen like the one in the next screenshot will be displayed. Note that the use of the Wizard is a one-way path; we cannot return from the code in order to modify it with the Wizard. Once we have generated our page object, any additional changes must be made in the **Page Designer**.



Press the *Esc* key to close the Page Designer screen. A "save changes" form will be displayed. Enter **50000** for the ID number and a name of Donor Card. The Page caption will default to the page name. Leave the compiled checkbox checked, and click on **OK**. It is likely that you will see a screen that resembles the following screenshot:



Click on **Yes**. When the object has been compiled and saved, we have a new Donor Card page, which we should now test. If your system has the original release of NAV 2009 installed, you must test the new page by running it from the system's Run option. From the Start menu, select **Run**. In the command box, enter the following:

Dynamicsnav:////runpage?page=50000

This will invoke the Role-Tailored Client, open it, and call up your new page 50000. If you have NAV 2009 SP1 (Service Pack 1) installed, you can just highlight the page in the **Object Designer** and click on the **Run** button (this is the last time we'll concern ourselves with testing by "Run from the Start button", but please note that using the **Run** button will default to the last company opened in the RTC). In either case, you should see a screen that looks like this:

The screenshot shows a window titled "View - Donor Card - 1001 - Juan O'Hara". The window contains a form with four main sections: General, Communications, Status, and Activity. The General section includes fields for Donor ID (1001), Name (Juan O'Hara), Address (1123 Riveria Way, Suite B-2, Duluth), State/Province (MN), Post Code (55701), Country/Region Code (US), and Date Added (03/21/09). The Communications section includes Phone No. (+1-609-555-1234), Alt. Phone No. (+1-651-222-5555), E-mail (johara@ajuan.com), Contact Method (Email), Newsletter (checked), and Alpha Name (Ohara, Juan). The Status section includes Donor Type (IND/FAM), Status (Active), Solicitation OK (checked), Matching Gift Donor (blank), and Volunteer (checked). The Activity section includes Recognition Level (Friend), Max Recognition Level (Benefactor), Estimated Gift Value (298.00), and Hours Volunteered (17). A Close button is located at the bottom right of the window.

If you haven't entered much test data, many fields may be blank. You may also have to expand one or more of the FastTabs in order to have this appearance. Enter at least a couple of test records before proceeding to the next task.

Now let's change the **Importance** property for a few fields to see the effect it has on our page. Open Page 50000, Donor Card, in the Page Designer by highlighting Page 50000 in the Object Designer and clicking on the **Design** button. Highlight each of the four fields, access the **Field Properties** form, and change the **Importance** property as follows:

- Date Added: Change **Importance** to **Additional**
- Donor Type: Change **Importance** to **Promoted**
- Status: Change **Importance** to **Promoted**
- Recognition Level: Change **Importance** to **Promoted**

Now exit from the page, save, compile, and run the page. View the page with the **General** FastTab expanded, and all the other FastTabs collapsed. You should see a page similar to the following screenshot:

View - Donor Card - 1001 - Juan O'Hara

1001 - Juan O'Hara

General

Donor ID:	1001	City:	Duluth
Name:	Juan O'Hara	State/Province:	MN
Address:	1123 Riveria Way	Post Code:	55701
Address 2:	Suite B-2	Country/Region Code:	US

Show more fields

Communications

Status IND/FAM Active

Activity Friend

Close

You can see the promoted fields on a FastTab when it is collapsed. When the FastTab is expanded, the promoted fields will display in the same manner as the standard **Importance** fields. Note also that the **Date Added** field is not visible in the **General** FastTab—instead, the **Show more fields** option is displayed.

If you click on **Show more fields**, the fields with the **Additional Importance** value will then be displayed, and the option will change to **Show fewer fields** (see the following screenshot). Another significant change that occurs in the display (from the preceding to the following screenshots) is that the column placement of the **City** field changes between the first and second columns. This is due to the fact that many aspects of information placement on the screen are dynamically determined by NAV, not by information provided by the developer.

The screenshot shows a NAV window titled "View - Donor Card - 1001 - Juan O'Hara". The window has a blue title bar and a standard Windows-style interface. Inside, there's a "General" tab with several input fields: "Donor ID" (1001), "Name" (Juan O'Hara), "Address" (1123 Riveria Way), "City" (Duluth), "State/Province" (MN), "Post Code" (55701), "Country/Region Code" (US), and "Date Added" (03/21/09). A "Show fewer fields" button is located at the bottom right of the General tab. Below the General tab, there are sections for "Communications", "Status" (IND/FAM), "Activity" (Active), and "Friend" (Friend). A "Close" button is at the bottom right of the window.

Page Part controls

Page Parts are used for FactBoxes and SubPages. Many of the properties of Page Parts are familiar to other NAV components, and operate essentially the same way in a Page Part as they operate elsewhere. Those "typical" properties include **ID**, **Name**, **Visible**, **Enabled**, **Editable**, **Caption**, **CaptionML**, **ToolTip**, **ToolTipML**, and **Description**.

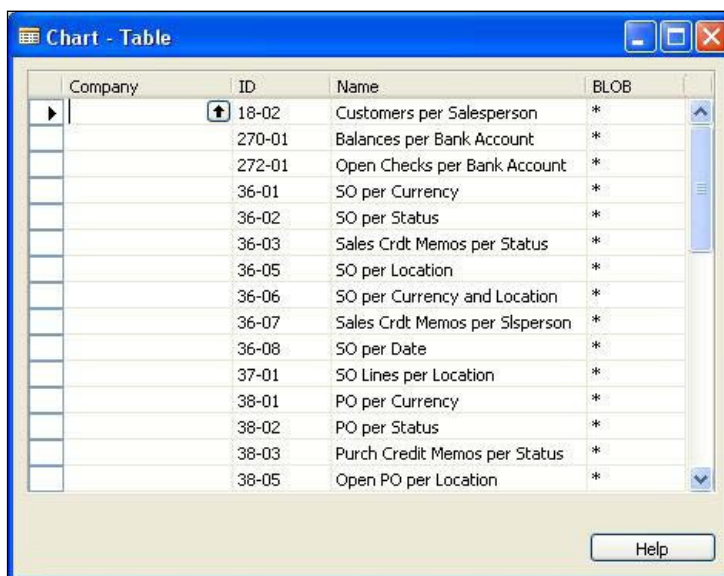
Other properties are specific to Page Part controls:

- **SubFormView**: It defines the table view that applies to the named subpage (see WhseMovLines Part in **Page 7315 – Warehouse Movement**). Note that this property applies to a subpage even though its name is SubFormView.
- **SubFormLink**: It defines the field(s) that links to the subpage and the link (based on a constant, a filter, or another field).
- **ProviderID**: It contains the ID of another Page Part control within the current page. This enables you to link one part to another. For example, **Page 42 – Sales Order** uses this property in order to update the Sales Line FactBox by creating a ProviderID link to the SalesLines FastTab. Other pages with similar links include **Page 41 – Sales Quote** as well as Pages 43, 44, 50, 507, and 5768.

- **PartType**: It defines the type of part to be displayed in a FactBox. There are three options. Each option also requires another related property to be defined:

PartType Option	Required property
Page	PagePartID
System	SystemPartID
Chart	ChartPartID

- **PagePartID**: It must contain the page object ID of a FactBox part, if the **PartTypeOption** is set to **Page**. Standard FactBoxes are in the page object number range of 9080 to 9125 in the initial release of NAV 2009.
- **SystemPartID**: This must contain the name of a predefined system part if the **PartTypeOption** is set to **System**. Available choices are **Outlook**, **Notes**, **MyNotes**, and **RecordLinks**.
- **ChartPartID**: It must contain a chart ID if the **PartTypeOption** is set to **Chart**. The Chart ID is a link to the selected entry in the Chart table (table number 2000000078—see following screenshot).

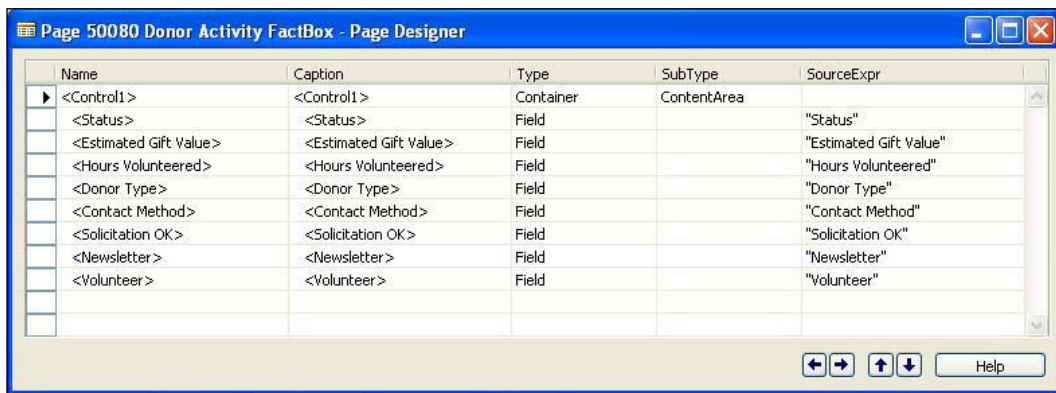


Creating a Card Part FactBox

The creation of a Card Part page to be used as a FactBox is a simple variation of creating a general purpose Card page. Open the **Object Designer**, click on **Page**, and then click on **New**. The Page Wizard's screen will appear. Enter the name (**Donor**) or table number (**50000**). Choose the **Create a page using a wizard:** option, and then choose **Card Part**. Click on **OK**, and you will be presented with the Card Part page Wizard ready for you to lay out a new Card Part.

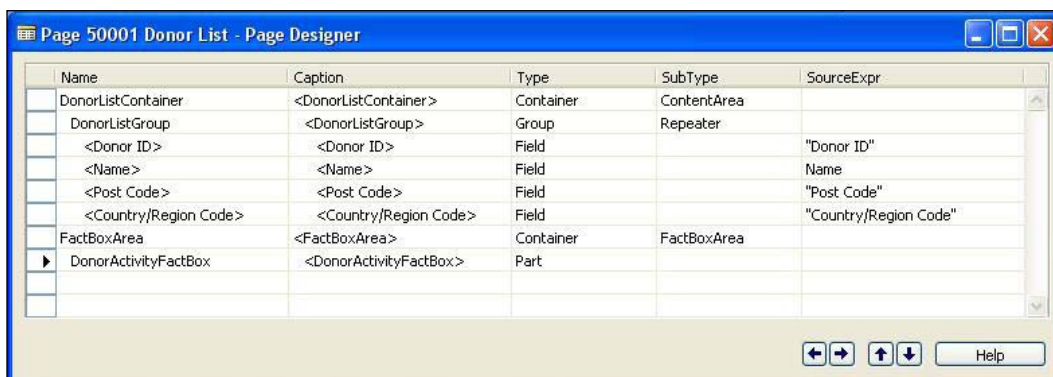
The Wizard will display the same two-panel form we saw when creating the Card page earlier. The list of available fields will be in the left panel with the fields that have been selected shown in the right panel. Select fields from the left column, using the single > key (use the following screenshot as a guide for which fields to select).

Save the new FactBox as **Page 50080 – Donor Activity FactBox**. If you reopen your saved page, you should see the following screenshot:

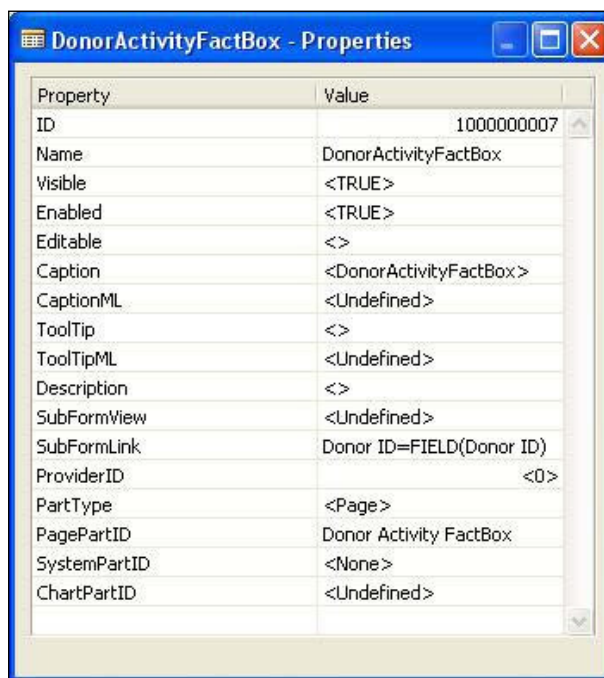


The caption defaults to the page name, in this case Donor Activity FactBox, so let's change the page caption to just Donor Activity. Highlight the new page 50080, and click on the **Design** button. Focus on the first blank line at the bottom of the **Page Designer** screen, and access the **Page Property** screen. Enter **Donor Activity** in the **Caption** property field, exit, save, and compile the page.

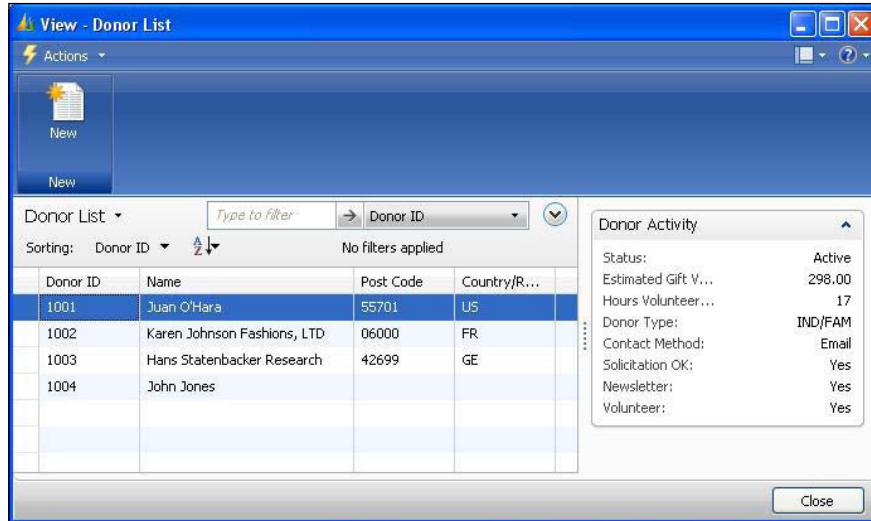
Now, let's modify our original Donor List page in order to take advantage of the FactBox we just created – **Design Page 50001 – Donor List**. Following the field controls that are already in the list page, add a new **Container** control line of subtype **FactBoxArea** using the left arrow key to move the indent left two columns. Below the new Container line, add a Part control. Your Page Designer screen should look like the following screenshot:



Access the **Properties** form for the Part control. The **PartType** property should already have defaulted to **Page**. Set the **PagePartID** to **50080** (or the name you chose for your FactBox). Then, lastly, define the field link between the FactBox page and the List page. In this case, the link will be based on the Donor ID field in each of the two related pages, as shown in the following screenshot:




Save and compile your modified Donor List page 50001. Now test using the **Run** button. Your result should be very similar to the following screenshot:



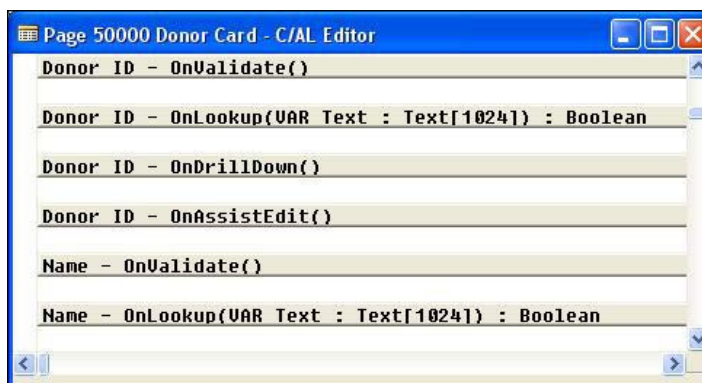
The FactBox area and FactBoxes are added to other types of pages in the same way as what you just did for your List page.

You can have many more FactBoxes tied to a content page than will logically be displayed at one time. In such an instance, you will choose one to three FactBoxes to be initially visible, with all of the others not visible. Users can then change the visibility properties through the personalization tools.

 Since FactBoxes are references to other data, frequently including FlowFields, they usually must be updated when the primary page display contents change. For this reason, too many FactBoxes associated with a page can result in performance degradation.

Page Control triggers

The following screenshot shows Page Control triggers. There are four triggers for each field control. Container, Group, and Part controls do not have associated triggers.



The guideline for the use of these triggers is the same as the one for Page triggers—if there is a choice, don't put C/AL code in a Control trigger. It is always a good policy not to put code in pages, even though NAV doesn't always follow that advice.

There may be occasions where you must put code in a Control trigger but don't choose to do so just because it's the easy way out. Not only will this make your code easier to upgrade in the future, but it will also make it easier to debug and easier for the developer following you to decipher your changes.

Adding more List pages to our ICAN application

Before we move on to less-structured activities, let's create the minimum necessary List pages for our **International Community and Neighbors (ICAN)** application. We originally created List pages for the Donor table and the Donor Type table. In addition, we just enhanced the Donor table List page by adding a FactBox. Now, we will create basic List pages (and one Card page) for the remaining application tables that have been defined so far. In addition to needing these for the ICAN application, we'll see how easy it is to develop basic List and Card pages. The following shows the application tables, the new page object name, and number assignments.

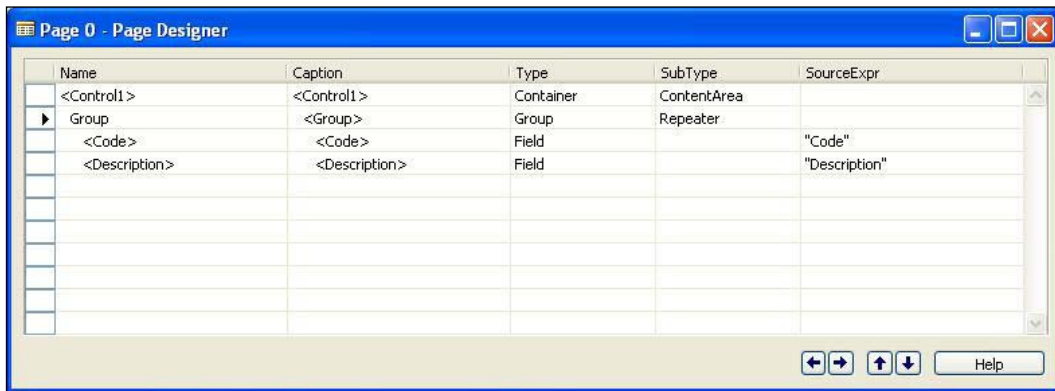
Table no.	Table name	Page no.	Page name
50002	Gift Category	50003	Gift Categories
50004	Client	50004	Clients (List page)
50004	Client	50005	Client Card
50005	ICAN Campaign	50006	ICAN Campaigns
50006	Gift Ledger	50007	Gift Ledger Entries
50007	Aid Ledger	50008	Aid Ledger Entries

Creating a simple list page

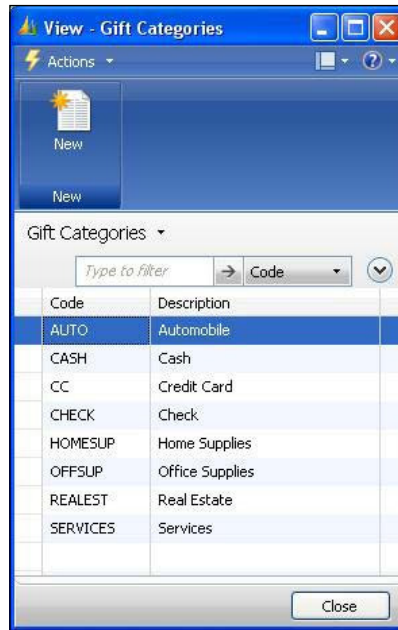
Let's step quickly through the creation of **Page 50003 – Gift Categories**:

1. In order to access the **Page Designer** and create a new blank page bound to the Gift Category table, go to **Tools | Object Designer | Page**, click on the **New** button, enter the table number **50002** (simply because it's easier to type the number than the name, if you know it). Choose the option **Create a page using a wizard**;, and select **List**.
2. Choose both of the available fields. You can choose them individually with the single **>** button, or choose all of the fields at once, by using the **>>** button.
3. Click on the **Finish** button which will display the generated page in the Page Designer (see the following screenshot).
4. Exit the Page Designer, after saving the new page as **50003 – Gift Categories**.
5. Test the new page using the **Object Designer | Run** function.

The page you created in the Page Designer should look like the following, before it is saved and named.



The following screenshot shows what you should see when you run the new page 50003 for a test:



Now, you should try creating the list and card page for the client table on your own. After you have attempted that, read the following to see if you did it the same way.

Creating related List and Card pages

We won't go through the basic detail of creating the list page **50004 - Clients** for the client table, as it can be done in the same way as creating the Gift Categories list page that we have just completed. Some options for creativity (and learning) exist in this task. You can simply select all of the fields, as we did previously. If you do that, you might want to limit the fields that are displayed by default to just the **Client ID**, **Name**, **City**, and **Type** (for example). If you want to limit what fields are displayed, you should set the field control property of **Visible** to **FALSE** for all the other fields. Those fields will still be available to users, who may change the property in their display in order to make them visible. Another option, of course, is just to have only the minimum necessary fields included in the List page. That's reasonable because we will also have a Card page connected with all of the fields on it.

In a simple case, creating the Card page is almost the same as creating the List page, except for any extra FastTabs or special field properties that we might want to set. For our application, we'll keep it simple because there are not enough fields in the client record to justify having separate FastTabs. After we finish creating our Client Card page, there will be one additional step to set a property in the client's list page.

Let's step quickly through the creation of **Page 50005 – Client Card** as follows:

1. In order to access the **Page Designer** and create a new blank page bound to the client table, go to **Tools | Object Designer | Page**, click on the **New** button, and enter the table number as **50004** or the name as **Client**. Choose the option **Create a page using a wizard:** and select **Card**.
2. Keep the default **General** tab assignment and click on the **Next** button.
3. Use the **>>** button to add all the client table fields to the Client card layout.
4. Click on **Finish** to exit the Wizard and enter the Page Designer.
5. Exit the Page Designer, saving your new page as **50005 – Client Card**.
6. Test your new page using the **Object Designer | Run** function.

After you have named and saved your new Card page and run it for a test, it is likely that it will come up blank. Click on the **New** icon and enter some test data. As we have not set up a relationship for the **Type** field, or a default for the **Date Added** field, or any other data entry aids, you'll have to simply key in everything. When you are finished, click on **OK**—then, in the **View** mode, your new Client Card should look something like the following screenshot.

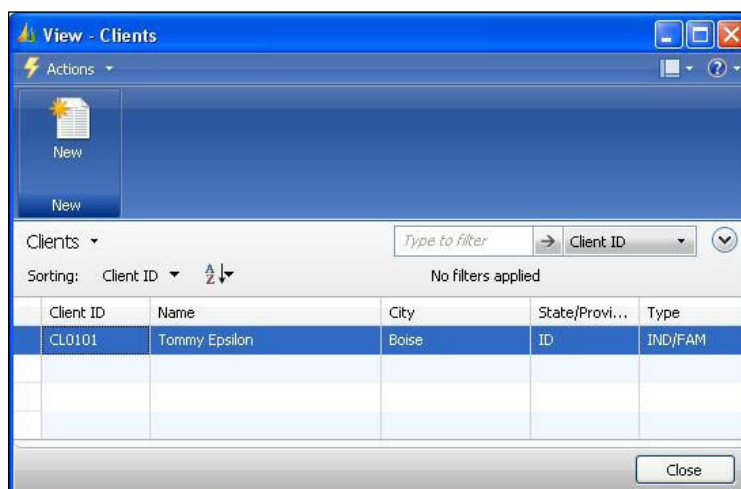
The screenshot shows a window titled "View - Client Card - CL0101". Inside the window, there is a tab labeled "General" with a list of fields and their corresponding values:

Field	Value
Client ID:	CL0101
Name:	Tommy Epsilon
Address:	111 Maple Lane
Address 2:	Apt 3B
City:	Boise
State/Province:	ID
Post Code:	31123
Country/Region Code:	US
Phone No.:	+1 921-555-1234
Type:	IND/FAM
Date Added:	03/22/09

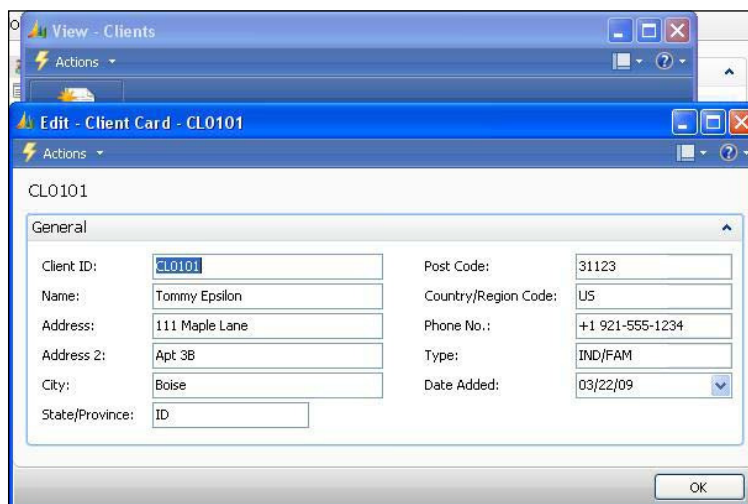
At the bottom right of the window, there is a "Close" button.

The final step is to create a logical connection between the list page and the card page. Highlight page **50004 – Clients** and click on **Design**. Go to the first blank line following the field controls and bring up the **Page Properties** form. Change the property **CardFormID** to **Client Card** (or to **50004**, same result). Save and compile your Clients page.

In order to test the result, run page 50004 – Clients. Your list page should run, showing whatever fields you decided should be present and visible. If you have previously entered any Client test data, it should be visible like the record in the screenshot following.



If you have test data, double click on it and, if you were successful in creating the List page - Card page connection, the record will be displayed in the Client Card as shown in the following screenshot:



If you clicked on the **New** icon instead, the Card page should appear blank, ready for you to enter data for a new client.

With all the experience you have gained by this point, it should be easy for you to proceed and create the remaining three list pages for our application. You should do that now.

Learning more about pages

Descriptions follow of several excellent ways for you to learn more about pages, how they work and how to put them together.

UX (User Experience) Guidelines

The **User Experience (UX)** Guidelines documents developed by Microsoft are available for download from various Internet locations. Search for UX Guidelines for NAV. Created in the form of a standalone interactive set of documents, the UX Guidelines serve both as a summary tutorial to the construction of pages and as recommendations for good design practices. Topics covered include the following list:

- Design principles for great user experiences
- Customization, configuration, and personalization
- The Navigation windows of the Role Center and Departments
- Task windows including Cards, Documents, Journals/Worksheets, and Lists
- Dialog pages: Confirmation, navigate, and request
- How to design most Microsoft Dynamics NAV 2009 User Interface elements
- User interface components
- User interface text design and guidelines
- User interaction via the mouse and keyboard
- Dos and don'ts
- Terminology

This document is a great starting place for anyone wanting good foundation information for understanding and designing NAV 2009 pages.

Creative plagiarism

When you want to create new functionality that you haven't developed recently (or at all), start with a simple test example. Better yet, find another object that has that capability and study it. In many lines of work, the term plagiarism is a nasty term. But when it comes to modifying a system such as NAV, plagiarism is a very effective research and design tool.

There is an old saying: "Plagiarism is the sincerest form of flattery". When designing modifications for NAV, the more appropriate saying might be: "Plagiarism is the quickest route to a solution that works". Especially if you like to learn by exploring (a very good way to learn more about how NAV works), you should allocate some study time for simply exploring the NAV Cronus demo system.

Define what you want to do. Search through the Cronus demonstration system (or an available production system) in order to find one or more pages that have the feature you want to emulate (or a similar one). If there are both complex and simple instances of pages that contain this feature, concentrate your research on the simple one first. Make a test copy of the page and dig into it.

Your best guide will be an existing object that does something which is much like what you want to do. One of your goals should be to identify pages that represent good models to study further. At the extreme, you might plagiarize these (though a better phrase—"use them as models for your development work"). At the minimum, you will learn more about how the expert developers at NAV design their pages.

Experimenting with page controls and control properties

If you have followed along with the exercises so far in this book, it's time for you to do some experimenting on your own. No matter how much information someone else describes, there is no substitute for a personal, hands-on experience. You will combine things in a new way from what was described here. You will either discover a new capability that you would not have learned otherwise, or you will have an interesting problem to solve. Either way, the result will be significantly more knowledge about the tools that NAV C/SIDE provides.

There are at least two good ways to experiment with NAV pages. One is to start with an existing, working page and change it in various ways to see what happens. The other is to start with a blank slate (that is a new, empty page) and experiment in that less cluttered environment. We'll go over a few examples of each here, and then you should go off exploring on your own for a while.

Don't forget to make liberal use of the **Help** information while you are experimenting. Almost all of the available documentation is in the help files that are built into the product. Some of the help material is a bit sparse, but it is being updated on a frequent basis. In fact, if you find something missing or something that you think is incorrect, please use the **Documentation Feedback** function that is built into the NAV help system. The product team responsible for help pay close attention to the feedback they receive and use it to improve the product. Thus, we all benefit.

Help searching

Some help file search functions are not documented. You can search on a partial string by using the asterisk wildcard (that is, walk* will find Walkthroughs). You can search a phrase by enclosing it in double quotes (that is, "page properties" will work). A question mark can be used as a single character wildcard, multiple question marks will equate to the same number of characters as the number of question marks.

Experimentation

Let's start with the blank slate approach, because that allows us to focus on specific features and functions. As we've already gone through the mechanical procedures of creating a new page of the card and list types using the Page Designer to add controls and modify control properties, we won't detail those steps here. But as you move the focus of your experimentation from one feature to another, you may want to review what we covered in that area in this chapter.

Let's walk through some examples of experiments you might do in order to start, and then build on as you get more adventuresome:

1. Create a list page for the client table with three or four fields.
2. Change the **Visible** property of a field, by setting it to **False**.
3. Save and run the page (number it in a range that you know is all test material).
4. Confirm that the page looks as what you expected. Go into **Edit** mode on the page. See if the field is still invisible.
5. Use the page Personalization feature (the Customize icon on the upper-right corner of the page) in order to add the invisible field, perhaps also to remove a field that was originally visible. Exit Personalization. View the page in various modes (such as **View**, **Edit**, **New**).
6. Go back into the **Page Designer** and design the page again.
7. One or two at a time, experiment with setting the **Editable**, **Caption**, **ToolTip**, and other control properties.

8. Don't just focus on text fields. Experiment with integer, decimal, and option fields as well. Create a text field that's 200 characters long. Try out the **MultiLine** property.
9. After you get comfortable with the effect of changing individual properties, you may want to change multiple properties at a time in order to see how they interact.

When you feel that you have thoroughly explored individual field properties in a list, try similar tests in a card page. You will find that some of the properties have one effect in a list, while they may have a different (or no) effect in the context of a card (or vice-versa). Test enough to find out. If you have some "Aha!" experiences, it means that you are really learning.

The next logical step is to begin experimenting with the group level controls. Add one or two to your page, then begin setting the properties for that control, again experimenting with only one or two at a time, in order to understand very specifically what each one does. Do some experimenting to find out which properties at the group level override the properties at the field level, and which do not.

Once you've done group controls, do part controls. Build some FactBoxes using a variety of the different components that are available. Use the System components and some ChartParts as well. There is a wealth of pre-built parts that come with the system. Even if the parts that are supplied aren't exactly right for your application, they can often be used as a model for the construction of custom parts. Remember that using a model can significantly reduce both the design and the debugging work when doing custom development.

After you feel you have a grasp of the different types of controls in the context of cards and lists, maybe it's time to check out some of the other page types. Some of those won't require too much in the way of new concepts. Examples of these are the List+, List Parts, Card Parts, and, to a lesser extent, even the Document pages.

It may be at this point that you decide to learn by studying samples of the page objects that are part of the standard product. One way to start that is by copying an object, such as **Page 22 – Customer List**, and then begin to analyze how it is put together, in order to present the user experience that it presents. Again, you can tweak various controls and control properties in order to see how that affects the page. Remember, you should be working on a copy, not the original!

You can also copy **Page 21 – Customer Card**, connect your Customer List copy to your Customer Card copy and extend your experimenting to the related page pair. Obviously, there is no end to the possibilities of exploring and learning, except for your available time and interest level.

Testing

What we did before you started your experimentation are normal development activities. It's also normal during the development of a new feature to do some experimental testing like you've been doing. A point you must remember is that you should test and retest thoroughly. Make backup copies of your work every time you get a new set of changes working relatively well. While it is exceedingly rare for the NAV database to be corrupted, it is not all that unusual for something you are performing in a test mode to confuse C/SIDE to the point that it crashes. If that happens, any unsaved changes will be lost. Therefore, after each change, or controlled group of changes, you should save your work. In fact, if you are doing complicated things, it is not a bad idea to have a second backup copy of your object that you refresh after every hour or two of development effort.

Your testing should include everyday items as well as the complicated ones, which are absorbing most of your time. Check out your work from the user's point of view, not just the technician's viewpoint. Whenever possible, the C/AL Testability toolset should be used to create and run test scripts. This toolset allows you to create repeatable (that is regression) tests relatively easily which can be used again the next time the software is modified.

Design

Whether you are making modifications to Order Entry or Journal Entry pages, or creating brand new pages, you need to keep the touch typist's needs in mind. Many NAV pages can be designed so that no mouse action is required; that is, everything can be done from the keyboard. If volume data entry is involved, it's good to have that goal in mind.

Use shortcut keys for frequently used functions. Whenever there is a standard shortcut key defined in the out of the box product, use the same shortcut for your modifications (a list of all of the shortcuts is available in the system Help). Group information on pages similar to the way it will be entered from the source material. It is a good idea to spend time with users reviewing the page before locking in a final page design. If necessary, you may want to develop two pages for a particular table, one laid out for ease of use in inquiry purposes, and another laid out for ease of use in volume data entry.

Wherever it is feasible and rational to do so, make your new pages similar in layout and navigation to the pages delivered in the standard NAV product. These generally conform to Windows' design standards and the standards, which are the result of considerable research. As someone has spent a lot of money on that human interface research, it's usually safer to follow the Windows and NAV standards than to follow one's own intuition. Following the standards makes upgrades easier, training cheaper, and requires less support.

The exceptions come, of course, when your client says: "this is the way I want it done". Even then, you may want to work on changing the client's opinion. There is no doubt that training is easier and error rates are lower when the different parts of a system are consistent in their operation. Users are often challenged by the complications of a system with the sophistication of Dynamics NAV. It is not really fair to make their job even more difficult. In fact, your job is to make the user's job easier and more effective.

Summary

At this point, you should be feeling relatively comfortable in the navigation of NAV and with the use of the Object Designer. You should be able to use the Page Wizard Designer in an advanced beginner mode. Hopefully, you have taken full advantage of the various opportunities to create tables and pages, both with our recipes and experimentally on your own.

We have reviewed different types of pages and worked with some of them. We have reviewed all of the controls that can be used in pages and have worked with several of them. We also lightly reviewed page and control triggers. We've acquired a good introduction to the Page Designer and significant insight into the structure of some types of pages.

With the knowledge gained, we have expanded our ICAN application system, creating a number of pages for data maintenance and inquiry, as well as experimented with a number of controls and pages.

In the next chapter, we will learn our way around the NAV Report Designer. We will dig into the various triggers and controls that make up reports. We will also create a number of reports to better understand what makes them tick and what we can do within the constraints of the Report Designer tool.

Review questions

1. Once a Page has been developed and Tailored for implementation, the user has very little flexibility in the layout of the Page. True or False?
2. The Address Bar in the Role Tailored Client simply displays the navigation path leading to the current display. It cannot be used directly to navigate to another display. True or False?
3. Actions appear on the Role Center screen in several places. Choose three:
 - a. Address Bar
 - b. Action Pane
 - c. Filter Pane
 - d. Navigation Pane
 - e. Command Bar
4. The Filter Pane includes "Show results – Where" and "Limit totals to" options. True or False?
5. RTC Navigation Pane entries always invoke which one of the following page types?
 - a. Card
 - b. Document
 - c. List
 - d. Journal/Worksheet
6. All page design and development is done within the C/SIDE Page Designer. True or False?
7. Two Activity Buttons are always present in the Navigation Pane. Which two?
 - a. Posted Documents
 - b. Departments
 - c. Financial Management
 - d. Home

8. Inheritance is the passing of property definition defaults from one level of object to another. If a field property is explicitly defined in a table, it cannot be less restrictively defined for that field displayed on a page. True or False?
9. Which of the following are true about the control property Importance?
 - a. Applies only to Card and CardPart pages
 - b. Cannot affect FastTab displays
 - c. Has three possible values: Standard, Promoted, and Additional
 - d. Applies to Decimal fields only
10. Generally, all of the development work for a Card or List page can be done using the appropriate Wizard. True or False?
11. FactBoxes are delivered as part of the standard product. They cannot be modified nor can new FactBoxes be created. True or False? 12.

5 Reports

Making the simple complicated is commonplace; making the complicated simple, awesomely simple, that's creativity – Charles Mingus

Design is directed toward human beings. To design is to solve human problems by identifying them and executing the best solution – Ivan Chermayeff

Some consider the library of reports, provided as part of the standard NAV product distribution from Microsoft, to be relatively simple in design and limited in its features. Other people feel that the provided reports satisfy most needs because they are simple. Their basic structure is easy to use, and made much more powerful and flexible through the multiplier of NAV's filtering and SIFT capabilities. Some say that the simplicity of the standard product provides more opportunities for creative enhancement.

The fact remains that NAV's standard reports are basic. In order to obtain more complex or more sophisticated reports, we must use the Report Designer features that are part of the product. Through creative use of these features, many different types of complex report logic may be implemented. You can also use NAV reports to feed processed data to other reporting tools such as Excel or "third-party" reporting products.

In this chapter, we will review different types of reports and the components that make up reports. We'll look in detail at the triggers, properties, and controls that are part of NAV reports. We will study the Report Designer tools that are a combination of pure NAV (the C/SIDE Report Designer) and the Visual Studio Report Designer that is tightly integrated into NAV 2009. We'll create some reports with these Report Designer tools. We'll also modify a report or two using the Report Designer. We'll examine the data flow of a standard report and the concept of reports used for processing only (with no printed or displayed output).

What is a report?

A **report** is a vehicle for organizing, processing, and displaying data in a format suitable for outputting. In the past, reports went to hardcopy devices (for example printers). Reporting technology is now more general purpose and flexible. Reports may be displayed on-screen in preview mode rather than being printed, or output to another device (for example, disk storage in PDF format), but with the same formatting as though they were printed. In fact, all of the report screenshots in this book were taken from reports generated in preview mode.

Once generated, the data contents of a report are static. Part of the new flexibility of NAV 2009 is the capability to output reports in preview mode, which have interactive capabilities. However, those capabilities only affect the presentation of the data, not the data included in the report dataset. Examples include dynamic sorting and show or hide data (expand or collapse). Even so, all specification of the data selection criteria for a report must be done at the beginning, before the report is generated. NAV 2009 also allows dynamic functionality for drill down into the underlying data, drill through to a page, and even drill through into another report.

In NAV, report objects can also be classified as processing only by setting the correct report property (that is, by setting the **ProcessingOnly** property to **Yes**). A **ProcessingOnly** report will display no data to the user in the traditional reporting manner, but will simply process and update data in the tables. A report can add, change, or delete data in tables, whether the report is **ProcessingOnly** or a normal printing report.

In general, reports are associated with one or more tables. A report can be created without being externally associated with any table, but that is an exception, not a rule. Even if a report is associated with a particular table, it can freely access and display data from other referenced tables.

Two NAV report designers

NAV 2009 report design uses a pair of Report Designer tools. The first is the Report Designer that is part of the C/SIDE development environment. The second is the Visual Studio Report Designer. For simplicity, we will refer to these as C/SIDE RD and VS RD in this chapter.

The C/SIDE RD is the only tool needed to create reports for the Classic Client. If a NAV 2009 system is using only the Classic Client, then only reports created using the C/SIDE RD can be run. However, when using the RoleTailored Client, both C/SIDE RD and VS RD reports can be run. The RTC runs C/SIDE RD reports by invoking a

temporary instance of the Classic Client, running the report, and then closing down the Classic Client instance (no additional license slots are used). In this book, we will focus totally on the design of reports for the RoleTailored Client using the VS RD.

The typical report development process for an RTC report begins by doing foundation work in the C/SIDE RD. That's where all the data structure, working data elements, data flow, and C/AL logic are defined. The only way to create or modify report objects is to start in the C/SIDE RD. Once all of the elements are in place, the development work proceeds to the VS RD where the layout and presentation work is done, including any desired dynamic options.

The following flow chart provides a conceptual view of the creation of a new report using the two different Report Design approaches – one for the Classic Client and the other for the RoleTailored Client. The functions in the center and left chart paths are those done in the C/SIDE RD (steps 1 through 7). Those in the right set of the chart are the ones done in the VS RD (steps 4 and 6 through 10). Steps 1, 2, 3, and 5 are essentially the same (but not quite) regardless of the target client. Step 4 is done in the C/SIDE RD Sections Designer for both clients, but what you do is quite different in each case.

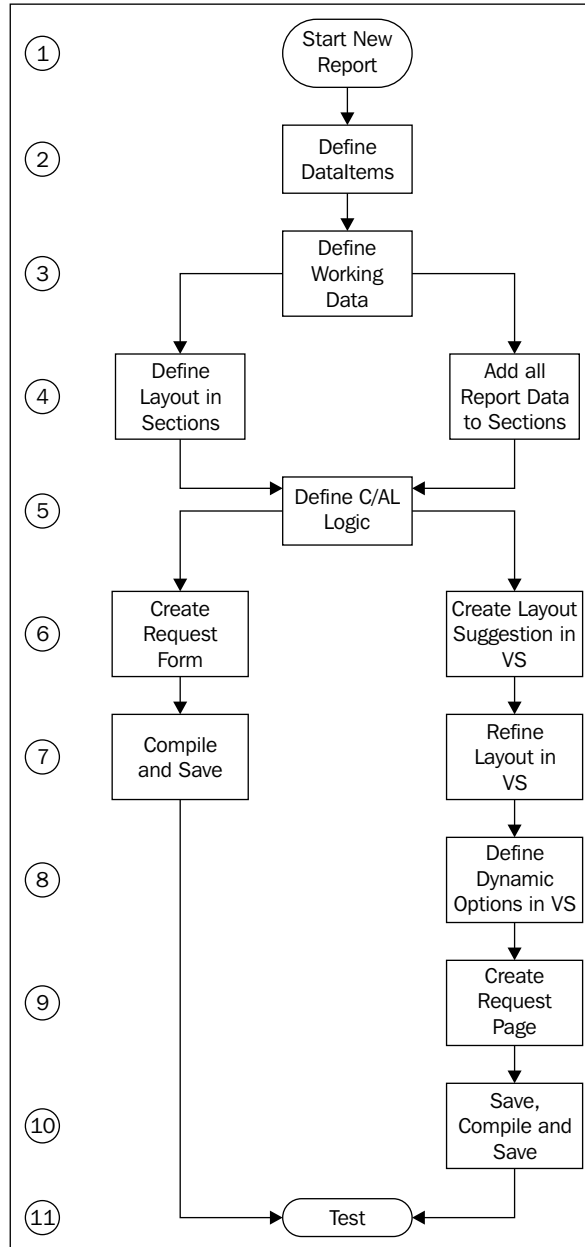
As you can see, many of the functions are the same regardless of the target client. Most of those are done within the Classic Report Designer. Therefore, the accurate claim for NAV 2009 that, even though the layout function uses Visual Studio Report Designer, a large part of the report design is still done within the traditional NAV Report Designer.

For the experienced NAV Classic Client developer who is moving to RTC projects, the biggest challenges will be to learn exactly which tasks are done using which development tool, and to learn the intricacies of the Visual Studio Report Designer layout tools. Those intricacies include understanding just how the VS RD features interact with the NAV data structures and the C/SIDE RD definitions.

This chart shows the general flow of NAV report design in order to make it easier to understand which functions are done in which of the Report Design tools, and allows comparison of the Classic and RoleTailored design processes. In practice, the actual flow will depend on the specifics of a particular report.

It's feasible for a simple C/SIDE report to be entirely generated by the Wizard, but that is generally not true for a VS RD report. It's important to note that some of the steps defined in the chart can be performed in a different sequence than that is shown, and some can be repeated in an iterative fashion. Nevertheless, the chart that follows is a good introductory guide to NAV Report Design.

Terminology for the following chart: **Working Data** is all the non-database data needed to process the report; **Report Data** is what will be displayed in the report.



A hybrid report designer

The Report Designer toolset in NAV 2009 represents a set of compromises tied back to some initial NAV 2009 product feature goals. One product feature goal was to retain the ability of developers of developers to do their to do their work within C/SIDE, thus avoiding scrapping more than a decade of knowledge and experience. A second product feature goal was to provide a much more fully featured set of reporting capabilities.

After much thought and experimentation, the decision was made to create a toolset that would target report generation using the functionality of **SQL Server Reporting Services (SSRS)**. The method of accomplishing that was to "glue together" the data and logic definition parts of the C/SIDE Report Designer to the layout parts of Visual Studio Report Designer, in order to create a hybrid.

When a report is designed, VS RD builds a definition of the report layout in the XML-structured **Report Definition Language Client-side (RDLC)**. When you exit VS RD, the latest copy of the RDLC code is stored in the current C/SIDE Report object. When you exit the Report Designer and save your Report object, the C/SIDE RD saves the combined set of report definition information, C/SIDE and RDLC, in the database. If you export a report object in text format, you will be able to see the two separate sets of report definition. The XML-structured RDLC is quite obvious (beginning with the heading RDLDATA). We will discuss exporting objects in more detail in Chapter 8, *Advanced NAV Development Tools*.

NAV report—look and feel

NAV allows you to create reports of many different kinds with vastly different "look and feel" attributes. The consistency of report look and feel does not have the same level of design importance as the consistency of look and feel for pages does. The standard NAV application only uses a few of the possible report styles, most of which are in a relatively "plain-Jane" format.

While good design practice dictates that enhancements should integrate seamlessly unless there is an overwhelming justification for being different, there are many opportunities for providing replacement or additional reporting capabilities. The tools that are available within NAV for accessing and manipulating data in textual format are very powerful. Unlike the previous versions of NAV, this new version includes a reasonable set of graphical reporting capabilities. And, of course, there is always the option to output report results to another processing/presentation tool such as Excel.

NAV report types

The following are the types of reports:

- **List:** This is a formatted list of data. A sample list report in the standard system is the **Customer - Order Detail** list shown in the following screenshot:

Customer - Order Detail

4/1/2010 8:49 PM

Page 1

ISAAC\Dave

Shipment Date	Type No.	Description	Quantity	Outstanding Quantity	Quantity on Back Order	Unit Price Excl. VAT	Line Discount Amount	Inv. Discount Amount	Outstanding Orders
01454545		New Concepts Furniture							
	Order No.	101018 1/29/2010							
01/29/10	Item 1980-S	MOSCOW Swivel Chair, red	6	6	6	221.146	66.34	0.00	1,260.54 US
		New Concepts Furniture							1,260.54 US
10000		The Cannon Group PLC							
	Order No.	2001 1/20/2010							
01/20/10	Item LS-MAN-10	Manual for Loudspeakers	4	4	4	0.00	0.00	0.00	0.00
	Order No.	2006 1/20/2010							
01/20/10	Item LS-MAN-10	Manual for Loudspeakers	10	10	10	0.00	0.00	0.00	0.00
	Order No.	2011 1/20/2010							
01/20/10	Item LS-150	Loudspeaker, Cherry, 150W	10	10	10	129.00	0.00	0.00	1,290.00
		The Cannon Group PLC							1,290.00
20000		Selangorian Ltd.							
	Order No.	101017 1/29/2010							
01/29/10	Item 1928-W	ST.MORITZ Storage Unit/Dra	2	2	2	342.10	68.42	0.00	615.78
01/29/10	Item 1964-W	INNSBRUCK Storage Unit/G.	1	1	1	292.00	29.20	0.00	262.80
01/29/10	Item 1976-W	INNSBRUCK Storage Unit/W.	1	1	1	256.10	25.61	0.00	230.49
	Order No.	2002 1/20/2010							
01/20/10	Item LS-75	Loudspeaker, Cherry, 75W	10	10	10	79.00	0.00	0.00	790.00
01/20/10	Item LS-120	Loudspeaker, Black, 120W	6	6	6	88.00	0.00	0.00	528.00
01/20/10	Item LS-10PC	Loudspeakers, White for PC	20	20	20	69.00	0.00	0.00	1,180.00
	Order No.	2007 1/20/2010							
01/20/10	Item LS-2	Cables for Loudspeakers	20	20	20	21.00	0.00	0.00	420.00
01/20/10	Item LS-S15	Stand for Loudspeakers LS-1	12	12	12	79.00	0.00	0.00	948.00
01/20/10	Item LS-MAN-10	Manual for Loudspeakers	30	30	30	0.00	0.00	0.00	0.00

- **Document:** This is formatted along the lines of a pre-printed form, where a page (or several pages) represents a complete, self-contained report. Examples are Customer Invoice, Packing List (even though it's called a list, it's a document report), Purchase Order, and Accounts Payable check.

The following screenshot is a Customer **Sales-Invoice** document report:

Sales - Invoice

6 of 21 100% Find | Next

Sales - Invoice
Page 1

Designstudio Gmunden
Fr. Birgitte Vestphael
Seepromenade 1b
AT-4810 Gmunden
Austria

CRONUS International Ltd.
5 The Ring
Westminster
W2 8HG London

Phone No. 0666-354-9282
Fax No. 0666-354-9283
VAT Reg. No GB716487762
Giro No. 888-9999
Bank World Wide Bank
Account No. 99-99-888

Bill-to Customer No. 43687129
VAT Registration No ATU89759098
Salesperson John Roberts

January 15, 2010
Invoice No. 103007
Order No. 101011
Posting Date 01/16/10
Due Date 02/15/10
Prices Including VAT No

No.	Description	Posted Shipment Date	Quantity	Unit of Measure	Unit Price	Disc. %	VAT Identifier	Amount
1920-S	ANTWERP Conference Tabl	01/15/10	2	Piece	609.452		VAT25	1,218.90
1900-S	PARIS Guest Chair, black	01/15/10	6	Piece	181.357		VAT25	1,088.14
1996-S	ATLANTA Whiteboard, base	01/15/10	1	Piece	1,314.439		VAT25	1,314.44
Total EUR								3,621.48

Payment Terms 1 Month/2% 8 days
Shipment Method Ex Warehouse

Reports

The List and Document report types are defined based on their layout. The next three report types are defined based on their usage rather than their layout.

- **Transaction:** These reports provide a list of ledger entries for a particular Master table. For example, a Transaction list of Item Ledger entries for all of the items matching a particular criteria, or a list of General Ledger entries for some specific accounts, as shown in the following screenshot:

G/L Register

CRONUS International Ltd.

Thursday, April 01, 2010

Page 4

ISAAC\Dave

Posting Date	Doc ume nt T	Document No.	G/L Account No.	Name	Description	VAT Amount	G e n n	Gen. Bus. Posti	Ge n. Pro	Amount	Entry No.
01/01/09		2009-1	9310	Unrealized FX Gain	Entries, January 20	0.00				119.60	192
01/01/09		2009-1	5310	Revolving Credit	Entries, January 20	0.00				-196,261.99	193
01/01/09		2009-1	2310	Customers Domestic	Entries, January 20	0.00				-912,284.73	194
01/01/09		2009-1	2320	Customers, Foreign	Entries, January 20	0.00				-130,371.57	195
01/01/09		2009-1	5410	Vendors, Domestic	Entries, January 20	0.00				576,265.99	196
01/01/09		2009-1	5420	Vendors, Foreign	Entries, January 20	0.00				95,162.79	197
01/01/09		2009-1	5310	Revolving Credit	Entries, January 20	0.00				-445,121.95	198
01/01/09		2009-1	2940	Giro Account	Entries, January 20	0.00				445,121.95	199
01/01/09		2009-1	5310	Revolving Credit	Entries, January 20	0.00				371,227.52	200
Register No. 5											
01/25/09	Invoi	108001	8640	Miscellaneous	Invoice 108001	5,000.00	P	NATI	MIS	20,000.00	201
01/25/09	Invoi	108001	5630	Purchase VAT 25	Invoice 108001	0.00				5,000.00	202
01/25/09	Invoi	108001	5410	Vendors, Domestic	Invoice 108001	0.00				-25,000.00	203
01/25/09	Pay	108001	2940	Giro Account	Invoice 108001	0.00				-25,000.00	204
01/25/09	Pay	108001	5410	Vendors, Domestic	Invoice 108001	0.00				25,000.00	205
Register No. 6											
01/31/09		D2009010	1340	Accum. Depreciatio	Depreciation Janua	0.00				-500.00	206
01/31/09		D2009010	8830	Depreciation, Vehicl	Depreciation Janua	0.00				500.00	207
01/31/09		D2009010	1240	Accum. Depr., Oper	Depreciation Janua	0.00				-55.00	208
01/31/09		D2009010	8820	Depreciation, Equip	Depreciation Janua	0.00				55.00	209
Register No. 7											
02/01/09	Invoi	108011	1220	Increases during th	Order 106019	1,128.00	P	NATI	MIS	4,512.00	210

- **Test:** These reports are printed from Journal tables prior to posting the transactions. Test reports are used to pre-validate data before posting.

The following screenshot is a Test report for a **General Journal** batch:

General Journal - Test

CRONUS International Ltd.

4/1/2010 9:06 PM
Page 1
ISAAC\Dave

Journal Template Name: CASHRCPT
Journal Batch: BANK

Gen. Journal Line: Journal Template Name: CASHRCPT, Journal Batch Name: BANK

Posting Date	Document No.	Type	Account No.	Name	Description	Ge. n. Posting Type	Ge. n. Bus. Posting Group	Ge. n. Prod. Posting Group	Amount	Bal. Account No.	Balance (LCY)
02/03/10	P G02001	G/L	2330	Accrued Intere	Accrued Interest				157.23	WWVB-0	0.00
02/03/10	P G02002	G/L	2340	Other Receiva	Other Receivable				1,750.50	WWVB-0	0.00
02/03/10	P G02003	G/L	2330	Accrued Intere	Accrued Interest				11.94	WWVB-0	0.00
02/03/10	P G02004	G/L	2340	Other Receiva	Other Receivable				450.00	WWVB-0	0.00
Total (LCY)									157.23		0.00

Reconciliation

No.	Name	Net Change in Jnl.	Balance after Posting
5310	Revolving Credit	-2,369.67	-1,385,616.10

The following screenshot is for another General Journal batch, containing only one transaction, but with multiple problems, as indicated by the warning messages displayed:

General Journal - Test

CRONUS International Ltd.

Journal Template Name: CASHRCPT
Journal Batch: GENERAL

Gen. Journal Line: Journal Template Name: CASHRCPT, Journal Batch Name: GENERAL

Warning! Payment G02001 is out of balance by -172.45.
Warning! As of 01/28/10, the lines are out of balance by -172.45.
Warning! The total of the lines is out of balance by -172.45.

Posting Date	Document No.	Type	Account No.	Name	Description	Amount	Bal. Account No.	Balance (LCY)
01/28/10	P G02001	Cus	10000	The Cannon G	The Cannon Gro	-250.00 EUR		-172.45
Total (LCY)						-172.45		-172.45

- Posting:** This is a report printed as an audit trail as part of a "Post and Print" process. The printing of these reports is actually controlled by the user's choice of either a **Posting Only** option or a **Post and Print** option. The Post portions of both the options work in a similar manner. The Post and Print option runs a report that is selected in the application setup. The default setup uses the same report that one would use as a transaction history report. This means that such a posting audit trail report, which is often needed by accountants, can be regenerated completely and accurately at any time.

G/L Register										Thursday, April 01, 2010	
CRONUS International Ltd.										Page 1	
										ISAAC/Dave	
G/L Register: No.: 113											
Posting Date	Docu ment Type	Document No.	G/L Account No.	Name	Description	VAT Amount	Gen. Bus. Postl	Gen. Pro d. P	Amount	Entry No.	
Register No. 113											
01/28/10	Paym	G02001	2930	Bank Currencies	The Cannon Group P	0.00			172.45	2763	
01/28/10	Paym	G02001	2310	Customers Domestic	The Cannon Group P	0.00			-172.45	2764	

Report types summarized

The following table contains a list of the basic different NAV report types.

Type	Description
List	Used to list volumes of like data in a tabular format, such as Sales Order Lines, a list of Customers, or a list of General Ledger Entries.
Document	Used in "record-per-page header" + "line item detail" situations, such as a Sales Invoice, a Purchase Order, a Manufacturing Work Order, or a Customer Statement.
Transaction	Generally a list of transactions in a nested list format, such as a list of General Ledger Entries grouped by GL Account, Physical Inventory Journal Entries by grouped Item, or Salesperson To-Do List by Salesperson.
Test	Printed in list format as a prevalidation test and data review, prior to a Journal Posting run. A Test Report option can be found on any Journal page such as General Journal, Item Journal, or the Jobs Journal. Test reports show errors that must be corrected prior to posting.
Posting	Printed in list format as a record of which data transactions were posted into permanent status (that is, moved from a journal to a ledger). A posting report can be archived at the time of original generation or regenerated as an audit trail of posting activity.

There are other standard reports which don't fit within any of the above categories. Of course, many custom reports are also variations on or combinations of standard report structures.

Report naming

Simple reports are often named the same as the table with which they are primarily associated, plus a word or two describing the basic purpose of the report. The report type examples that we've already looked at illustrate this: **General Journal-Test**, **G/L Register**, and **Customer Order-Detail**.

Common key report purpose names include the words Journal, Register, List, Test, and Statistics.

The naming of reports can have a conflict between naming based on the associated tables and naming based on the use of the data. Just as with pages, the usage context should take precedence in naming reports. One requirement for names is that they must be unique; no duplicate names are allowed for a single object type.

Report components overview

What we generally refer to as the report or report object is technically referred to as a Report Description. The **Report Description** is the information describing the layout for the planned output and processing logic to be followed when processing the data. Report Descriptions are stored in the database in the same way as other table or form/page descriptions.

As with pages, we will use the term "report" whether we mean the output, the description, or the object. Reports share some attributes with pages including aspects of the designer, features of various controls, some triggers, and even some of the properties. Where those parallels exist, we should take notice of them. The consistency of any toolset, including NAV, makes it easier to learn and to use. This applies to developers as well as to the users.

The overall structure of an NAV RTC Report consists of the following elements. Any particular report may utilize only a small number of the possible elements (for example, Section Triggers are not used by the RTC), but many different combinations are feasible and logical.

- Report Properties
- Report Triggers
- Data Items
 - Data Item Properties
 - Data Item Triggers
 - Data Item Sections
 - Section Triggers
 - Data Field Controls
- Visual Studio Layout
 - VS Controls
 - VS Control Properties

- Request Page
 - Request Page Properties
 - Request Page Triggers
 - Request Page Controls
 - Request Page Control Properties
 - Request Page Control Triggers

The components of a report description

A report description consists of a number of primary components, each of which in turn is made up of secondary components. The primary components **Report Properties** and **Triggers** and **Data Item Properties** and **Triggers** define the data flow and overall logic for processing the data. These are all designed in the C/SIDE Report Designer.

A subordinate set of primary components, **Data Field Controls** and **Working Storage**, are defined within the **DataItem Sections**, which are also designed in the C/SIDE Report Designer.



Data Fields are defined in this book as the fields contained in the **DataItems** (that is application tables). **Working Storage** (also referred to as Working Data) fields are defined in this book as the data elements that are created within a report (or other object) for use in that object. Working Storage data elements are not permanently stored in the database.

These components constitute the data elements that will be made available to the **Visual Studio Report Designer (VS RD)**.



The VS RD cannot access any data elements that have not been defined within the Report Sections (each of which must be associated with a DataItem).

The **Report Layout** is designed in the VS RD using the data elements made available to the VS RD by the C/SIDE RD, defined in the DataItem Sections. The Report Layout includes the Page Header, Body, and Page Footer. In most cases, the Body of the report is based on a layout table.



Note that the VS RD layout table is a data grid used for layout purposes and is not the same as a table of data stored in the NAV database. This terminology is confusing. When the NAV Help files regarding reports refer to a table, you will have to read very carefully to determine which meaning for "table" is intended.

Within the Report Body, there can be none, one, or more Detail rows. There can also be Header and Footer rows. The Detail rows are the definition of the primary, repeating data display. A report layout may also include one or more Group rows, used to group and total data that is displayed in the Detail row(s).

All of the report formatting is controlled in the Report Layout. The Font, field positioning, visibility options (including expand/collapse sections), dynamic sorting, and graphics are all defined as part of the Report Layout. The same is true for pagination control, headings and footers, some totaling, column-width control, and a number of other display details.

Of course, if the display target changes dramatically in the future versions of NAV, for example, from a desktop workstation display to a browser supporting cellular phone, then the appearance of the Report Layout will change dramatically as well. One of the advantages of SSRS is to support such a level of flexibility. But, if you expect that degree of variability in output devices, you will have design accordingly.

There is another primary functional component of a report description, the Request Page. It displays as a page when a report is invoked. The purpose of the Report Request Page is to allow users to enter information to control the report. Control information entered through a Request Page may include filters, control dates, other control parameters, and specifications as well as defining which available formatting or processing options to use for this instance of the report (that is for this run). The Request Page appears once at the beginning of a report at run time.

Report Data Flow

One of the principal advantages of the NAV report is its built-in data flow structure. At the beginning of any report, you must define the data item(s), the tables that the report will process. It is possible to create a working report that has no data items, but that situation normally calls for a codeunit to be used. There are rare exceptions to this, such as a report created for the purposes of processing only, perhaps to control branching or choice of objects to be run subsequently. In that case, you might have no data item, just a set of logic whose data flow is totally self-controlled. Normally in a report, NAV automatically creates a data flow process for each data item. This automatically created data flow provides specific triggers and processing events:

1. Preceding the data
2. For each record of the data
3. Following the end of the data

The underlying "black-box" report logic (the part we can't see or affect) loops through the named tables, reading and processing one record at a time. That flow is automatic, that is we don't have to program it. Therefore, any time we need a process that steps through a set of data one record at a time, it is quite likely we will use a report object.

If you've ever worked with some of the legacy report writers or the RPG programming language, it is likely that you will recognize this looping behavior. That recognition may allow you to understand how to take advantage of NAV reports more quickly.

The reference to a database table in a report is referred to as a **Data Item**. One of the capabilities of the report data flow structure is the ability to nest data items. If Data Item 2 is nested within Data Item 1 and related to Data Item 1, then for each record in Data Item 1, all of the related records in Data Item 2 will be processed. The next screenshot shows the data item definition screen.

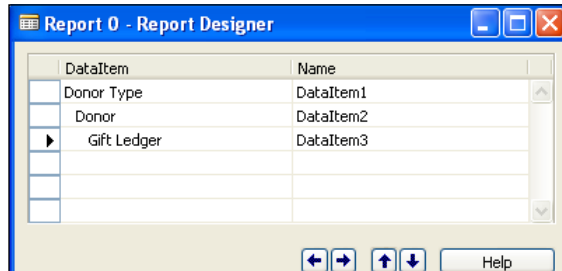
This example uses tables from our ICAN system. The design is for a report to list all the Gifts by Donor for each Donor Type. Thus **Donor Type** is the primary table (**DataItem1**). For each Donor Type, we want to list all the Donors that have given Gifts to ICAN (**DataItem2**). And for each **Donor** of each **Donor Type**, we want to list their Gifts which are recorded in the **Gift Ledger** (**DataItem3**).

On the Data Item screen, we initially enter the table name **Donor**, as you see in the following screenshot. The Data Item Name, to which the C/AL code will refer, is **DataItem1** in our example here. When we enter the second table, **Donor**, then we click on the right arrow at the bottom of the screen. That will cause the selected data item to be indented relative to the data item above (the "superior" data item). That causes the nesting of the processing of the indented data item within the processing of the superior data item.

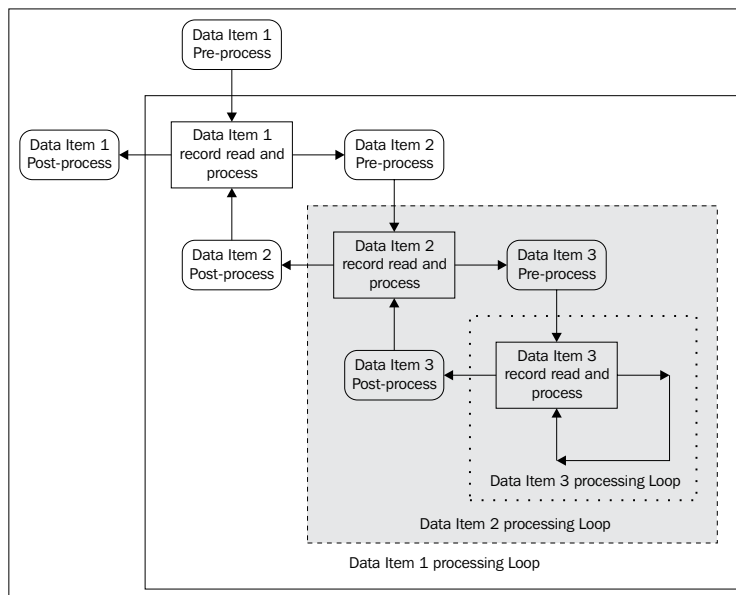
In this instance, we have renamed the Data Items just for the purpose of our example, in order to illustrate data flow within a report. The normal default behavior would be for the **Name** in the right column to default to the table name shown in the left column (for example, the Name for Donor would be displayed as **<Donor>** by default). This default Data Item Name would only need to be changed if the same table appeared twice within the Data Item list. For the second instance of Donor, for example, you could simply give it the Name **Donor2**.

For each record in the superior data item, the indented data item will be fully processed. Which records are actually processed in the indented table will depend on the filters, and the defined relationships between the superior and indented tables. In other words, the visible indentation is only part of the necessary definition. We'll review the rest of it shortly.

For our example, we enter a third table, **Gift Ledger**, and enter our example name of **DataItem3**.



The following chart shows the data flow for this Data Item structure. The chart boxes are intended to show the nesting that results from the indenting of the Data Items in the preceding screenshot. The Donor Data Item is indented under the Donor Type Data Item. That means for every processed Donor Type record, all of the selected Donor records will be processed. That same logic applies to the Donor records and Gift Ledger records (that is, for each Donor record processed, all selected Gift records are processed).



The blocks visually illustrate how the data item nesting controls the data flow. As you can see, the full range of processing for DataItem2 occurs for each DataItem1 record. In turn, the full range of processing for DataItem3 occurs for each DataItem2 record.

In Classic Client reporting, the formatting and output for each record processed happened in sequence as the last step in processing that record. In other words, once a record was read and processed, it was rendered for output presentation before the next record was read.

In the NAV 2009 Role Tailored Client, report processing occurs in two separate steps, the first tied primarily to what has been designed in the Classic RD, the second tied to what has been designed in the VS RD. The processing of data represented in the preceding image occurs in the first step, yielding a complete dataset containing all the data that is to be rendered for output.

This intermediate dataset is a flattened version of the hierarchically structured dataset represented in the Classic RD. Each record in the new dataset contains all the fields from the sections (parent, child, grandchild, and so on) "de-normalized" into a completely flat data structure. This structure also includes Grouping, Filtering, Formatting, MultiLanguage, and other control information required to allow the Visual Studio Report Viewer to properly render the defined report.

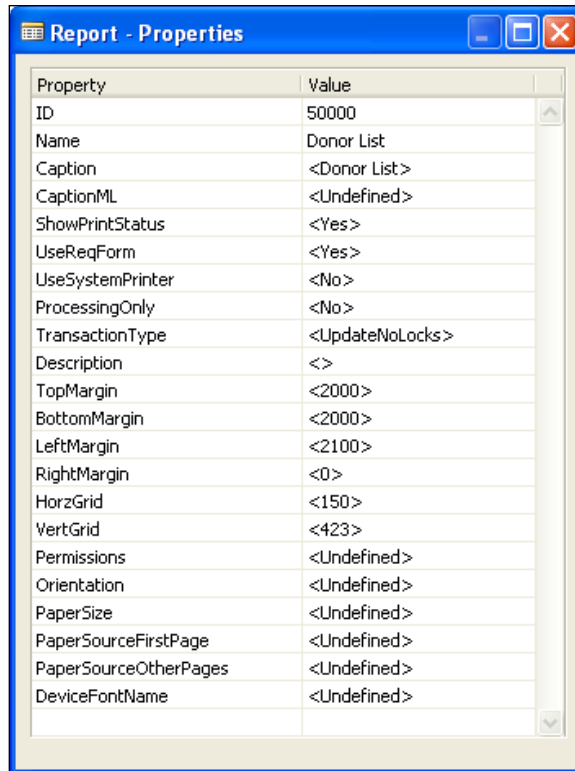
That flattened dataset is then handed off to the Visual Studio Microsoft Report Viewer. The Microsoft Report Viewer provides the new NAV 2009 reporting capabilities such as various types of graphics, interactive sorting and expand/collapse sections, output to PDF and Excel, and other advanced reporting features based on RDLC created by the VS RD design work.

The elements of a report

Earlier we reviewed a list of the elements of a Report object. Now we're going to learn about each of those elements. Our goal here is to understand how the pieces of the report puzzle fit together to form a useful, coherent whole. Following that, we will do some development work for our ICAN system to apply some of what we've reviewed.

Report properties

The Classic RD Report Properties are shown in the following screenshot. Some of these properties have essentially the same purpose as those in pages and other objects. Many of these Report Property settings only apply to Classic reports and are replaced by equivalent Report Properties in the Visual Studio RD.



- **ID:** The unique report object number.
- **Name:** The name by which this report is referred to within C/AL code.
- **Caption:** The name that is displayed for this report; **Caption** defaults to **Name**.
- **CaptionML:** The **Caption** translation for a defined alternative language.
- **ShowPrintStatus:** If this property is set to **Yes** and the **ProcessingOnly** property is set to **No**, then a Report Progress window, including a Cancel button, is displayed. When **ProcessingOnly** is set to **Yes**, if you want a Report Progress Window, you must create your own dialog box.

- **UseReqForm:** Determines if a Request Page should be displayed to allow the user the choice of Sort Sequence and entry of filters and other requested control information.
- **UseSystemPrinter:** Determines if the default printer for the report should be the defined system printer, or if NAV should check for a setup-defined User/Report printer definition.
- **ProcessingOnly:** This should be set to **Yes** when the report object is being used only to process data and no report output is to be generated. If this property is set to **Yes**, then that overrides any other property selections that would apply in a report-generating situation.
- **TransactionType:** This can be in one of four basic options: **Browse**, **Snapshot**, **UpdateNoLocks**, and **Update**. These control the record locking behavior to be applied in this report. The default is **UpdateNoLocks**. This property is generally only used by advanced developers.
- **Description:** This is for internal documentation; it is not used often.
- **TopMargin, BottomMargin, LeftMargin, RightMargin:** Does not apply to an RTC report. There are applicable VS RD properties.
- **HorzGrid, VertGrid:** Does not apply to an RTC report. VS RD layout has its own grid for control positioning.
- **Permissions:** This provides report-specific setting of permissions, which are the rights to access data, subdivided into Read, Insert, Modify, and Delete. This allows the developer to define report and processing permissions that override the user-by-user permissions security setup.

The following printer-specific properties do not apply to an RTC report. Several can be overridden by user selections made at run time.

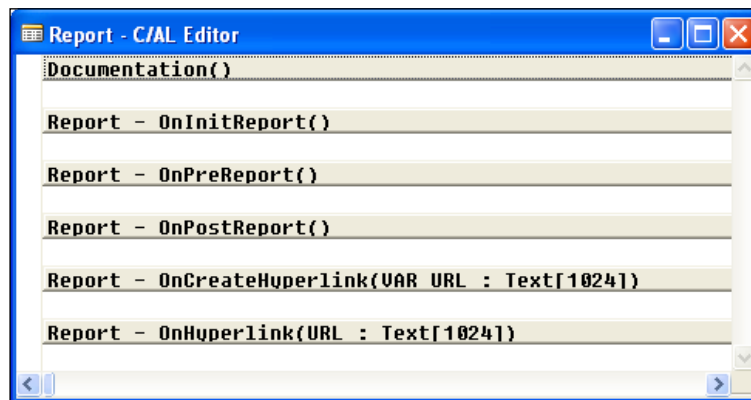
- **Orientation:** There is an applicable VS RD property
- **PaperSize:** There is an applicable VS RD property
- **PaperSourceFirstPage, PaperSourceOtherPages:** This is controlled in Page Setup for RTC reports
- **DeviceFontName:** This is controlled by the Report Viewer

The Visual Studio RD Report Properties are shown in the following screenshot:

Report	
Data	
DataElementName	
DataElementStyle	AttributeNormal
DataSchema	
DataTransform	
ReportParameters	
Design	
DrawGrid	True
GridSpacing	0.125in
SnapToGrid	True
Layout	
InteractiveSize	
Width	8.5in
Height	11in
Margins	
Left	2.1cm
Right	0cm
Top	2cm
Bottom	2cm
PageSize	
Width	20.95cm
Height	29.7cm
Misc	
Author	
AutoRefresh	0
Classes	
Description	
DescriptionLocID	
EmbeddedImages	
Language	=User!Language
References	
Data	

Report triggers

The following screenshot shows the Report triggers available in a report:



- `Documentation()`: Documentation is technically not a trigger, but a section which serves only the purpose of containing whatever documentation you care to put there. No C/AL code is in a Documentation section. You have no format restrictions, other than common sense and your defined practices.
- `OnInitReport()`: It executes once when the report is opened.
- `OnPreReport()`: It executes once after the Request Page completes. All the Data Item processing follows this trigger.
- `OnPostReport()`: If the report is completed normally, this trigger executes once at the end of all of the other report processing. All the Data Item processing precedes this trigger.
- `OnCreateHyperlink()`: It does not apply to an RTC report.
- `OnHyperlink()`: It does not apply to an RTC report.

There are general explanations of Report Triggers in the online C/SIDE Reference Guide (Help); you should also review those explanations.

Data Items

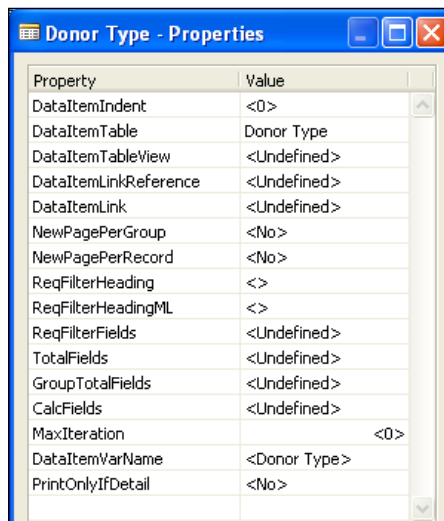
The following screenshot is very similar to the example we looked at when we reviewed Data Item Flow. This time though, we allowed the Name assigned to the Data Items to default. As a result, the names are assigned equal to the DataItems they refer.



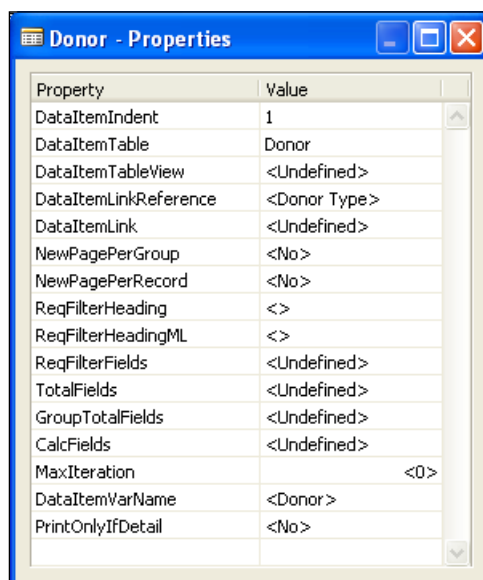
Good reasons for changing the assigned names include making them shorter for ease of coding or making them unique, which is required when the same table is referred to multiple times in a report. For example, suppose you were creating a report that was to list first Open Sales Orders, then Open Sales Invoices, and then Open Sales Credit Memos. As all the three of these data sets are in the same two tables (*Sales Header* and *Sales Line*), you might create a report with Data Item names of *SalesHeader1*, *SalesHeader2*, and *SalesHeader3*, all referencing *Sales Header* Data Items.

Data item properties

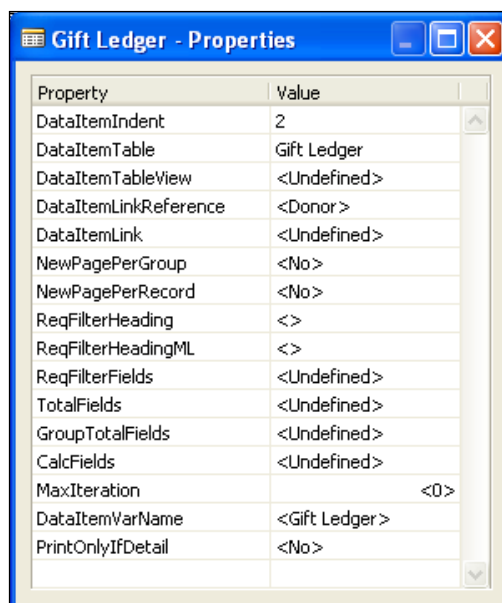
The following screenshots show the properties of the three Data Items in the previous screenshot. The first one shows the **Donor Type – Properties**:



The following screenshot shows **Donor – Properties**:



The screenshot that follows shows the **Gift Ledger – Properties**:



These are the descriptions of each of the properties mentioned:

- **DataItemIndent:** This shows the position of the referenced Data Item in the hierarchical structure of the report. A value of 0 (zero) indicates that this Data Item is at the top of the hierarchy. Any other value indicates the subject Data Item is subordinate to (that is nested within) the preceding Data Item. In other words, a higher valued **DataItemIndent** property is subordinate to any Data Item with a lower valued **DataItemIndent** (for example, a **DataItemIndent** of 1 is subordinate to 0).

If we look at the **DataItemIndent** property listed in each of the three preceding screenshots, we see that Donor Type has a **DataItemIndent=0**, Donor has a **DataItemIndent=1**, and Gift Ledger has a **DataItemIndent=2**. Referring back to the earlier discussion about data flow, we see that the specified Donor table data will be processed through for each record processed in the Donor Type table, and the specified Gift Ledger table data will be processed through for each record processed in the Donor table.

- **DataItemTable:** This is the name of the database table assigned to this Data Item.
- **DataItemTableView:** This is the definition of the fixed limits to be applied to the Data Item (the key, ascending or descending sequence, and what filters to apply to this field). If you don't define a key, then the users can choose the key they want. This allows control of the data sequence to be used during processing.

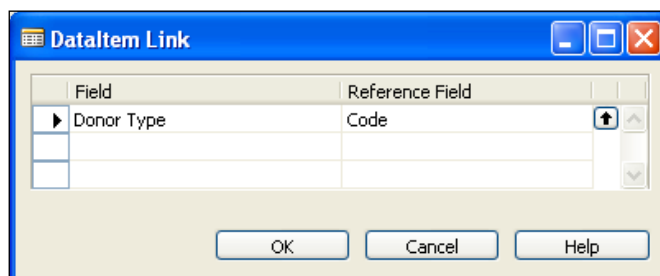


If you define a key in the Data Item properties and, in the **ReqFilterFields** property, you do not specify any Filter Field names to be displayed, this Data Item will not have a tab displayed as part of the Request Page. That will stop the user from filtering this Data Item, unless you provide the capability in C/AL code.

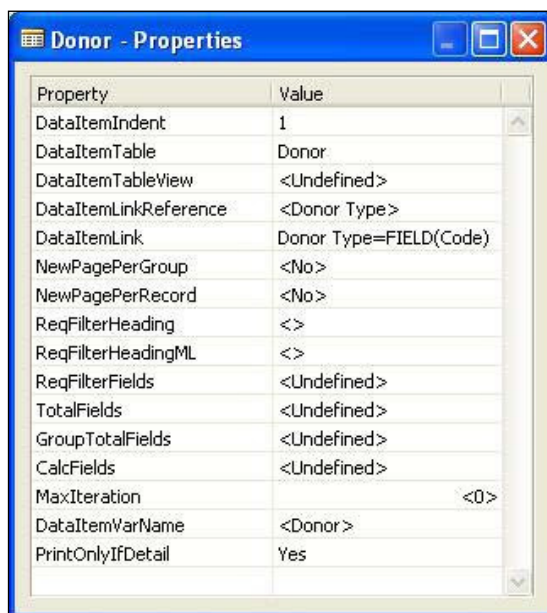
- **DataItemLinkReference:** This names the Data Item in the hierarchy above the Data Item to which this one is linked. The linked Data Item can also be referred to as the parent Data Item. As you can see, this property is **Undefined** for Donor Type, because Donor Type is at the top of the Data Item hierarchy for this report.
- **DataItemLink:** This identifies the field-to-field linkage between this Data Item and its parent Data Item. That linkage acts as a filter because only those records in this table will be processed that have a value that matches with the linked field in the parent data item. In our sample, the Donor Data Item does not have a **DataItemLink** specified.

If this is not changed, no field linkage filter will be applied and all of the records in the Donor table will be processed for each record processed in its parent table that is the Donor Type table. In order to correct this, so that only Donors of the proper type are processed subordinate to each Donor Type, the **DataItemLink** needs to be changed to define the linkage.

- The following screenshot shows the screen where the **DataItem Link** is defined:



The next screenshot shows the **Donor – Properties** screen after the **DataItem Link** has been defined. Defining the **DataItemLink** in this way will cause the Donor table to be filtered by the active Donor Type Code for each loop through the Donor Type table. This will group Donors by Donor Type for report processing.



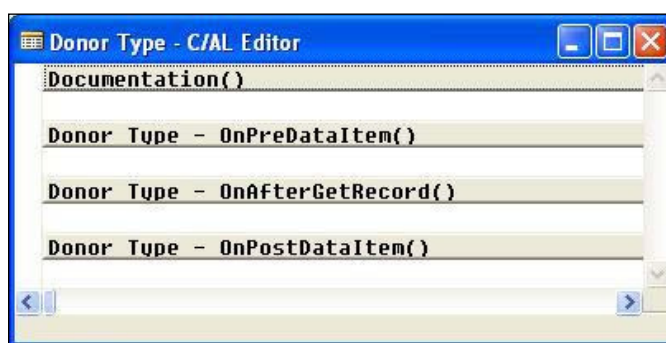
- **NewPagePerGroup, NewPagePerRecord:** Does not apply to an RTC report. There are replacement VS RD properties.
- **ReqFilterHeader, ReqFilterHeadingML:** Does not apply to an RTC report. There are replacement VS RD properties.
- **ReqFilterFields:** This property allows you to choose certain fields to be named in the Report Request Page, to make it easier for the user to access them as filter fields. So long as the Report Request Page is activated for a Data Item, the user can choose any available field in the table for filtering, regardless of what is specified here. Note the earlier comments for the **DataItemTableView** property are relative to this property. A screenshot of a sample Report Request page follows:

- **TotalFields, GroupTotalFields:** Does not apply to an RTC report. Built-in VS RD functions are to be used instead.

- **CalcFields:** This names the FlowFields that are to be calculated for each record processed. As FlowFields do not contain data, they have to be calculated to be used. When a FlowField is displayed on a page, NAV automatically does the calculation. When a FlowField is to be used in a report, you must instigate the calculation. That can either be done here in this property or explicitly within C/AL code.
- **MaxIteration:** This can be used to limit the number of iterations (that is loops) the report will make through this Data Item to a predefined maximum. An example would be to set this to 7 for processing with the virtual Date table to process one week's worth of data.
- **DataItemVarName:** This contains the name shown in the right column of the Data Item screen, the name by which this table is referenced in this report's C/AL code.
- **PrintOnlyIfDetail:** This should only be used if this Data Item has a child Data Item, that is one indented/nested below it. If **PrintOnlyIfDetail** is **Yes**, then sections associated with this Data Item will only print when data is processed for the child Data Item.
- This property is set to **Yes** only for the **Donor** Data Item (see the immediate previous screenshot). That is done so that if there is no **Gift Ledger** for a particular **Donor**, the **Donor** will not print. If we wanted to print only Donor Types that have Donors, we could also set the **PrintOnlyIfDetail** property to **Yes** for the **Donor Type** Data Item.

Data item triggers

Each Data item has the following triggers available:



- `Documentation()` is actually the same instance of this section that showed when we looked at the report triggers. There is only one `Documentation` in any object.

The data item triggers are where the bulk of the flow logic is placed for any report. Additionally, developer defined functions may be freely and voluminously added, but, for the most part, they will be called from within these three triggers.

- `OnPreDataItem()` is the logical place for any preprocessing to take place that couldn't be handled in report or data item properties or in the two report preprocessing triggers.
- `OnAfterGetRecord()` is the data "read/process loop". Code placed here has full access to the data of each record, one record at a time. This trigger is repetitively processed until the logical end of table is reached for this table. This is where you would likely look at data in the related tables. This trigger is represented on our report Data Flow diagram as any one of the boxes labeled data item processing Loop.
- `OnPostDataItem()` executes after all the records in this data item are processed, unless the report is terminated by means of a User **Cancel** or execution of a `C/AL BREAK` or `QUIT` function, or by an error.

Data item Sections

Earlier in our discussion on reports, we referred to the primary components of a report. The Triggers and Properties we have reviewed so far are the data processing components. Next in the report processing sequence are **Sections**. In Classic RD reports, Sections are the output layout and formatting components. In RTC reports, Sections have a much more limited, but still critically important, role.

In the process of creating the initial report design, you may be entering data either completely manually as we've done in our example work, or you may use the Classic Report Wizard. If you use the Wizard, you will end up with Sections defined suitable for Classic Client Report processing.

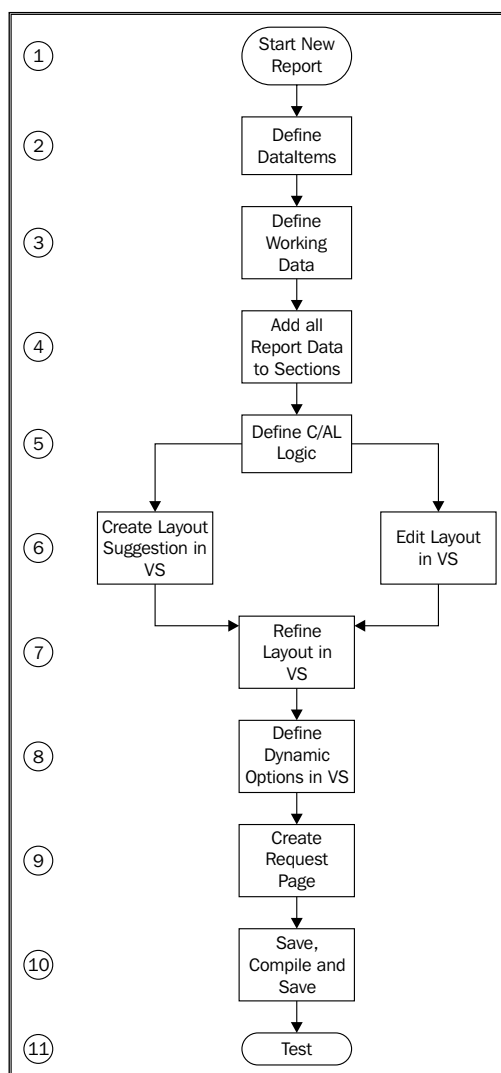
Those Sections may be only rough draft equivalents of what you may want your final report to look like, but they are a suitable starting place for the Classic RD layout work, if that were the tool you were going to use. If you are creating your report completely manually, that is by not using the Wizard, you may also find it appropriate to define Sections to the point that the Classic Client could print a basic, readable report.

In our case, we are focusing our production report development effort on the RoleTailored Client, so we will invest minimal effort on Classic Client compatible report layouts. We might do just enough to allow test report runs for data examination purposes and logic flow debugging. However, creating basic Section layouts provides us with another benefit relative to our VS RD layout work,

especially if we can create them using the Report Wizard, because all the fields to be used by VS RD must be specified in the Sections.

Creating RTC reports via the Classic Report Wizard

Let's look at the RTC report development flow again. The preceding image is very similar to the one we studied earlier in this chapter, but this flowchart only shows the steps that are pertinent to VS RD.

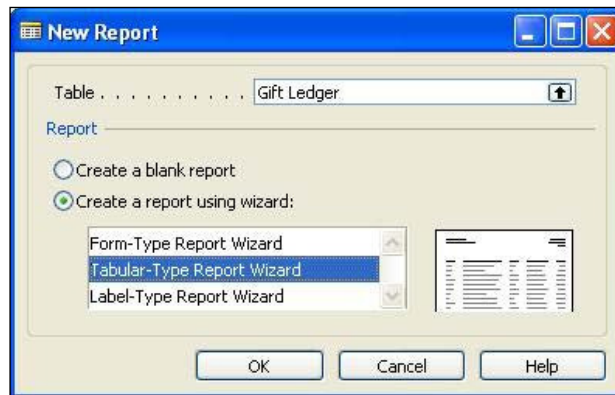


In Step 6 of this flow, there is an option to Create Layout Suggestion in the Visual Studio Report Designer as shown in the following screenshot:



When you choose **Create Layout Suggestion**, the C/SIDE Report Designer will invoke a process that transforms the layout in Sections to a layout in the Visual Studio Report Designer. If a VS RD layout previously existed, the newly created layout will overwrite it. Therefore, this option will normally be used only once, in the initial stages of report design.

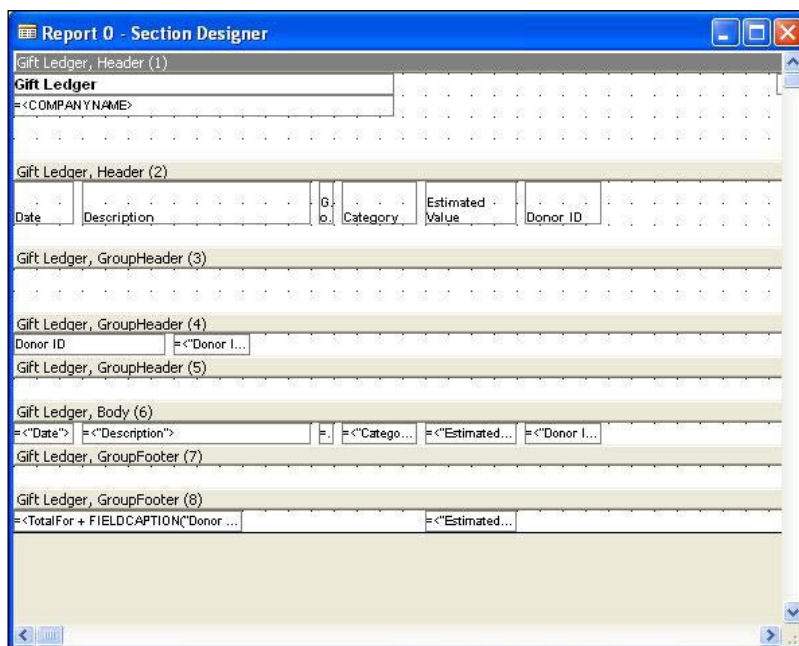
Let's experiment by using the Report Wizard to create a simple report listing the gifts received by ICAN. We will access the Report Wizard in the Object Designer. Click on **Reports | New**, then fill in the Wizard screen as shown in the following image.



Then click on OK and choose fields to display in the report as shown in the following screenshot.



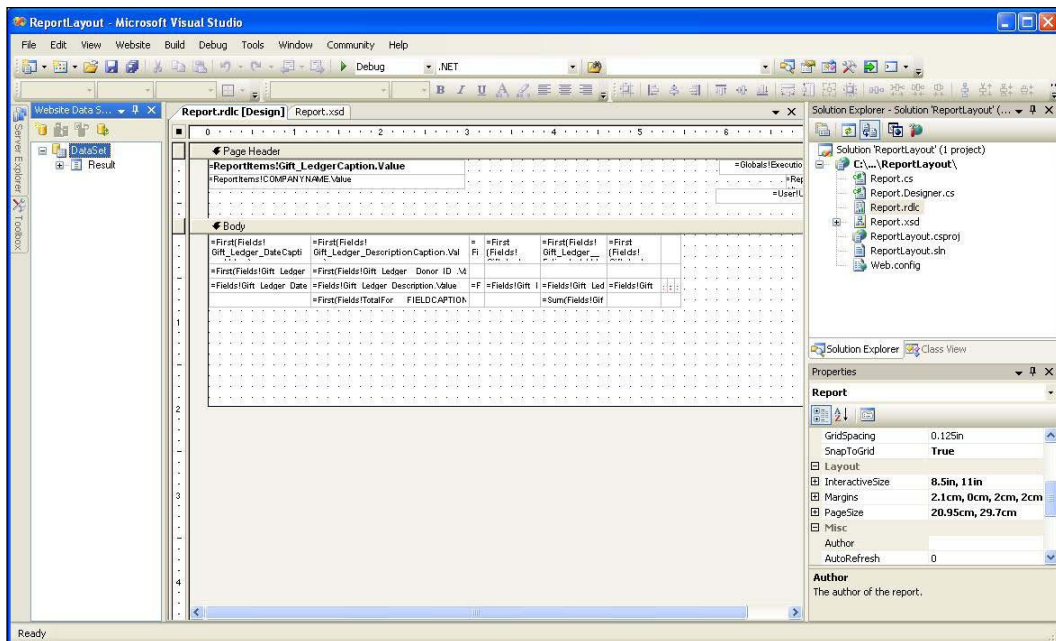
Click on **Next**, then choose the sorting order (that is index or key) that starts with Donor ID. Click on **Next** again and choose to Group the data by Donor ID. Click on **Next** again and choose to create totals for the **Estimated Value** field. One more, click on **Next** and choose the **List Style** for the report, then click on **Finish**. At this point, you will have generated a Classic Client report using the Report Wizard. If you **View | Sections**, you should see a C/SIDE report layout that looks much like the following screenshot.



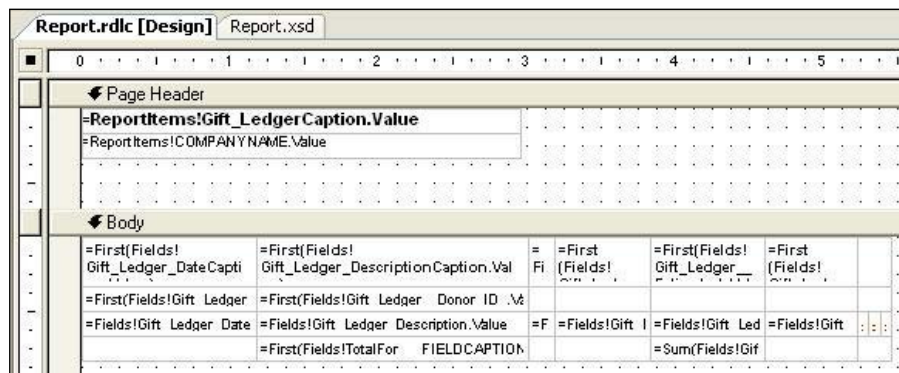
Reports

Let's save our newly generated report so that, if we need to, we can come back to this point as a checkpoint. Click on **File | Save As** and assign the report to **ID 50002** with the **Name of Gifts by Donor**.

Now click on **Tools | Create Layout Suggestion**. The process of transforming the Classic Report Layout to a Visual Studio Report Designer Layout will take a few seconds. When the report layout transformation process completes, you should see a screen that looks very similar to the following screenshot.

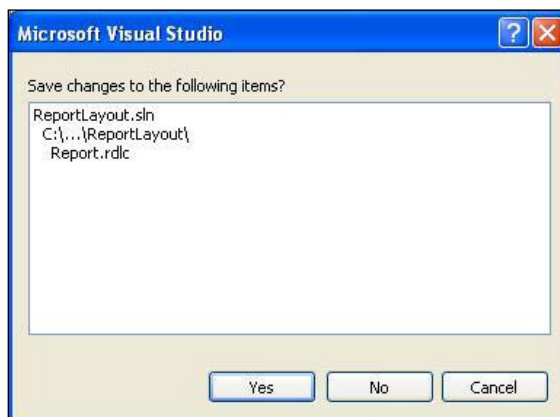


The primary data layout portion of the same VS RD screen is shown in the next image.

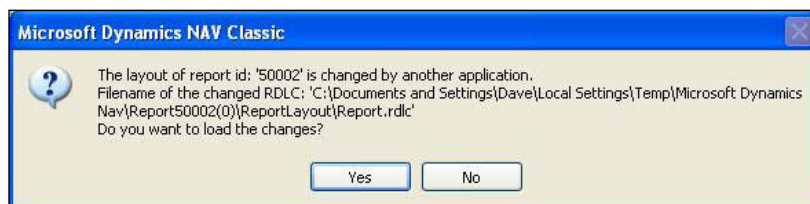


Compare this to the Classic RD data layout we just looked at a couple of steps ago. You will see some similarities and some considerable differences.

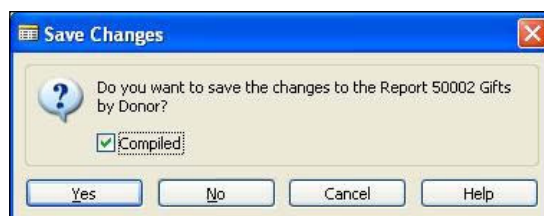
Without doing anything else, let's save the VS RD layout we just created for the RoleTailored Client, then run both versions of the report to see the differences in the generated results. To save the VS RD layout, start by simply exiting the VS Report Designer. Once the VS RD screen closes, you will see the following question.



Respond by clicking **Yes**. Then, when you exit the Classic Report Designer, you will see this question.



Respond by clicking **Yes**. You will then be presented with the following message.

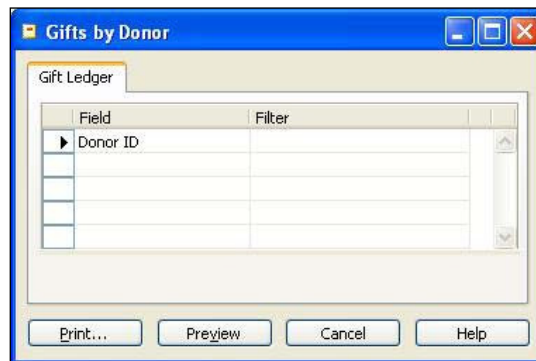


Again, click on **Yes**. If there were an error in the RDLC created within the VS RD (such as an incorrect variable name used), an error message similar to the following would display.



Since, hopefully, we didn't get such an error message, we can proceed to test both the Classic Client and the RoleTailored Client versions of our generated report.

We can test the Classic Client (or C/SIDE RD) version of Report 50002 from the same Object Designer screen where we did our initial design work. Highlight the line for Report 50002 and click on the **Run** button. You should see the following screen:



If we were running this as users, we might want to make a selection of specific Donors here on which to report. As we are just testing, simply click on **Preview** to see our report onscreen. The report will then appear, looking like the following:

Print Preview

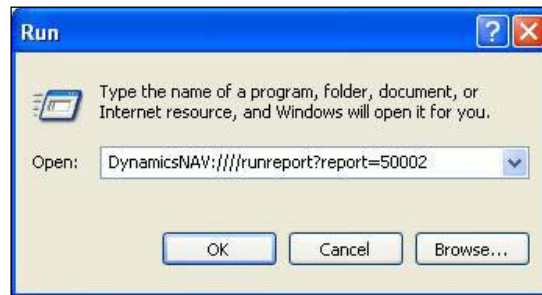
Gift Ledger
CRONUS International Ltd.

Date	Description	Gi or	Category	Estimated Value	Donor ID
Donor ID 1001					
12/15/09	Credit Card	Gi	CC	100.00	1001
12/24/09	Holiday Appeal	Gi	CC	123.00	1001
12/20/09	Check	Gi	CHECK	75.00	1001
11/05/09	House Painting	Gi	SERVICES	0.00	1001
11/24/09	Yard Work	Gi	SERVICES	0.00	1001
12/15/09	Miscellaneous	Gi	SERVICES	0.00	1001
Total for Donor ID				298.00	
Donor ID 1002					
04/15/09	Credit Card	Gi	CC	125.00	1002
05/15/09	Credit Card	Gi	CC	75.00	1002
07/30/09	Check	Gi	CHECK	90.00	1002
Total for Donor ID				290.00	

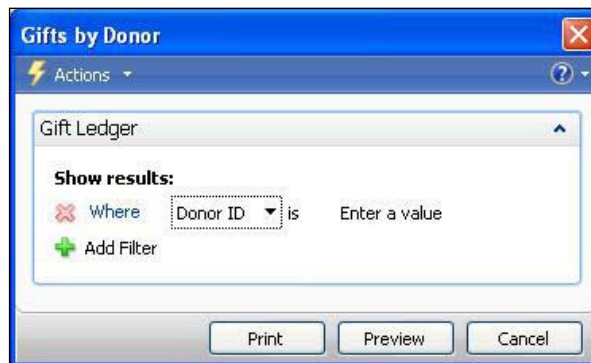
100% Page 1 Report generation... Help

As you can see, with minimum development effort (and a minimum of technical knowledge), we have designed and created a report listing Gifts by Donor with subtotals by Donor. The report has proper page and column headings. Not only that, but the report was initiated with a Request Form allowing application of filters.

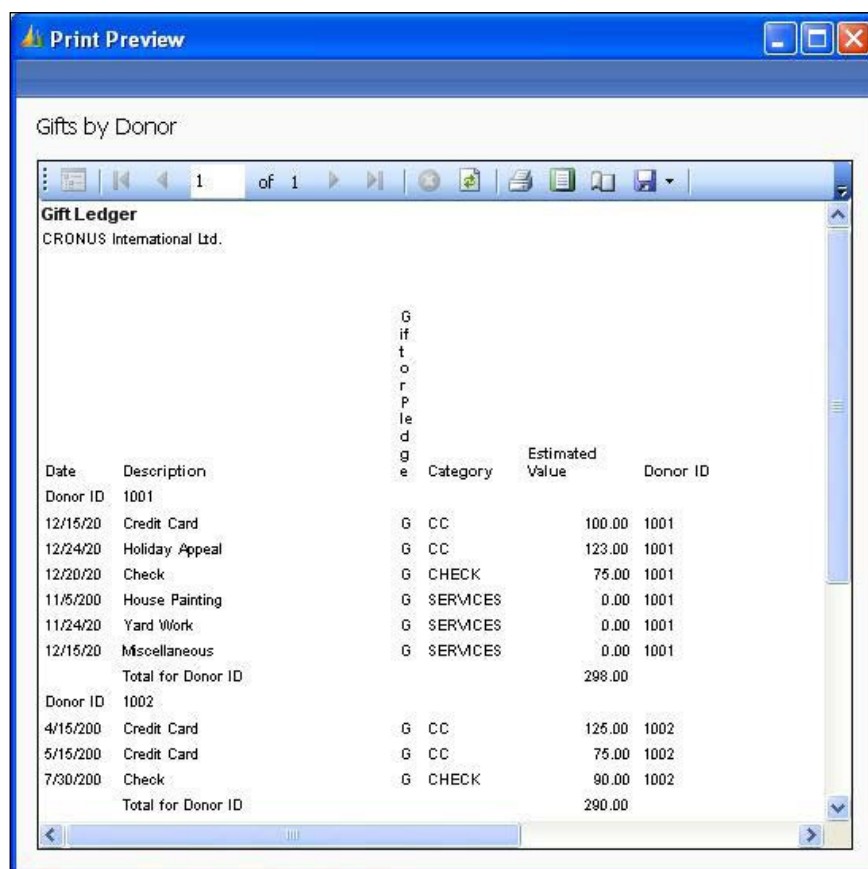
Close the Classic Client report; now let's run the RTC version. Just like we could do with Pages, we will run our Report test from the Windows Run option. Click on **Run** and enter the command to run Report 50002, as shown in the next screenshot.



Click on **OK**. If the RoleTailored Client is not active, after a short pause, it will be activated. Then the Request Page will appear. Compare the look and contents of this Request Page with the one we saw previously for the Classic Client.



As before, click on **Preview** and view the report. Of course, this time we're looking at the RTC version.



Gifts by Donor

CRONUS International Ltd.

Date	Description	Category	Estimated Value	Donor ID
Donor ID 1001				
12/15/20	Credit Card	G CC	100.00	1001
12/24/20	Holiday Appeal	G CC	123.00	1001
12/20/20	Check	G CHECK	75.00	1001
11/5/200	House Painting	G SERVICES	0.00	1001
11/24/20	Yard Work	G SERVICES	0.00	1001
12/15/20	Miscellaneous	G SERVICES	0.00	1001
Total for Donor ID			298.00	
Donor ID 1002				
4/15/200	Credit Card	G CC	125.00	1002
5/15/200	Credit Card	G CC	75.00	1002
7/30/200	Check	G CHECK	90.00	1002
Total for Donor ID			290.00	

As you see, the generated layout in the RTC is very similar to the Classic version. There are some differences in the headings, date formatting, and field alignment among others, but the essence of the report is the same.

This method of automatic transformation is very useful for getting an initial base for a new report or, obviously, for the complete generation process for a simple report where the requirements for layout are not too restrictive.

Learn by experimentation

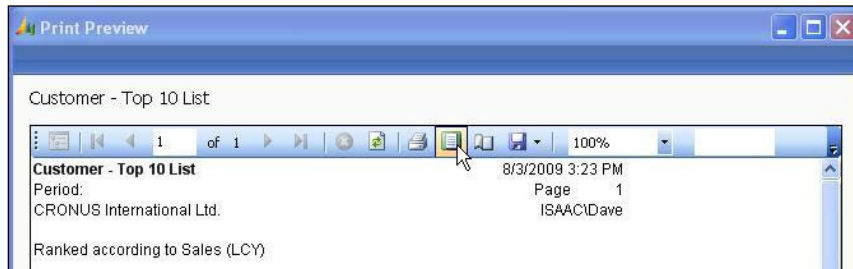
One of the most important learning tools you have available is experimentation. This is one of the areas where experimentation will be extremely valuable and informative. It will be very good for you to know which types of report transformations work well and which do not. The best way to find out is by experimentation. You should create test Classic reports, using a variety of the Report Wizard options, then transform those to RTC reports and examine the results.

You should also work on test copies of some of the standard reports that are part of the NAV system. Make sure you are working on test copies, not the originals! Open the test copy of a report in Design mode, then Create Layout Suggestion. This will wipe out the VS RD layout that came with the report from Microsoft, hence the emphasis on using test copies. You will find that some reports will transform fairly well and some not well at all. The more you test, the better you will be able to determine which RTC report designs can be generated using this approach, and which ones will need to have most or all of their layout design work done manually within the VS RD.

Runtime formatting

When NAV prints a report (to screen, to hardcopy, or to PDF), NAV will format using the printer driver for the currently assigned printer. If you change the target printer for a report, the output results may change depending on the attributes of the drivers.

When you preview a report in the RTC, by default, it will be displayed in an interactive preview mode. This mode will allow you to access all of the dynamic functions designed into the report, functions such as sorting, toggling for expand/collapse display, and drilling into the report. However, it may not look like the hardcopy, if you print it. If you click on the Print Layout button (pointed to by the cursor icon in the following screenshot), then the printer layout version of the report will be displayed.



In most cases, the display on screen in **Preview - Print Layout** mode will accurately represent how the report will appear when actually printed. In some cases though, NAV's output generation on screen differs considerably from the hardcopy version. This appears to most likely occur when the selected printer is either an impact printer (such as a dot matrix) or a special-purpose printer (for example, a barcode label printer).

Inheritance

Inheritance operates for data displayed through report controls just as it does for page controls, but is obviously limited to print-applicable properties. Properties, such as decimal formatting, are inherited. Remember, if the property is explicitly defined in the table, it cannot be less restrictively defined elsewhere. This is one reason why it's so important to focus on table design as the foundation of the system.

Other ways to create RTC reports

In the example we've gone through to create an RTC report, we generated everything using the Report Wizard and the automatic report transformation tools. There are at least two other ways to create RTC reports. They are probably obvious, but let's list them:

- Use the Report Wizard to create the basic report. Then use the Classic RD to modify the Sections, followed by transforming the sections into the VS RD Layout. If you are going to use any data in your final report other than that inserted in sections by the Report Wizard, you will have to include that data in a section. For example, if you want to have some heading fields such as a display of the Filters applied to the report, you will have to add those fields to the sections before moving on to work in the VS RD.
- Create the Classic RD without use of the Report Wizard, in other words, manually design the Data Items and manually insert all data elements into the Sections. If you are not going to run this report as a Classic Client report at all (that is, have no concern about its appearance in the Classic Client environment), this is perfectly acceptable. In this case, it doesn't matter how the Sections are structured or where the data elements appear in the Sections. What matters is that all of the data elements needed by the VS RD are defined in the Sections Designer. Of course, if you want the transformation of the Sections to create a useful VS RD layout, then you should layout the Sections accordingly.

A very important item to remember is that even though Sections created for use in the Classic Client have triggers and those triggers may contain code, the transformation tool will discard that code in the process of creating the VS RD Layout definition. If the Classic Client version of a report has any code embedded in the Sections, it must be removed and redesigned to operate inline within the processing of the Data Items or as part of RDLC property based logic. If a report is to operate in both Classic and RTC environments, you must be careful that such code will work in both without conflict.

Modify an existing RTC report

Not surprisingly, when we modify an existing report, the path into the Visual Studio Report Designer is different than when we are creating a report for the first time. Let's make some modifications to the last report we created, the simple list of Gifts by Donor, Report 50002.

The first step is to open the report in Design mode in the Classic RD. You should know the route there by now, but just in case, it's **Tools | Object Designer | Reports | highlight Report 50002 | Design**. The report will open in the Classic Report Designer.

Assuming that we don't need to keep the Classic version of the report up to date, the only reason we have to do anything within the Classic RD is to update the Sections for any additional required data elements. Before we can judge what additional data we might need, obviously we need to decide what we're doing.

Let's do the following:

- Count the number of donations
- Add the ICAN Campaign data field to the report lines with a heading of "ICAN Campaign"
- Add an interactive expand/collapse feature
- Add interactive sorts by Date and by Amount
- Add a Report Page option to optionally include/exclude gift of Services entries

Before you get started with this effort, you should enter enough test data into the Gift Ledger table to make your testing meaningful. As the system is in the testing stage, it's reasonable to simply **Run the Gift Ledger** table directly from the **Object Designer**. This will give you a basic form showing all the data fields. That form is a good way to enter test data.

We can guess that the interactive functions will not require any additional data fields. Adding the ICAN Campaign data field to the report lines will obviously require defining that data element in sections, so that it will be available to the VS RD. The same can be said for adding a heading field showing the user filter applied to the report.

Adding a donation count could be done by adding the count to the Classic report (that is to Sections). Or perhaps we can find a record Count function in the VS RD that will do the job without a new field. We'll try the latter approach first, then, if we need to do so, we will come back and make additional changes to the data definition.

Adding the ICAN Campaign field to Sections is very simple. With the report open to the DataItem screen in the Report Designer, click on **View | Sections**. That will display the following screen:

Click on the Field Menu, shown in the following image:



This will open a form listing all the fields in the Gift Ledger table. Highlight the desired field, ICAN Campaign, then click on one of the Sections to show a locating crosshairs. Click again to drop the field at the desired spot. It makes sense to put the new field on the same line with the other data fields, but that's not required.

Adding the data field this way will also add a header label field for ICAN Campaign. It can be moved or left where it landed. Position will not matter to the VS RD when we are simply adding the field as a modification to the RTC report. It would matter only if we were to go through the transformation process again (that is, choosing the **Tools | Create Layout Suggestion** option).

The Visual Studio Report Designer layout screen

[illegible]

- [284] -

- **Data Sources:** This panel shows all of the data elements that are available to this report. All data elements that are shown here must have been previously defined in the Classic RD Sections Designer for this report. That is the only source of data for the VS RD.
- **Table Elements:** This column is displayed only when one of the data elements in the report body has been highlighted. These icons represent the different types of reporting table elements (that is lines) that are present. Note that "table" here refers to a data display grid in the layout tool, not a NAV data table.

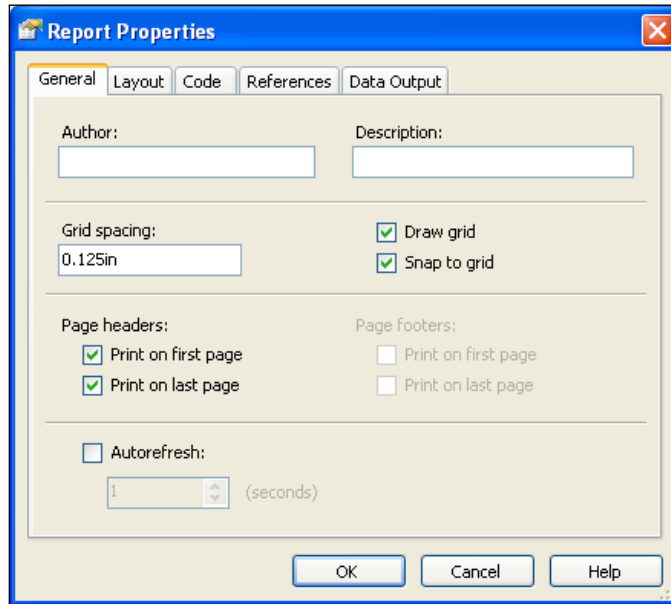
Table Elements—icons and purpose

The following list shows the individual Table Element icons and a description of the purpose of each Table Element type:

	Table header – one is allowed at the top of the layout table. This may be used, for example as the primary location for column headers (or other header information)
	Group header – allowed for each data item which has indented subordinate items
	Table Detail – Any number of these are allowed. Table Detail lines contain information from any of the data sources (i.e. any data that was originally assigned into the Classic RD Sections Designer).
	Group Footer - allowed for each data item which has indented subordinate items. Group Footers are generally used for grouping and totalling of detail data.
	Table Footer – one is allowed at the end of the layout table. The Table Footer can be used for report totalling (or other footer type information).

- **Page Header:** This section's content appears at the top of pages, depending on how properties have been defined.
- **Body Section:** The Body section usually contains a layout table. The layout table is a grid format, which contains the definition of the data that is to be printed repeatedly based on the dataset passed in from the NAV report processing.
- **Element Properties:** This panel shows the properties for the highlighted report element. It could be the properties for the overall Report, for the Page Header, for a Table Element, for an individual Data Control.

- **Property Pages Button:** Selecting this button will display the Property Pages for the highlighted report element. This is an easier to read display of element properties, displaying a more complete set than that shown in the Element Properties panel. The Property Pages for the Report Properties for Report 103 is shown in the following image.
- **Page Footer:** This section's content appears at the bottom of pages, depending on how properties have been defined.



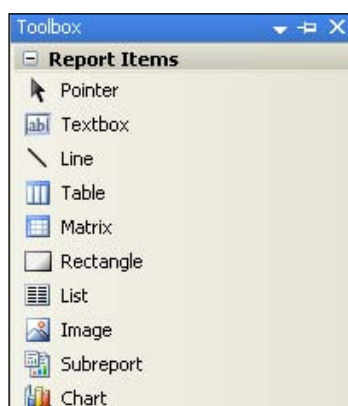
Report Items

SSRS Reports designed in Visual Studio have a number of **Report Items** available, all accessible from the VS RD Toolbox as shown in the screenshot following (note that the very similar term "ReportItems" is the name of a collection of text boxes that can be displayed on a report). Most of these items are not utilized when reports layouts are transformed automatically. But, with thoughtful design work, they are available to the developer to create useful, sophisticated outputs as part of NAV enhancements. There are some instances of creative use of new SSRS reporting capabilities in the standard reports. For example, Report 111 – Customer – Top 10 List offers a choice of graphic displays as well as the option to sort the report on any of several columns of data, when in interactive preview mode.

Among the features that are new to NAV 2009 reporting are the following:

- Interactively hide or show data, expand or collapse details, sort data, change filtering
- Display multiple formats as charts as well as images
- Standard output to PDF and Excel formats
- Drill down to underlying data, drill through to pages and other reports

A snapshot follows of the Toolbox from which Report Items can be selected, for drag-and-drop addition to a VS RD layout.



An excellent method of learning about the properties for each Report Item is to view the Help for that Report Item. The easiest way to view the Help for a Report Item is to display the properties form for that item, then click on the Help button on that form.

One way to view the Properties form for each of the Report Items, once you are in the VS RD layout screen, is to highlight a Report Item of the type of interest, right click on it and then select **Properties**. If you are interested in what properties are available for a Report Item which is not part of the layout you are viewing, you can drag and drop the Report Item to your report layout, examine the properties form, then delete the Report Item from the VS RD layout.

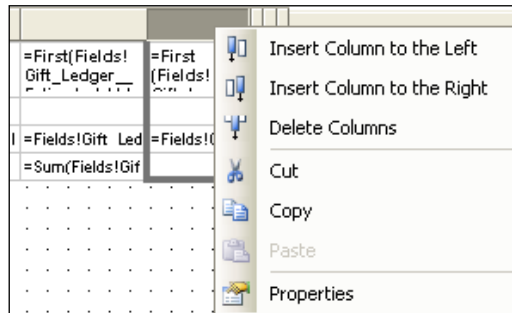
Studying the properties for Report Items (and all of the other components of a NAV system) is highly recommended for the developer who really wants to dig in and learn about the inner workings of the system. Combining that technique with experimentation and studying the makeup of objects in the standard product are very effective ways to become more knowledgeable about design and development options for NAV 2009.

Among the new features in NAV 2009 are properties that can be assigned values based on expressions. The values of the expressions can then be changed dynamically at run time, thus allowing properties to be changed on the fly. In previous versions of NAV, a more limited variation of this capability to dynamically change property values was provided through functions such as `Visible`, `Editable`, and `UpdateFontBold`. These functions are not supported in the RTC, but have been replaced by the ability to assign an expression to the property. Examples are the **Visible** property in Pages and the **Hidden** property in various Report Items (or even the **SHOWOUTPUT** function used in RD Sections). In fact, almost all Report Items can be assigned values based on expressions. Determining how these will behave in a particular situation is a perfect example of where you must let existing designs and your testing be your guide.

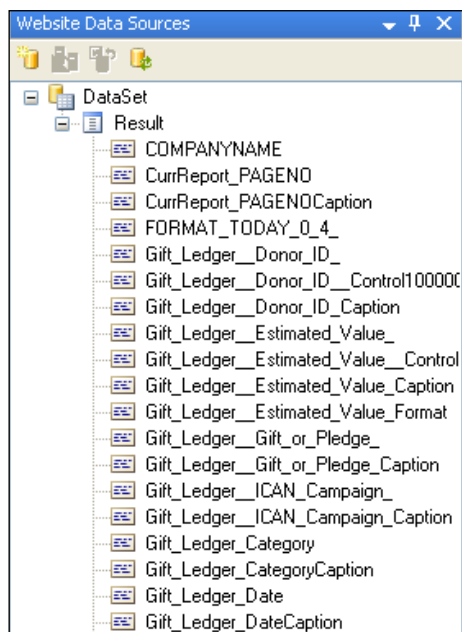
Make the report changes

Having reviewed the VS RD in some detail, let's begin the task of making the changes we defined earlier.

First, we will add the ICAN Campaign column to each data line. In the VS RD showing the original VS RD layout for Report 50002, right-click on the top line of the last data column. Ignore the three very small columns at the right end of the data line. These are **Hidden Fields** which we will explain later. You will see the following:



Then choose the option to **Insert Column to the Right**. That will (surprise!) insert another column—to the right. Now expand the **Data Sources**, so that you can access the fields we added earlier to the Classic Sections Designer. Those fields are **Gift_Ledger_ICAN_Campaign_** (data) and **Gift_Ledger_ICAN_Campaign_Caption** (header).



Grab, drag-and-drop the data field into the new column, third row (table detail row). Grab, drag-and-drop the caption field into the new column, top row (table header row). Save, save, compile, and test. Depending on your data, the new column on your report may look similar to the following:

ICAN Campa ign
2009ANNUA L
2009HOLIDA Y
2009HOLIDA Y

If so, then you need to change the properties of the data field so that the full data contents can be displayed. Return to the VS RD, highlight the field containing **Gift_Ledger_ICAN_Campaign_**. Confirm that the property **Layout | CanGrow** is False, rather than True. This will keep the data field from wrapping to the next line when it exceeds the width of the available space. Instead, it will be truncated when it exceeds the available space (obviously, you may prefer different behavior in different situations). The other change required in this case is to revise the **Size | Width** property to a larger number. After those changes are made, the next test gives the following results for a sample of the ICAN Campaign column of data.

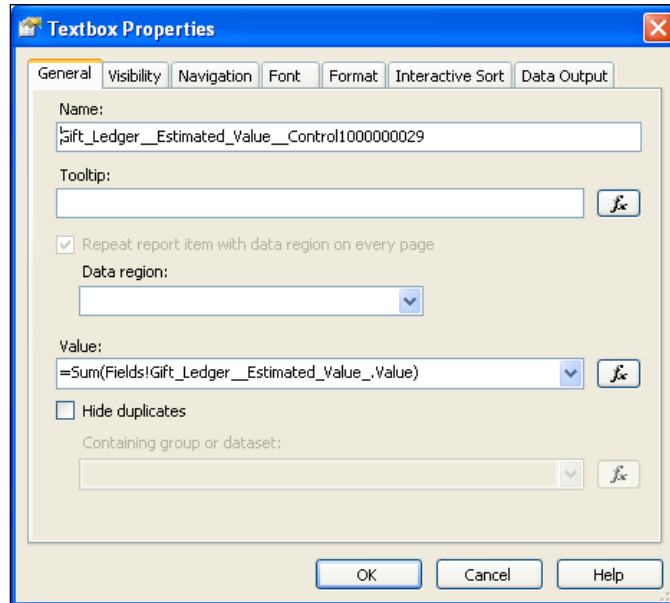
ICAN Campaign
2009ANNUAL
2009HOLIDAY
2009HOLIDAY

Our second change is to add a Gift count to each total line. As all we need to do in this case is to count the detail lines printed, we should check if we can use a function which is built into the VS RD. We can count the lines if we can count the instances of one of the data elements, such as the Donor ID. Let's go exploring to see if we can find the tools and the technique to accomplish our goal.

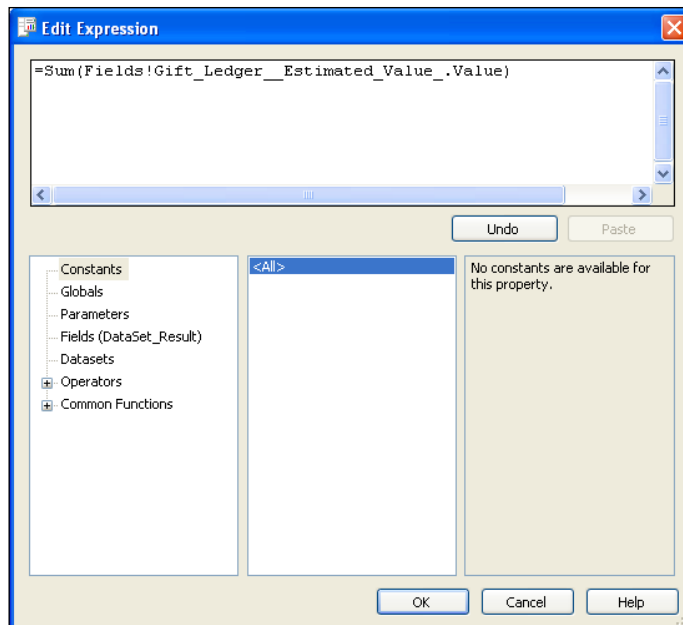
We will start our exploration by investigating how a report transformation generated total has been created. Right click on the textbox containing the Estimated Value total (shown in the following screenshot):

	=First(Fields! Gift_Ledger_DateCapti	=First(Fields! Gift_Ledger_DescriptionCaption.Val	=	=First (Fields! CAN_	=First(Fields! Gift_Ledger_ ESTIMATED_VAL	=First (Fields! CAN_	=First(Fields! Gift_Ledger_CAN_ ESTIMATED_VAL		
	=First(Fields!Gift_Ledger	=First(Fields!Gift_Ledger Donor ID .Val							
	=Fields!Gift_Ledger Date	=Fields!Gift_Ledger Description.Value	=F	=Fields!Gift	=Fields!Gift_Led	=Fields!Gift	=Fields!Gift_Ledger	:	:
		=First(Fields!TotalFor FIELDCAPTION			=Sum(Fields!Gif				

Then click on the Properties option. As shown in the following screenshot, the **Value** field contains `=Sum(Fields!Gift_Ledger_Estimated_Value_.Value)`, which is not listed in the **Data Sources** window. But, as `Gift_Ledger_Estimated_Value_.Value` is one of the items in the **Data Sources window**, it seems reasonable to assume that `=Sum` is a VS RD function and that we might find a Count function if we keep looking.

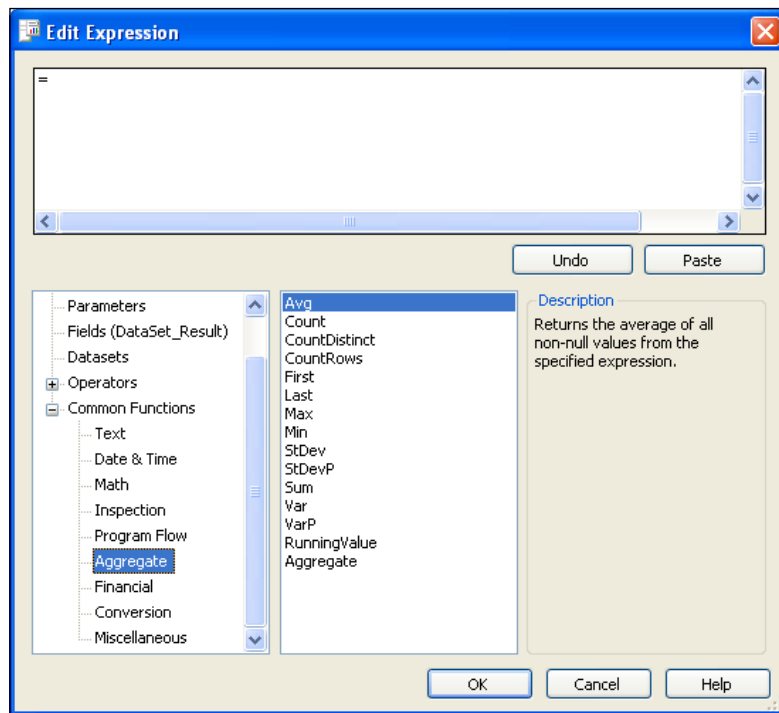


As we've determined that the **Value** field contains an expression, let's go back to the text box, right-click again, and choose **Expression....** That will open the following form, where we can see the tools that are used to construct an expression like the one that sums the total gift amounts.

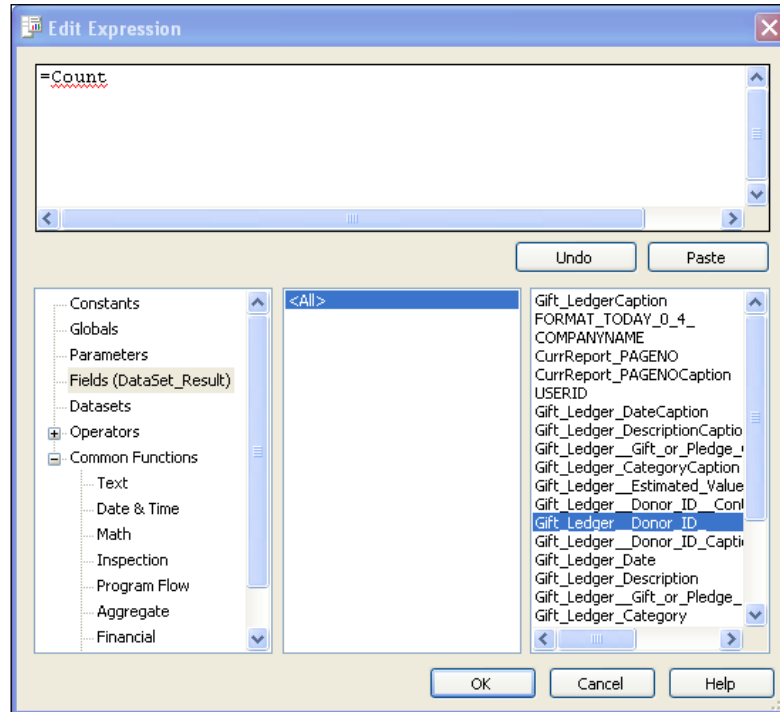


Our task now is to find a count function (or equivalent). We can continue our exploring, but let's move to a blank instance of the **Edit Expression** form, so we don't accidentally change the existing, working expression. Close the **Edit Expression** form for the **Gift_Ledger_Estimated_Value** and open an **Edit Expression** form for the Textbox in the same row, but in the column containing the Donor ID field (that is, the next column to the right in our example).

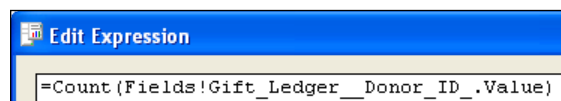
With that empty **Edit Expression** form open, if we can find a count function, perhaps we can build an expression to give us our desired result. Using the very practical "peek and poke" method (that is, clicking on various fields to see what we can find), we uncover the following.



There's the **Count** function we were hoping to find (along with quite a few other functions we can use in future situations). Double-click on the **Count** function or click on the Paste button, and the Count function will be pasted into the workspace above. Looking at the expression for the Estimated Value total, we can guess that now we want to identify what to count, in this case the Donor ID instance. If we click on the left pane, **Fields (DataSet_Result)**, then we will see a list of all of the available data fields. Included in those is **Gift_Ledger_Donor_ID**, as shown in the following screenshot. The wavy line under the Count function in the workspace simply indicates that we don't have a complete operable expression defined yet.



Double-click on the field **Gift_Ledger__Donor_ID** (or highlight and click on **Paste**). The field name will appear in the **Edit Expression** workspace after the **Count** function. You will likely still see the wavy underline, indicating you're still not quite done. If you look at the expression for the **Sum**, you will see that it begins with an equals sign (=) and the field name is enclosed in parentheses. Let's make those additional changes. The final expression should look like the following:



Once you have that, you can close the Edit Expression form. You may notice at this point that the font for the new Count expression is different from (for example, larger) the other fields. If you want all of the fonts to be the same, obviously, you will need to change the font property of the new field to match the old fields. Once that is done, exit and save the VS RD layout changes, save and compile the report, and test it. You should now see a count of the gifts for each donor on the group total lines, looking something like the following:

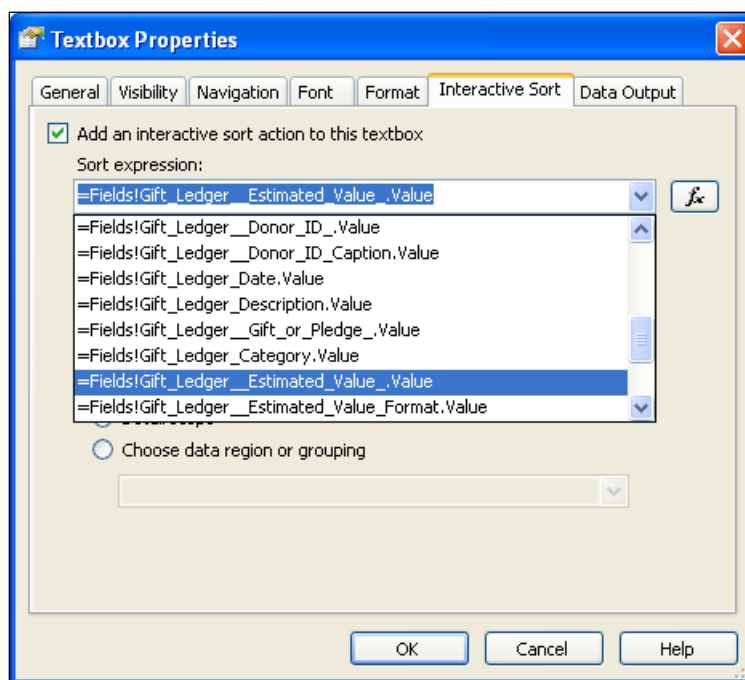
Gifts by Donor						
Gift Ledger						
CRONUS International Ltd.						
4/16/20						
Date	Description	Gift Category	Estimated Value	Donor ID	C/AN Campaign	
Donor ID	1001					
12/15/2009	Credit Card	G CC	100.00	1001	2009ANNUAL	
12/24/2009	Holiday Appeal	G CC	123.00	1001	2009HOLIDAY	
12/20/2009	Check	G CHECK	75.00	1001	2009HOLIDAY	
11/5/2009	House Painting	G SERVICES	0.00	1001		
11/24/2009	Yard Work	G SERVICES	0.00	1001		
12/15/2009	Miscellaneous	G SERVICES	0.00	1001		
	Total for Donor ID		298.00	6		
Donor ID	1002					
8/18/2009	Credit Card	G CC	125.00	1002	2009ANNUAL	
4/15/2009	Credit Card	G CC	75.00	1002	2009ANNUAL	
7/30/2009	Check	G CHECK	90.00	1002	2009OPERATIONS	
	Total for Donor ID		290.00	3		
Donor ID	1003					
8/25/2009	In Memoriam	G CHECK	1,000.00	1003	2009OPERATIONS	
	Total for Donor ID		1,000.00	1		

If this were to be a production report to be given to a client, it is likely that we would work on the layout some more, perhaps more clearly identifying the gift count total. But we certainly have achieved our original goal of obtaining and printing the count.

Some interactive report capabilities

Our third change is to add some interactive capabilities to our report. We want the report to display in either a summary or detail format, and we want to allow sorting the detail into different sequences. Return again to the VS RD layout, focused on Report 50002.

Let's allow the users to sort on the gift amount or on the campaign. And, to provide full flexibility, let's also allow sorting on the gift date. Right-click on the heading for the column on which we wish to control the sort, and display the properties form—let's start with gift amount. Click on the **Interactive Sort** tab. Click on the option **Add an interactive sort action to this textbox**. Then choose the field on which the sort should occur. In this case, that's the **=Fields!Gift_Ledger__Estimated_Value_.Value**:



Click on **OK**, and then perform a similar set of actions on the layout table column for the ICAN Campaign and Date fields. Once you've done that, exit VS RD, saving the changes, then save, compile and test the changed report design. You should see a report, the top of which looks something like the following. There will be pairs of up/down arrowheads that can be used for sort control. The last one of them used will show just the option not invoked (that is, only an up or a down arrowhead, not both).

Gifts by Donor

Gift Ledger 4/16/

CRONUS International Ltd.

Date	Description	Gift to Donor P e Category	Estimate ed Value	Donor ID	C/AN Campaign
Donor ID	1001				
12/24/2009	Holiday Appeal	G CC	123.00	1001	2009HOLIDAY
12/15/2009	Credit Card	G CC	100.00	1001	2009ANNUAL
12/20/2009	Check	G CHECK	75.00	1001	2009HOLIDAY
11/5/2009	House Painting	G SERVICES	0.00	1001	
11/24/2009	Yard Work	G SERVICES	0.00	1001	
12/15/2009	Miscellaneous	G SERVICES	0.00	1001	
	Total for Donor ID		298.00	6	
Donor ID	1002				
8/18/2009	Credit Card	G CC	125.00	1002	2009ANNUAL
7/30/2009	Check	G CHECK	90.00	1002	2009OPERATIONS
4/15/2009	Credit Card	G CC	75.00	1002	2009ANNUAL
	Total for Donor ID		290.00	3	

The work required to implement our other interactive capability, **Expand/Collapse Detail**, is very similar to what we just did for the interactive sorting. Once again return to the VS RD for the layout of Report 50002. This time click on one of the layout table fields, so that the Table Element icons are visible. Then right click on the Table Detail icon.



Choose **Properties**, this time you will not get a Properties form, but will use the VS Property window (which defaults to the lower right panel of the layout screen, but is movable anywhere within the VS RD frame). Find the **Visibility** property group, expand it so that you can see the **Toggle Item** property. Click on the selection caret at the right end of the property space and choose the field with which you would like to control the Expand/Collapse option. The **Gift_Ledger_DateCaption** field is a good choice, because it is the topmost left field, so the Expand/Collapse option will be quite visible to the user there. After you exit, save, compile, and run the report. You will see the Expand/Collapse icon in the top left of the report, as shown in the following screenshot, where the detail data is expanded (that is visible):

Gifts by Donor

Gift Ledger 4/16

CRONUS International Ltd.

Date	Description	Gift to or from the Category	Estimated Value	Donor ID	C/AN Campaign
Donor ID	1001				
12/15/2009	Credit Card	G CC	100.00	1001	2009ANNUAL
12/24/2009	Holiday Appeal	G CC	123.00	1001	2009HOLIDAY
12/20/2009	Check	G CHECK	75.00	1001	2009HOLIDAY
11/5/2009	House Painting	G SERVICES	0.00	1001	
11/24/2009	Yard Work	G SERVICES	0.00	1001	
12/15/2009	Miscellaneous	G SERVICES	0.00	1001	
	Total for Donor ID		298.00	6	

In the next screenshot, the detail data is collapsed (that is, hidden):

Gifts by Donor

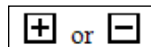
Gift Ledger

CRONUS International Ltd.

4/16/2

Date	Description	Gift to People Category	Estimated Value	Donor ID	C/AN Campaign
Donor ID	1001				
	Total for Donor ID		298.00	6	
Donor ID	1002				
	Total for Donor ID		290.00	3	
Donor ID	1003				
	Total for Donor ID		1,000.00	1	
Donor ID	1004				
	Total for Donor ID		100.00	1	
Donor ID	1005				
	Total for Donor ID		1,047.50	4	
Donor ID	1006				
	Total for Donor ID		700.00	1	

As you can see, the Expand/Collapse icon will be either plus or minus, depending on whether the related data is currently visible (plus) or hidden (minus). You should also experiment with using this feature to make columns on a report be visible or hidden, at the user's option.



Page Header fields

Fields that are displayed in the Page Header or Page Footer sections of the VS RD cannot contain variable data. As a result, a couple of different approaches are used to be able to display "normal" report headings that do contain variable data.

One approach is the use of **Hidden Fields**. In the following screenshot at the point of the cursor arrow, three Hidden Fields are present, which were all created as part of the automatic transformation process of **Create Layout Suggestion**. These fields are essentially similar to other VS RD data fields, except they are set to not be visible, and are only used to manage data rather than display it directly.



Hidden Fields that are created by the Report Transformation process are set to red by default (and by convention). Hidden Fields that are created in the C/SIDE RD to hold special data or logic should be set to yellow (see the Help titled "How to: Add and Identify Hidden Fields"). This color coding makes the hidden fields' purpose easier to identify for software maintenance.

Another approach takes advantage of the ReportItems collection. This is particularly useful for Page Header and Page Footer. The ReportItems collection contains all of the elements that are in the report Body. You can put a field on the report Body, make it invisible, then reference it from the header or footer via the ReportItems collection. In the layout generated by the Create Layout Suggestion transformation, this is the way the Report Caption, Company Name and Page Number information is displayed.

Page Header						
rCaption.Value			=Globals!ExecutionTime			
value			=Repo =GI			
			=User!UserID			
Body						
t(Fields!	=	=First	=First(Fields!	=First	=First(Fields!	
Ledger_DescriptionCaption.Val	Fi	(Fields!	Gift_Ledger__	(Fields!	Gift_Ledger__CAN_	
(Fields!Gift Ledger Donor ID .Va						
ds!Gift Ledger Description.Value	=F	=Fields!Gift I	=Fields!Gift Led	=Fields!Gift	=Fields!Gift Ledger	
(Fields!TotalFor FIELDCAPTION		=Sum(Fields!Gif	=Count(Field:			

In this screenshot, the leftmost of these fields handles the Company Name, the second handles the Report Caption, the third handles the Page No. The fields defined in the Page Header then refer to the Hidden Field textboxes, thus overcoming the limitation on displaying variable data from Data Sources.

Request Page

A Request Page is a page that is executed at the beginning of a report. Its presence or absence is under developer control. A Request Page looks similar to the following screenshot based on one of the standard system reports, the Customer - Order Detail report, Report 108.

Customer - Order Detail

Actions

Options

Show Amounts in LCY: ☐

New Page per Customer: ☐

Print to Excel: ☐

Customer

Sorting: No.

Show results:

Where No. is Enter a value

And Search Name is Enter a value

And Priority is Enter a value

+ Add Filter

Limit totals to:

+ Add Filter

Sales Order Line

Print Preview Cancel

There are three FastTabs in this page. The Customer and Sales Order Line FastTab are tied to the data tables associated with this report. These FastTabs allow the user to define both data filters and Flow Filters to control the report processing. The Options FastTab exists, because the software developer decided to allow some additional user options for this report.

As our example report only has one table, the default Request Page for it has only one FastTab. As we have not defined any of the processing options that would require user input before the report is generated, we have no Options FastTab (see following screenshot).

Gifts by Donor

Actions

Gift Ledger

Show results:

Where Donor ID is Enter a value

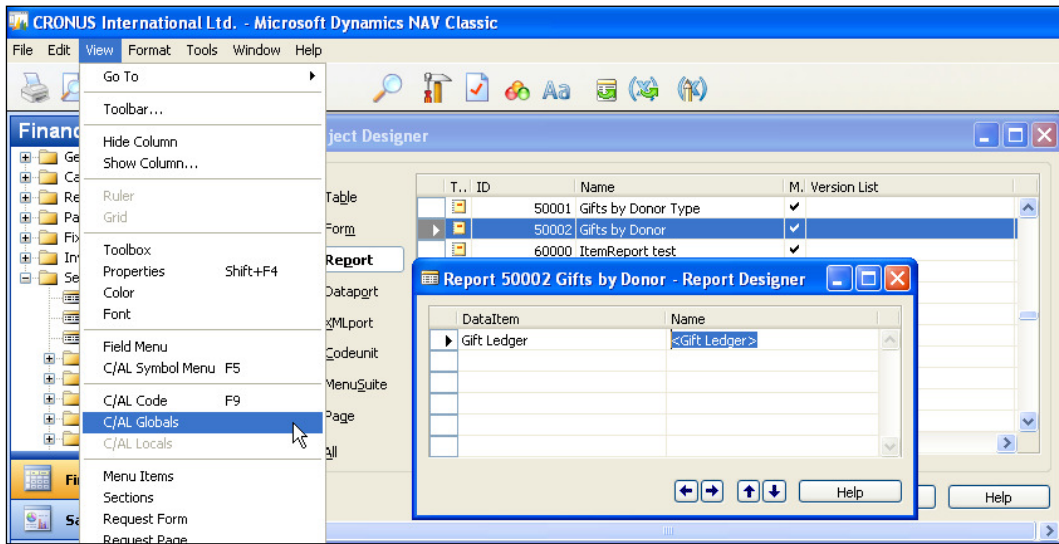
+ Add Filter

Print Preview Cancel

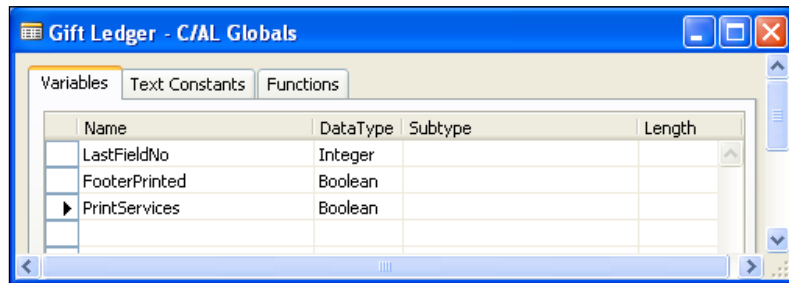
Our final enhancement of our Gifts by Donor report will be to add the option to allow the user to choose whether or not the report detail will include Gifts with a Category of SERVICES.

There is some very simple C/AL programming needed to support providing the user with this option:

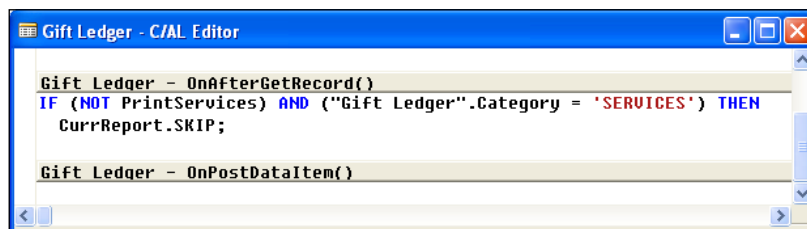
1. Open Report 50002 in the Classic RD.
2. Access the C/AL Globals form from **View | C/AL Globals**.



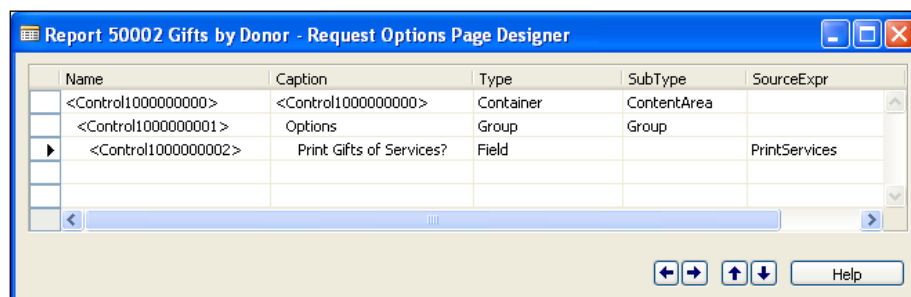
3. Add a new variable to the C/AL Globals as shown in the following screenshot.



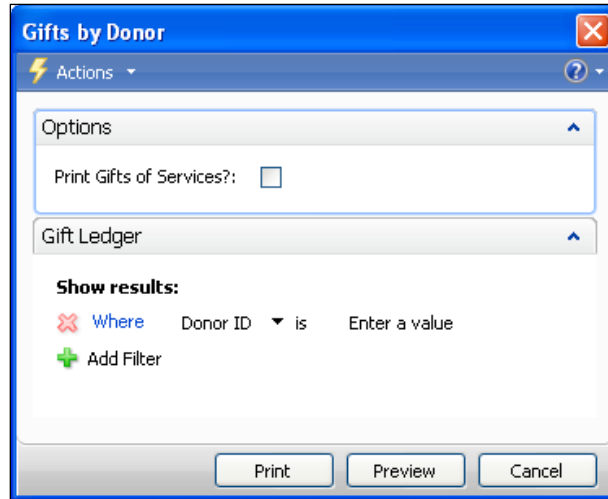
4. Exit the C/AL Globals form. Then click on the Gift Ledger DataItem and press F9. That will take us into the C/AL Editor. Add a line of C/AL code which will skip all Gift Ledger entries where the Category is 'SERVICES', unless the **PrintServices** variable is set to TRUE (see the following screenshot):



Finally, we need to design a Request Page that will display our Options FastTab along with the FastTab for filters on the Gift Ledger entries. Our Request Page design follows along the basic lines of the pages we created in Chapter 4, *Pages – Tools for Data Display*. The first line is a simple Container definition. The second line is a Type Group line, **SubType** Group, with a **Caption** of Options. The third line is **Type** Field with a **SourceExpr** equal to Global Variable (**PrintServices**) that we defined a moment ago. The **Caption** for this third line is the screen label we want for the variable, so that the user will understand the option available to them. If there were additional variables to be entered by the user, there would need to be additional field lines similar to the one in this page design. The page design is shown in the following screenshot:



When this is all done, the modified report, complete with the new Request Page design, should be saved and compiled, then **Run** for a test. Your result should look similar to the following screenshot. When you run the report with the **Print Gifts of Services?** option checked, the SERVICES entries should be printed. When the option is not checked, those entries should not be printed.



Processing-Only reports

One of the report properties we reviewed earlier was **ProcessingOnly**. If that property is set to **Yes**, then the report object will not output a report, but will simply do the processing you program it to do. The beauty of this capability is that you can use the built-in processing loop of the NAV report object along with its sorting and filtering capabilities to create a variety of data updating routines with a minimum of programming. Use of report objects also gives you access to the Request Page to allow user input and guidance for the run. You could create the same functionality using codeunit objects and programming all of the loops, the filtering, the user-interface Request Page, and so on yourself. But with a Processing-Only Report, NAV takes the load off you.

When running a Processing-Only object, at the beginning you see very little difference as a user. You see that there is no visible output at the end of processing, of course. However, at the beginning, the Processing-Only Request Page looks very much as it would for a printing report, except that the Print and Preview choices are not available. Everything else looks the same.

Creating a report from scratch

Even when you're going to create a report that cannot be done with the Report Wizard, it's still a good idea to use the Report Wizard as a starting point, if feasible. Choose the primary table in terms of data flow and layout, then rough out the report using the Wizard. Once that is done, begin modifying the report using the Designer, in order to add parent and child data items, additional total levels, and so on.

If your final target report is to be an Invoice format, for example, you could begin the report development by using the Wizard to lay out the Invoice Detail line. You would then use the Designer to create the Invoice Header sections and the appropriate Footer sections.

A good designer defines the goal before starting development. To work the other way around is known as the "Ready, Fire, Aim" approach. Obviously, the first step is to define the need the report is to satisfy, and what data should be displayed in order to meet that need.

A wide variety of design decisions must be made, including factors such as the reporting sequence (that is the key to be used), what subtotals and totals should be calculated, what format will be easiest to use, and so on. The totaling requirements are often the determinant of what key will be appropriate.

A part of the design and development process is testing. Quite often, when a report is developed, there isn't a lot of meaningful test data available. Frequently, therefore, another task is to design and create quality test data. Testing should be done incrementally as individual changes are made. If too many changes are made without an intermediate test, it may be difficult to identify the source of problems. It is a good idea to compare the generated result to your original layout and identify what remains to be done. It's also always useful at this point to also compare the printout of the generated report to the sections, to see what controls were used for each of the displayed fields. From that information, you can begin to plan how you want to manually change the sections to make the final report look like your original design layout.

A key point is to realize that even though the Wizards are of limited capability, they still can be used to make the work a lot easier. This is especially true for relatively straightforward reports. There will certainly be occasions when using the Report Wizard is simply not useful. In that case, you will begin with a totally blank slate. There will also be cases where you start with the Wizard's output and strip out some of what the Wizard puts in. The reason for using the Wizard is not the high quality of what it generates (it's adequate, but not very elegant), but for the time it saves you.

Creative report plagiarism

Just as we discussed in the chapter on pages, when you want to create a new report of a type that you haven't done recently (or at all), it's a good idea to find another report that is similar in an important way and study it. At the minimum, you will learn how the NAV developers solved a data flow or totaling or filtering challenge. In the best case, you will find a model that you can follow closely, respectfully plagiarizing a working solution, thus saving yourself much time and effort.

Often, it is useful to look at two or three of the standard NAV reports for similar functions to see how they are constructed. There is no sense in re-inventing the wheel (recreating the design for a report of a particular type) when someone else has not only invented a version of it already, but provided you with the plans and given you the ability to examine the C/AL code and the complete structure of the existing report object.

When it comes to modifying a system such as NAV, plagiarism is a very effective research and design tool. In the case of reports, your search for a model may be based on any of the several key elements. You might be looking for a particular data flow approach and find that the NAV developers used the Integer table for some Data Items (as many reports do).

You may need a way to provide some creative filtering similar to what is done in an area of the standard product. You might want to provide user options to print either detail or a couple of different levels of totaling, with a layout that looks good no matter which choice the user makes. You might be dealing with all three of these design needs in the same report. In such a case, it is likely that you are using multiple NAV reports as your models, one for this feature, another for that feature, and so forth.

If you have a complicated, application-specific report to create, you may not be able to directly model your report on a model that already exists. But, often you can still find ideas in standard reports that you can apply to your new design. You will almost always be better off using a model rather than inventing a totally new approach.

If your design concept is too big a leap from what has been done previously, you will find the tools available have built-in constraints, which make it difficult to achieve your goal. In other words, generally you should build on the obvious strengths of C/AL. Creating entirely new approaches may be very satisfying (when it works), but too often it doesn't work well and costs a lot.

Summary

In this chapter, we focused on the structural and layout aspects of NAV Report objects. We studied the primary structural components, data and format, along with the Request Page. We also experimented with some of the tools and modestly expanded our ICAN application.

In the next chapter, we are going to begin exploring the key tools that pull the other pieces together, the C/SIDE development environment, and the C/AL programming language.

Review questions

1. Reports are fixed displays of data extracted from the system, designed only for hardcopy output. True or False?
2. Data from tables can be defined either in Sections in the Classic Report Designer or in the Data Sources area of the Visual Studio Report Designer. True or False?
3. Within the Visual Studio Report Designer, you can define calculations on table data or can define literals for display on a report. True or False?
4. Visual Studio RTC reports only allow one header section and one footer section. True or False?
5. By convention, hidden fields in report layouts are color coded. Which two colors are used and what do they indicate?
 - a. Blue – indicating they were created by the Report Transformation process
 - b. Red – indicating they were created by the Report Transformation process
 - c. Yellow – indicating they were created by the developer to hold special data or logic
 - d. Green – indicating that they can be visible or invisible at the user's option
6. In Classic reports, C/AL code can be embedded in Sections. In RTC reports, hidden fields provide many similar capabilities. True or False?
7. Which two of these choices describe some (not all) of the steps involved in defining a new RTC report?
 - a. Define DataItems, define Working Data, define C/AL logic
 - b. Define Sections Layout, define data in Sections, define C/AL logic
 - c. Create VS Layout, define VS Dynamic Options, create Request Page
 - d. Create Request Form, create VS Layout, define DataItems

8. NAV 2009 provides the ability to output to what targets? Choose three:
 - a. Web pages
 - b. PDF files
 - c. Word files
 - d. Excel files
 - e. Installed printers
9. NAV Report data flow includes a structure that provides for "child" DataItems to be fully processed for each record processed in the "parent" DataItem. What is the visible indication that this structure exists in a report DataItem Design form? Choose one.
 - a. Nesting
 - b. Indentation
 - c. Linking
10. A report that only does processing and generates no printed output can be defined. True or False?

6

Introduction to C/SIDE and C/AL

A special kind of beauty exists which is born in language, of language, and for language – Gaston Bachelard

In the preceding chapters, we introduced the basic building block objects of NAV tables, pages, and reports. In each of these, we reviewed the triggers within various areas such as the overall object, controls, data items, the Request Page, and so on. The purpose of each trigger is to be a container in which C/AL code can reside. The triggers are "fired", that is invoked or executed, when certain predefined events occur.

In this chapter, we're going to begin learning more about the C/AL programming language. We'll start with the basics, but we won't spend a lot of time on those. Many things you already know from programming in other languages apply to C/AL. In addition, many of the basic definitions can be found in the online C/AL Reference Guide that is part of the NAV 2009 Help (as well as in the MSDN Library sections in Microsoft Dynamics NAV).

The goal of this chapter is to help you in learning to navigate, productively use the C/SIDE development environment, and to be comfortable in C/AL. We'll focus on the tools and processes that you will use most often. You will also learn concepts that you can apply in more complex tasks down the road.

As with most of the programming languages, you have considerable flexibility for defining your own model for your code structure. However, when you are inserting new code within existing code, there's a strong argument for utilizing the model and the structure that already exists in the original code. When you feel compelled to improve on the model of the existing code, do so in small increments and take into account the effect of your changes on upgradability.

Essential navigation

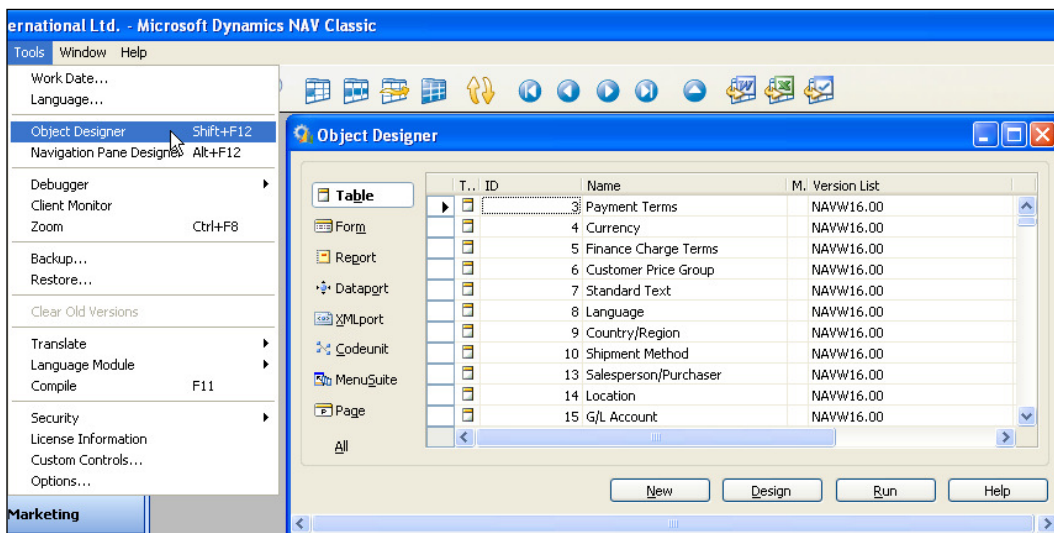
All of the development for NAV normally takes place within the C/SIDE environment with extension, as you saw in Chapter 5, *Reports*, into the Visual Studio Report Designer. The only exception to using C/SIDE is the possibility of doing development in Text mode using a text editor or the Developer's Toolkit. That approach is generally only appropriate for simple modifications to existing objects. In general, the recommendation is "stick with C/SIDE".

As an Integrated Development Environment, C/SIDE provides you with a reasonably full set of tools for your C/AL development work. By design, C/SIDE is not nearly as fully featured as Microsoft's Visual Studio, but the features it has are quite useful. C/SIDE includes a smart editor (it knows C/AL, though sometimes not as much as you would like), the one and only C/AL compiler, integration with the application database, and tools to export and import objects both in compiled format and as formatted text files.

We'll explore each of these C/SIDE areas in turn. Let's start with an overview of the Object Designer.

Object Designer

All the NAV object development work starts from within the C/SIDE **Object Designer**. The **Object Designer** is accessed by selecting **Tools | Object Designer** or by pressing *Shift+F12* keys, as shown in the following screenshot:



The type of object on which you're going to work is chosen by clicking on one of the buttons on the left side of the **Object Designer** screen. The choices match the eight object types **Table**, **Form**, **Report**, **Dataport**, **XMLport**, **Codeunit**, **MenuSuite**, and **Page**. When you click on one of these, the **Object Designer** screen display is filtered to show only that object type. There is also an **All** button, which allows objects of all types to be displayed on screen.

No matter which object type has been chosen, the same four buttons appear at the bottom of the screen: **New**, **Design**, **Run**, and **Help**. But, depending on which object type is chosen, the effect of selecting one of these options changes. When you select **Design**, you will open the object that is currently highlighted in a Designer specifically tailored to work on that object type. When you select **Run**, you will be requesting the execution of the currently highlighted object. The results of that, of course, will depend on the internal design of that particular object. When you select **Help**, the **C/SIDE Help** screen will display, positioned at the general Object Designer **Help for Developing Objects**.

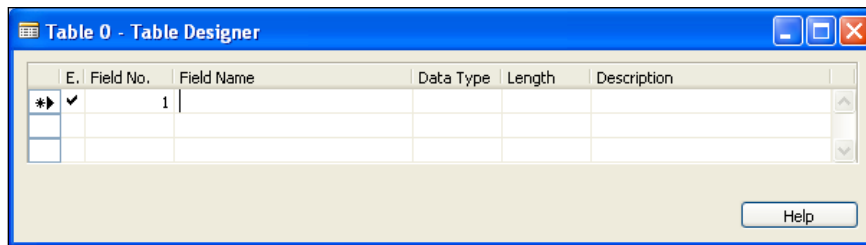
Starting a new object

When you select **New**, the screen you see will depend on what type of object has focus at the time you make the **New** selection. In each case, you will have the opportunity to begin creating a new object and you will be presented with the Designer for that particular object type.

The **New** Designer screens for each of the object types are discussed in the following sections.

Table Designer

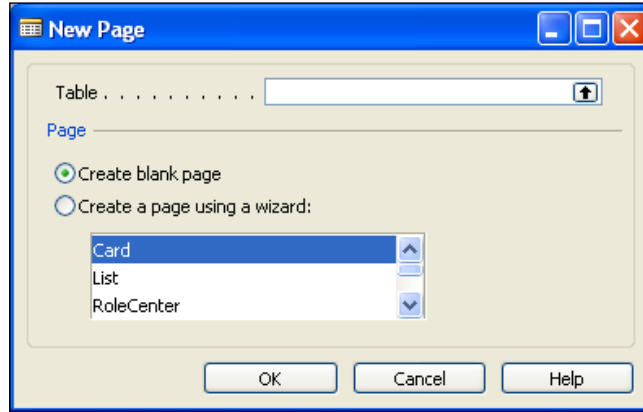
The **Table Designer** screen is shown in the following screenshot:



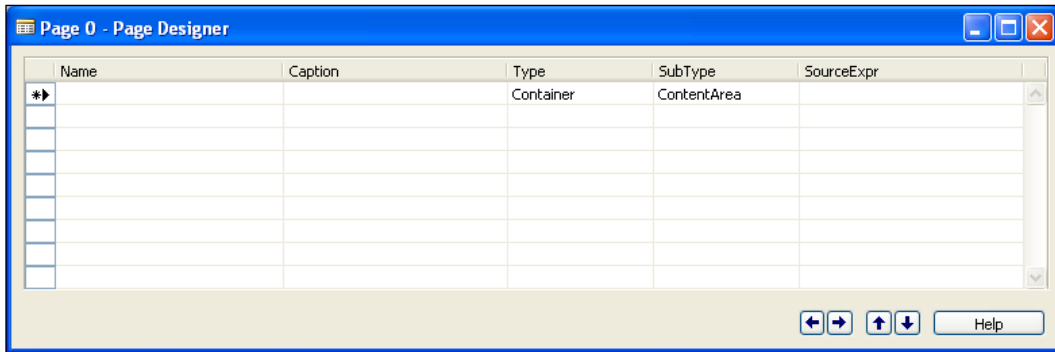
The **Table Designer** invites you to begin defining data fields. All the associated C/AL code will be embedded in the underlying triggers and developer-defined functions.

Page Designer

For **Page Designer**, the first screen is as follows:



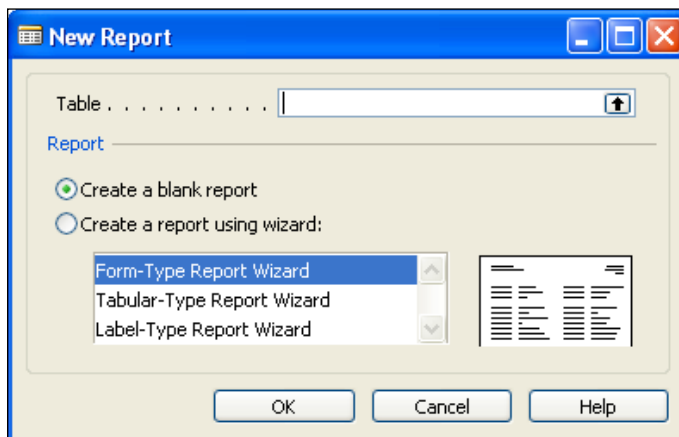
Any **New Page** effort begins with the choice of using the Wizard. If you choose not to use the Wizard and want to begin designing your page from scratch, you will select the **Create blank page** option and then will see the following screen, which is your page layout screen:



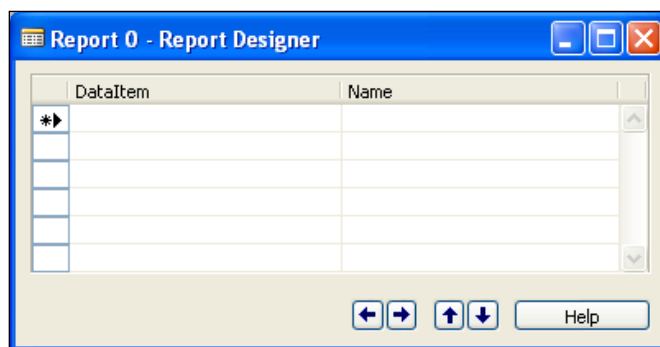
If you choose not to use the Wizard (that is choose Create), you will see a Page Designer form like the above image. If you use the Wizard, it will walk you through defining FastTabs and assigning fields to those tabs. When you finish with the Wizard, you will be dropped into the **Page Designer** with your page well on the way to completion.

Report Designer

For **New Report**, the following screen is displayed:



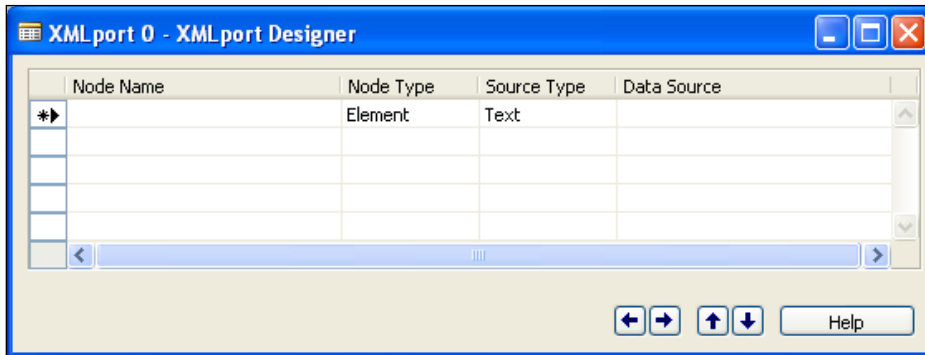
Just like with a new page, a **New Report** effort begins with the choice of using the Wizard. If you wish not to use the Wizard and want to begin designing your report from scratch, you will select the **Create a blank report** option. You'll then see the following screen where you can begin by defining the primary **DataItem** for your report:



XMLport Designer

In previous versions of NAV, Dataports were used to import or export text files and XMLports were used only for XML file imports and exports. In NAV 2009, while Dataports are still used for the Classic Client, XMLports handle the task of importing and exporting both XML files and text files.

The **XMLport Designer** is as seen in the following screenshot:



There is no Wizard for XMLports. When you click on **New**, you proceed directly to the **XMLport Designer** screen.

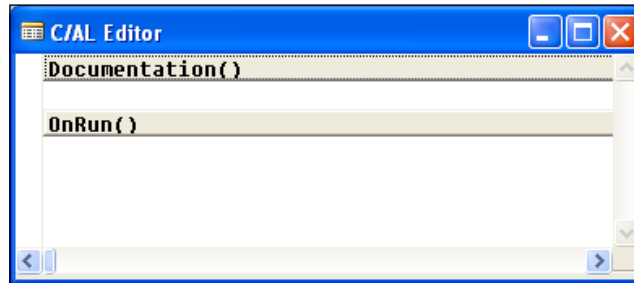
XMLports are tools for defining and processing text-based data structures, including those which are defined in XML format. XML is a set of data formatting rules for dissimilar applications to exchange data. XML-structured files have become an essential component of business data systems.

Even though XML is becoming more and more important for inter-system data exchanges, NAV applications also often must deal with data structures such as .csv files. The Classic Client uses Dataport objects to handle text file structures other than XML files. With RTC, Dataports are obsolete and we now use XMLports for all types of text files.

Once you become comfortable using C/SIDE and C/AL, we will learn more about XMLports both for XML files and for other text file formats. In the previous versions of NAV, XMLports had to be executed from other code (preferably Codeunits) as they cannot be run directly, but that is no longer true for the NAV 2009 RTC. XMLports can now be run directly from menu entries as well as from other objects. XMLport objects can also be passed as parameters to web services in a **Codeunit** function, thus supporting the easy passing of information such as a list of customers or inventory items.

Codeunit Designer

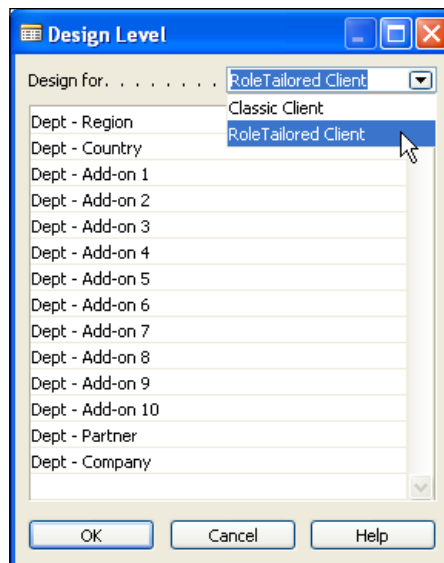
The **Codeunit Designer** is shown in the following screenshot:



Codeunits have no superstructure or surrounding framework around the single code **OnRun** trigger. Codeunits are primarily a shell in which you can place your own functions and code so that it can be called from other objects.

MenuSuite Designer

MenuSuites and XMLports were introduced in version 4.0 of NAV. The initial screen that comes up when you ask for a new MenuSuite, asks you to choose which Client type and what MenuSuite Design Level you are preparing to create. The following screenshot shows all 14 available **Design Level** values:



When one of those design levels has been used (created as a MenuSuite option), that design level will not appear in this list the next time **New** is selected for the MenuSuite Designer. MenuSuites can only exist at the 14 levels shown. Only one instance of each level is supported. Once you have chosen a level to create, NAV shifts to the MenuSuite Designer mode, which looks similar to a MenuSuite in production mode except for the heading at the top. The following screenshot shows a **navigation pane** (the MenuSuite displayed) ready for production use:



The following screenshot shows the same navigation pane in Designer mode:



As you can see, the main visible differences are the change in the heading to indicate what MenuSuite level is being designed (**Company** in this case) and the chevrons to the left of each menu bar icon.

Pressing the *Alt+F12* keys or selecting **Tools | Navigation Pane Designer** will also take you into the navigation pane (MenuSuite) Designer. The only way to exit the Navigation Pane Designer is by pressing the *Esc* key with focus on the navigation pane or by right-clicking on the **Navigation Pane Designer** heading and selecting the **Close Navigation Pane Designer** option. There are a number of basic look and feel differences between the MenuSuite Designer and the other object designers. Some of these are simply due to the ways MenuSuites are different from other NAV objects and some are undoubtedly due to design decisions made by the Microsoft developers of the tool.

We will discuss more about what you can do with MenuSuite development in a later chapter.

Some designer navigation pointers

In many places in the various designers, standard NAV data entry keyboard shortcuts apply. For example:

- *F3* to create a new empty entry.
- *F4* to delete the highlighted entry.
- *F8* to copy the entry from the same column on the preceding record.
- *F5* to access the **C/AL Symbol Menu**, which shows you a symbol table for the object on which you are working. This isn't just any old symbol table; this is a programmer's assistant. We'll dig into how this works after we learn more about C/AL.
- *F9* to access underlying C/AL code.

These last two (*F5* and *F9*) are particularly useful because sometimes the icons that you might normally use to access these areas disappear (a long-standing system idiosyncrasy). The disappearing icons are disconcerting, but only a minor problem if you remember *F5* and *F9*.

- *F11* to do an on-the-fly compile (very useful for error checking code as you write it).
- *Shift+F4* to access properties.
- *Ctrl+Alt+F1* to bring up a **Help** screen showing all the available **Function Key** actions.

- *Ctrl+X*, *Ctrl+C*, and *Ctrl+V* in normal Windows mode for deletion (or cut), copy, and paste, respectively.


You can cut, copy, and paste C/AL code, even functions, relatively freely within an object, from object to object, or to a text-friendly tool (for example, Word, or Excel text editor) much as if you were using a text editor. The source and target objects don't need to be of the same type.

- *Ctrl+F8* while highlighting any data record to zoom in on a display of the contents of all the fields in that record. This works for users and their data as well as for developers.


When you are in a list of items that cannot be modified, for example, the C/AL Symbol Menu or a Zoom display on a record, you can focus on a column, click on a letter, and jump to the next field in sequence in the column starting with that letter. This works in a number of places where search is not supported, so it acts somewhat like a search substitute. As it applies only to the first letter in the entry, this search capability is limited.

The easiest way to copy a complete object in order to create a new version is to:

1. Open the object in Design mode.
2. Click on **File | Save As** object, assign a new object number, and change the object name (no duplicate object names are allowed). A quick (mouseless) way to do a **Save As** is pressing *Alt+F* and *Alt+A* keys – continuously holding down the *Alt* key while pressing first *F* and then *A*.



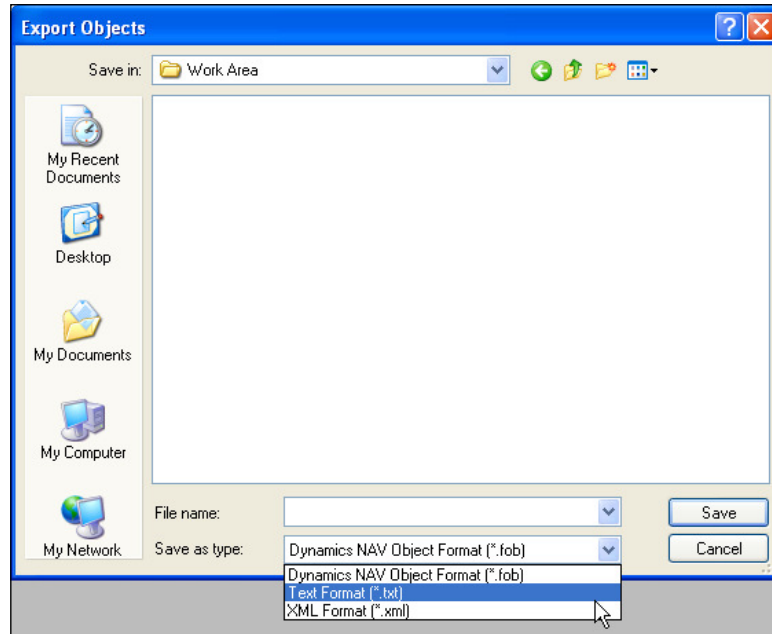
Don't ever delete an object or a field numbered in a range where your license doesn't allow creation of an object. If you don't have a compiled (for example, *fob*) back-up copy of what you've deleted available for import (that is, the entire object), you will lose the ability to replace the deleted item.



If you want to use an object or field number for a different purpose than the standard system assignment (not a good idea), make the change in place. Don't try a delete followed by add; it won't work.

Exporting objects

Object Export can be accessed for backup or distribution purposes via **File | Export**. Choosing this option, after highlighting the objects to be exported, brings up a standard Windows file-dialog screen with the file type options of *.fob* (NAV object), *.txt*, or *.xml* as shown in the following screenshot:



The safer, more general purpose format for exporting is as a compiled object, created with a file extension of `.fob`. But the alternative is available to export an object as a text file with a file extension of `.txt` or an XML structured text file with a file extension of `.xml`. The only object types whose content will export in an XML format are Forms, Reports, and Pages. An exported text file is the only way to use an external tool such as a third-party text editor to do before and after comparisons of objects or to search objects for the occurrences of strings (such as variable names). Such a file can also be used with a source-control tool such as Microsoft Visual SourceSafe.

A compiled object can be shipped to another system as a patch to install with little fear that it will be corrupted midstream. The system administrator at the other system simply has to import the new object with some directions from you. Exported compiled objects also make excellent backups. Before changing or importing any working production objects, it's always a good idea to export a copy of the "before" object images into a `.fob` file. It should also be labeled so you can easily find it for retrieval. Any number of objects can be exported into a single `.fob` file. You can later selectively import any one or several of the individual objects from that group `.fob`.



A full developer's license is required to export or import objects in the `.txt` format.

Importing objects

Object Import is accessed through **File | Import** in the **Object Designer**. The import process is more complicated than the export process because there are more possibilities and decisions that are to be made. As we've already mentioned exporting text, XML, and compiled versions of an object, your assumption that we can import all three formats as well would be correct. The difference is that when you import a compiled version of an object, the Object Designer allows decisions about importing and provides some information to help you make them.

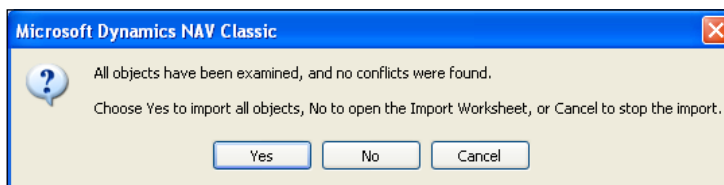


When you import a text version of an object, whether .txt or .xml, the new version is brought in regardless of what it overwrites and regardless of whether or not the incoming object can actually be compiled. The object imported from a text file is not compiled. In other words, by importing a text-formatted object, you could actually replace a perfectly good, modified production object with some trash that only had a passing resemblance to a functioning text object. It is best if text-formatted objects are never used when sending objects to an end user for installation.



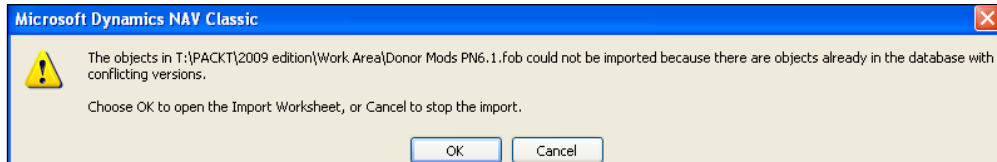
Never import a text object unless you have a current backup of all the objects that might be replaced.

When you import a compiled object from a .fob file, you will get one of two decision message screens, depending on what the Object Designer Import finds when it checks for existing objects. If there are no existing objects that the import logic identifies as matching and modified, then you will see the following dialog:



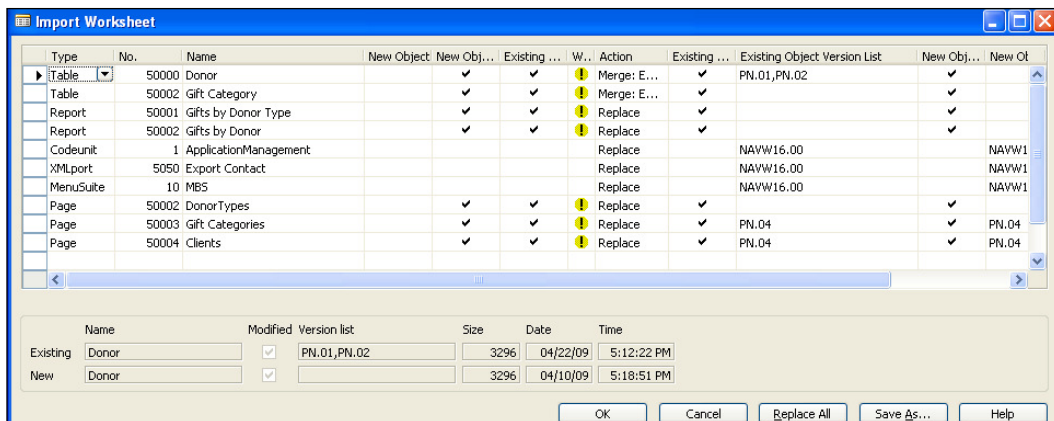
The safest thing to do is always open the **Import Worksheet** by clicking on the **No** button. Then examine what you see there before proceeding with the import.

If the .fob file you are importing is found to have objects that could be in conflict with existing objects that have been previously modified, then you will see the following dialog:

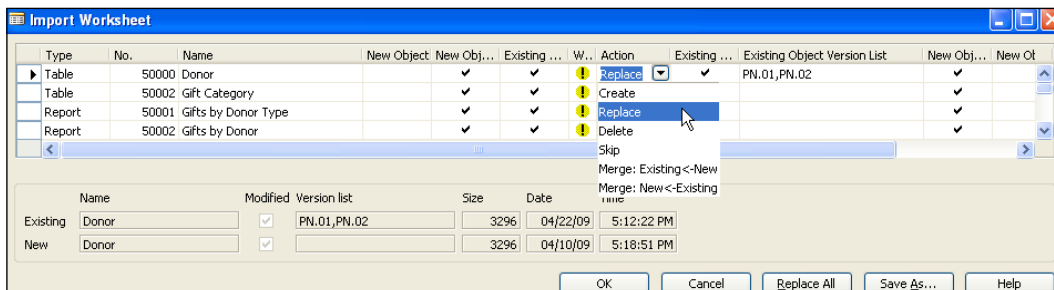


Of course you can always click on **Cancel** and simply exit the operation. Normally, you will click on **OK** to open **Import Worksheet** and examine the contents.

An example of what you might see in **Import Worksheet** is shown in the following screenshot:



While all of the information presented is useful at one time or another, usually you can focus on just a few fields. The basic question, on an object-by-object basis, is "Do I want to replace the old version of this object with the new copy?" In the case of tables, import also allows you to merge the incoming and existing table versions. Only very sophisticated developers should attempt to use this feature and then only if absolutely sure of the contents of the two table versions to be merged. The rest of us should always either choose the Import Action **Replace** or **Skip** (or **Create**, if it is a new object). This latter statement applies to all of the object types.



At the bottom of the preceding screenshot, you can see the comparison of the **Existing** object and the **New** object information. You must use this information to decide whether or not to accept the import of this object (that is, take an action of **Create** or **Replace** or action of **Skip**). More information on using the **Import Worksheet** and the meaning of various warnings and actions can be found in **C/SIDE Reference Guide Help** under **Import Worksheet**.

Text objects

A text version of an object is especially useful for a few specific development tasks. C/AL code or expressions can be placed in a number of different nooks and crannies of objects. In addition, sometimes object behavior is controlled by Properties. Consequently, it's not always easy to figure out just how an existing object is accomplishing its tasks. But an object exported to text has all its code and properties flattened out where you can use your favorite text editor to search and view. Text copies of two versions of an object can easily be compared in a text editor. Text objects can be stored and managed in a source code library. In addition, a few tasks, such as re-numbering an object, can be done more easily in the text copy than within C/SIDE.

Object number licensing

In Chapter 1, *A Short Tour through NAV 2009*, we reviewed some object numbering practices followed in NAV. The object number range for general purpose custom objects (those not part of an add-on) starts at 50000. If your client has purchased the license rights to **Table Designer**, the rights to 10 table objects are included, numbered 50000 to 50009. With **Page Designer** come the rights to either 10 or 100 page objects, numbered 50000 to 50009 (or 50099), depending on the system license level (Business Essentials or Advanced Management).

With **Report Designer** and **Dataport Designer** come the rights to 100 report objects and 100 Dataport objects, respectively, each numbered 50000 to 50099. With the **XMLport Designer** come the rights to 100 XMLport objects, numbered 50000 to 50099. With the **Application Builder** come the rights to 100 Codeunit objects, numbered 50000 to 50099. Otherwise, Codeunit objects must be licensed individually or in groups of 100. As a part of the standard system, the customer has access to the MenuSuite Designer, not to add new levels, but just to modify the company level.

If you are creating an add-on that will be distributed widely, you can apply to Microsoft for a range of object numbers that will be reserved for your add-on's objects. Some such ranges allow developers from other organizations to have modification access to your enhancement. Other ranges require a special license to view or modify your code, thus providing more security against unauthorized changes or copying.

Some useful practices

Liberally make backups of objects on which you are working. Always make a backup of the object before you start changing it. Do the same regularly during the development process. In addition to power outages and the occasional system crash, once in a while you may do something as a developer that upsets C/SIDE and it will go away without saving your work. If your project involves several developers, you may want to utilize a source control system that tracks versioning and has a check-out, check-in facility for objects.

Use *F11* to test-compile frequently. You will find errors more easily this way. Not all errors will be discovered just by compiling. Thorough testing is always a requirement.

When developing pages or reports, use the *Alt+F, R* or *Alt+F, Ctrl+R* keys to do test runs of the objects relatively frequently; whenever you reach a stage where you have a working copy, save it.

Never design a modification that places data directly in or changes data directly in a *Ledger* table without going through the standard Posting routines. It's tempting to, but doing so is an almost sure path to unhappiness. If you are creating a new *Ledger* for your application, for the sake of consistency with the NAV standard flow, design your process with a *Journal* table and a Posting process. Do the same for new *Registers*, *Posted Document* tables, and other tables normally updated during Posting.

If at all possible, try to avoid importing modifications into a production system when there are users logged-in to the system. If a logged-in user has an object active that is being modified, they will continue working with the old version until they exit and re-enter. Production use of the obsolete object version may possibly cause confusion or even more serious problems.

Always test modifications in a reasonably current copy of the production system. Do your final testing by using real data (or at least realistic data) and a copy of the customer's production license. As a rule, you should never develop or test in the live production system. Always work in a copy. Otherwise, the price of a mistake, even a simple typo, can be enormous.

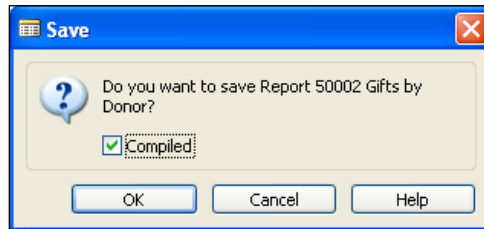
If you wish to reduce the likelihood that a change to a production system is incompatible with the rest of the system, recompile all of the objects in the system after importing changes into your test copy of the production system. You must have all referenced Automation or OCX components registered in your system for this to work well. Note that, the systems in which developers have left inoperable or obsolete "temporary" objects (that is, systems that have not had proper "housekeeping"), you may uncover serious problems this way, so be prepared.

Changing data definitions


The integration of the development environment with the application database is particularly handy when you are making changes to an application that is already in production use. C/SIDE is good about not letting you make changes that are inconsistent with existing data. For example, let's presume you have a text field that is defined as 30 characters long and there is already data in that field in the database, one instance of which is longer than 20 characters. If you attempt to change the definition of that field to 20 characters long, you will get an error message when you try to save and compile the table object. You will not be able to force the change until you adjust either the data in the database or you adjust the change so that it is compatible with all of the existing data.

Saving and compiling

Whenever you exit the Designer for an object, if you have changed anything, by default, NAV wants to save and compile the object on which you were working. You will see a dialog similar to the following screenshot:

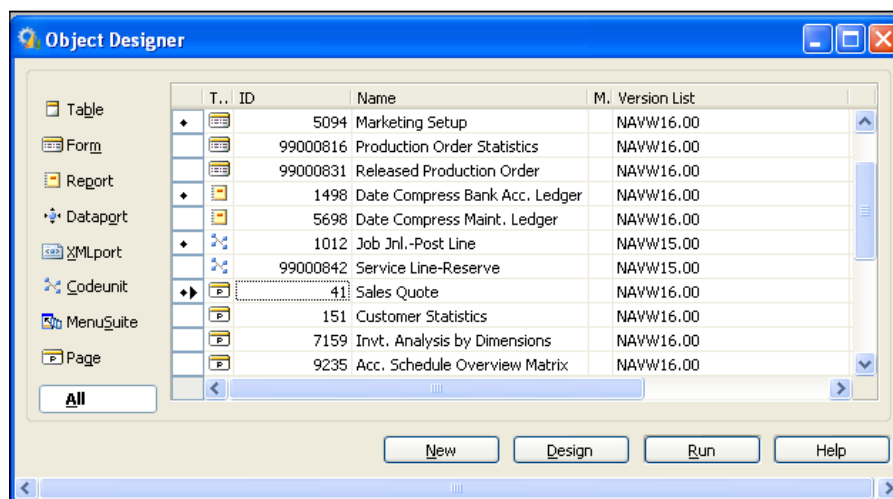


If you want to save the changed material under a new object number while retaining the original object, you must **Cancel** this option and exit the Designer by using the **File | Save As** option. If your object under development is at one of those in-between stages where it won't compile, you can deselect the **Compiled** checkbox and just save it by clicking on the **Save** button without compiling it.

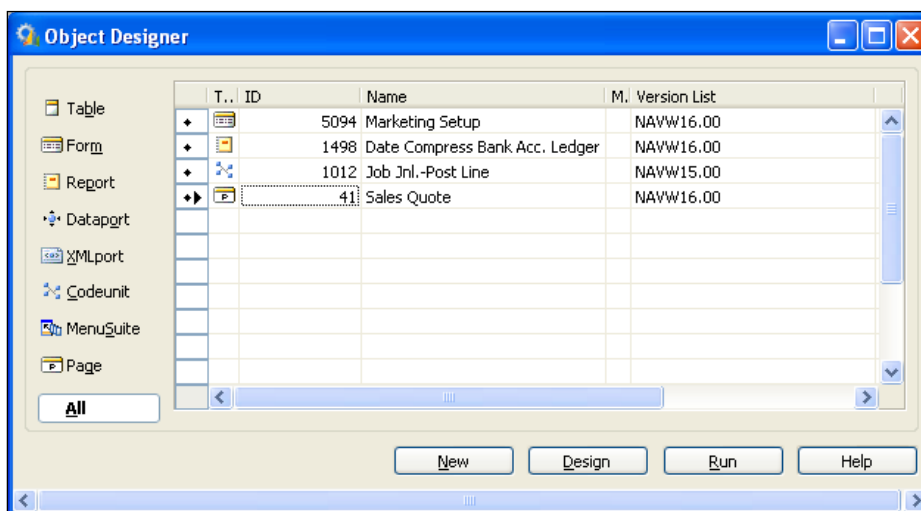
[ You should not complete a development session without getting an error-free compilation. Even if big changes are made in small increments, this will not be difficult most of the time. Exceptions should be rare.]

On occasion, you may make changes that you think will affect other objects. In that case, from the **Object Designer** screen, you can select a group of objects to be compiled. One relatively easy way to do that is to mark each of the objects to be compiled, then use the **View | Marked Only** function to select just those marked objects. That allows them to be compiled *en masse*. Marking an object is done by putting focus on the object and pressing the **Ctrl+F1** keys. The marked object is then identified with a bullet in the left screen column for that object's row.

See the four marked objects in the following screenshot:



Selecting **View | Marked Only** yields the following screenshot:



Select all the entries (using *Ctrl+A* keys is one way to do this), press *F11*, and respond **Yes** to the question **Do you want to compile the selected objects?** Once the compilation of all the selected objects is completed, you will get a message indicating how many of the objects had compilation errors. After you respond to that message, only the objects with errors will remain marked. As the **Marked Only** filter will be on, just those objects that need attention will be shown on the screen. If you had simply compiled a highlighted selection of objects (not marked), those with errors will be marked so that you can use the **Marked Only** filter to select the ones needing attention.

Some C/AL naming conventions

In the previous chapters, we discussed naming conventions for tables, pages, and reports. In general, the naming guidelines for NAV objects and C/AL encourage consistency, common sense, and intelligibility. You should use meaningful names. These make the system more intuitive to the users and more self-documenting.

When naming internal variables, try to keep the names as self-documenting as possible. Make sure you differentiate between similar, but different, values such as **Cost** (cost from the vendor) and **Amount** (selling price to the customer). Embedded spaces, periods, or other special characters should be avoided (even though you will find some violations of this in the product). If you want to use special characters for the benefit of the user, put them in the caption, not in the name. If possible, stick to letters and numbers.

There are a number of reasons to keep variable names simple. Other software products with which you may interface may have limitations on variable names. Some special characters have special meaning to other software or in another human language. In NAV, *?* and *** are wildcards and, therefore, must be avoided in variable names. *\$* has special meaning in other software. SQL Server adds its own special characters to NAV names and the resultant combinations can get quite confusing (not just to you but to the software). The same can be said for the names constructed by the internal RDLC generator.

When you are defining multiple instances of a table, either differentiate clearly by name (for example, *Item* and *NewItem*) or by a suffix number (for example, *Item1*, *Item2*, *Item3*). In the very common situation where a name is a compound combination of words, begin each abbreviated word with a capital letter (for example, *NewCustBalDue*).

Avoid creating variable names that are common words that might be reserved words (for example, Page, Column, Number, and Integer). C/SIDE will *not* warn you that you have done so and you may find your logic and the automatic logic working at very mysterious cross purposes. Do not start variables with a suffix "x", which is used in some automatically created variables (such as xRec). Make sure that you clearly differentiate between internal variable names and those originating in tables. Sometimes C/SIDE will allow you to have a global name, local name, and/or record variable name, all with the same literal name. If you do this, you are practically guaranteeing a variable misidentification bug where the compiler uses a different variable than what you intended to be referenced.

When defining a temporary table, preface the name logically, for example with Temp. In general, use meaningful names that help in identifying the type and purpose of the item being named. When naming a new function, you should be reasonably descriptive. Don't name two functions located in different objects with same name. It will be too easy to get confused later.

In short, be careful, be consistent, be clear, and use common sense.

Variables

As we've gone through examples showing various aspects of C/SIDE and C/AL, we've seen and referred to variables in a number of situations. Some of the following is obvious, but for clarity's sake we'll summarize here.

In Chapter 3, *Fields*, we reviewed various data types for variables defined within objects (referred to in Chapter 3 as *working storage data*). Working Storage consists of all the variables that are defined for use within an object, but whose contents disappear when the object closes. Working Storage data types discussed in Chapter 3 are those that can be defined in either the **C/AL Global Variables** or **C/AL Local Variables** tabs. Variables can also be defined several other places in an NAV object.

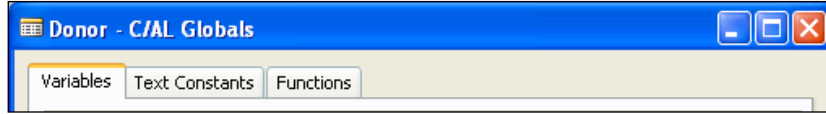
Global identifiers

Global variables are defined on the C/AL Globals form, in the **Variables** tab.

Global Text Constants are defined on the **Text Constants** tab section of the **C/AL Globals** form. The primary purpose of the Text Constants area is to allow easier translation of messages from one language to another. By putting all message text in this one place in each object, a standardized process can be defined for language translation. There is a good explanation in the **C/SIDE Reference Guide** Help on how to create Text Constants.

Global Functions are defined on the **Functions** tab of the **C/AL Globals** form.

The following screenshot shows **C/AL Globals** form:



Local identifiers

Local identifiers only exist defined within the range of a trigger. This is true whether the trigger is a developer-defined function or one of the default system triggers or standard application-supplied functions.

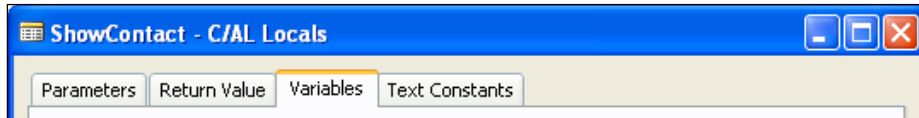
Function local identifiers

Function local identifiers are defined on one or another of the tabs on the **C/AL Locals** form that we use for defining a function.

Parameters and **Return Value** are defined on their respective tabs.

The **Variables** and **Text Constants** tabs for **C/AL Locals** are exactly similar in use to the **C/AL Globals** tabs.

The **C/AL Locals** form can be seen in the following screenshot:



Other local identifiers

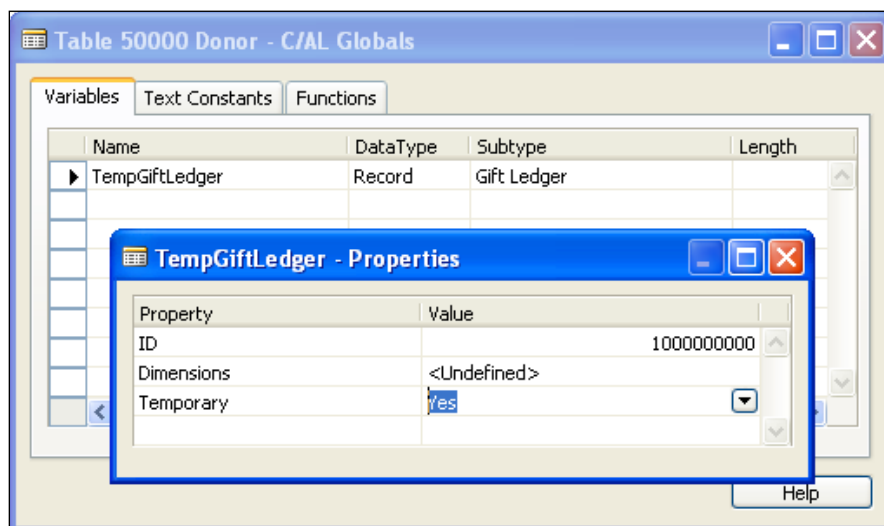
Trigger local variables are also defined on one or another of the tabs on the **C/AL Locals** form. The difference between trigger Local Variables and those for a function is that the first two tabs, **Parameters** and **Return Value**, are disabled for triggers that are not defined as functions. In a future version of NAV, these disabled tabs may disappear. The use of the **Variables** and **Text Constants** tabs are exactly the same for triggers as for functions. When you are working within a trigger, you can access the local variables through the menu option **View | C/AL Locals**.

Special working storage variables

Some working storage variables have additional attributes to be considered.

Temporary tables

Temporary tables were discussed in Chapter 2, *Tables*. Now let's take a quick look at how one is defined. Defining a Global Temporary table begins just like any other Global Variable definition of the Record data type. With an object open in the Designer, select **View | C/AL Globals**, enter a variable name, data type of **Record**, and choose the table whose definition is to be replicated for this temporary table as the **Subtype**. With focus on the new Record variable, click on the **Properties** icon (or press the *Shift+F4* keys). Set the **Temporary** property to **Yes**. That's it. You've defined a temporary table like the one in the following image.



You can use the temporary table just as though it were a permanent table with some specific differences:

- The table contains only the data you add to it during this instance of the object in which it resides.
- You cannot change any aspect of the definition of the table, except by changing the permanent table (which was its template) using the Table Designer, then recompiling the object containing the associated temporary table.

- Processing for a temporary table is done wholly in the NAV Server system in a user specific instance of the business logic. It is, therefore, inherently single user.
- A properly utilized temporary table reduces network traffic. It is often much faster than processing the same data in a permanent, database-resident table.



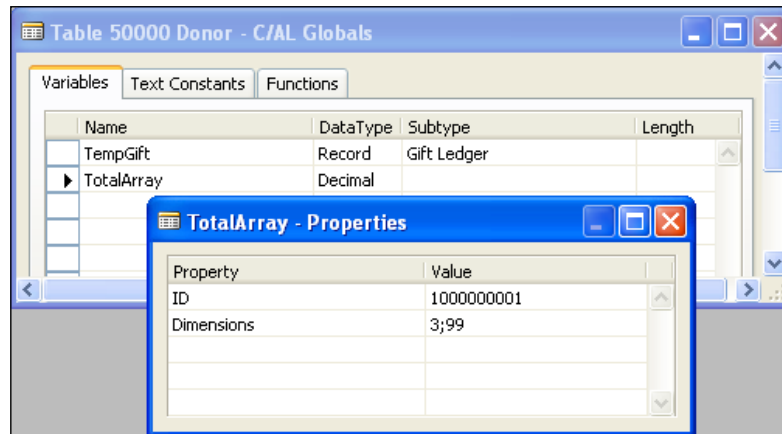
Sometimes it is a good idea to copy database table data into a temporary table for repetitive processing within an object. This can give you a significant speed advantage for a particular task. Occasionally data can be updated using this approach, then flushed back out to the database table at the end of processing.



When using temporary tables, you need to be careful that references from C/AL code in the temporary table (such as data validations) don't inappropriately modify permanent data elsewhere in the database.

Arrays

Arrays of up to 10 dimensions containing up to a total of 1,000,000 elements in a single variable can be created in an NAV object. Defining an array is done simply by setting the **Dimensions** property of a variable to something other than the default **<Undefined>**. An example is shown in the following screenshot:



The semicolon separates the dimensions of the array. The numbers indicate the maximum sizes of the dimensions. This example is a two-dimensional array which has three rows of 99 elements each. An array variable like **TotalArray** is referred to in C/AL as follows:

- The 15th entry in the first row is `TotalArray[1,15]`
- The last entry in the last row is `TotalArray[3,99]`

An array of a complex data type such as a record may behave differently than a single instance of the data type, especially when passed as a parameter to a function. In such a case, make sure the code is especially thoroughly tested so that you aren't surprised by unexpected results.

Initialization

When an object is initiated, the first time we reference a variable within the object, that variable is automatically initialized. Booleans are set to false. Numeric variables are set to zero. Text and code data types are set to the empty string. Dates are set to 0D (the undefined date) and Times are set to 0T (the undefined time). The system also automatically initializes all system-defined variables.

Of course, once the object is active, you can do whatever additional initialization you wish. And if you wish to initialize variables at intermediate points during processing, you can use any of several approaches. Initialize a Record variable (for example, the TempGift temporary table defined in the preceding example) with the `INIT` function in a statement in the form:

```
TempGift.INIT;
```

In that case, all the fields, except those in the Primary Key, are set either to their **InitValue** property value or to their data type default value. Primary Key fields must be explicitly set by C/AL code.

For other types of data, you can initialize fields with the `CLEAR` or `CLEARALL` function in a statement in the following form:

```
CLEAR (TotalArray[1,1]);
CLEAR (TotalArray);
CLEAR ("Shipment Code");
```

The first example would clear a single element of the array, the first element in the first row. As this variable is a Decimal data type, the element would be set to 0.0 when cleared. The second example would clear the entire array. In the third example, a variable defined as a Code data type would simply be set to an empty string.

System-defined variables

NAV also provides you with some variables automatically. Which variables are provided is dependent on the object in which you are operating. Descriptions of these can be found in the Help titled "System-defined Variables".


A definition of programming in C/SIDE

Many of the things that we do during development in C/SIDE might not be called programming by some because it doesn't involve writing C/AL code statements. But so long as these activities contribute to the definition of the object and affect the processing that occurs, we'll include them in our broad definition of C/SIDE programming.

These activities include setting properties at the object and Data Item levels, creating Request pages in Reports, defining Controls and their properties, defining Report Sections and their properties, creating Source Expressions, defining Functions, and, of course, writing C/AL statements in all the places that you can put C/AL. Our study will include C/SIDE programming primarily as it relates to tables, reports, and codeunits.

We will touch on C/SIDE programming for pages and XMLports. In the case of RTC reports, C/AL statements can reside only in the components that are developed within the Classic RD and not within Sections or Controls. As no coding can be done within MenuSuites, we will omit those objects from the programming part of our discussions. Because Dataports are not supported by the RTC, we will also bypass Dataports.

NAV objects are generally consistent in structure, just as you would expect. Most have some kind of properties and triggers. Pages and reports have controls, though the tools that define the controls in each are considerably different. Reports and Dataports, have very similar built-in data item looping logic. XMLports also have data item looping logic but structured differently from reports and Dataports. All the object types that we are considering can contain C/AL code in one or more places. All of these can have functions defined that can then be called either internally or externally, though good coding design says that any functions that are designed as "library" functions should be placed in a Codeunit.

 Don't forget, your fundamental coding work should focus on tables as much as possible, as tables are the foundation of the NAV system.

Functions

A function is a defined set of logic that performs a specific task. Similar to many other programming languages, C/AL includes a set of pre-written functions that are available to you to perform quite a wide variety of different tasks. The underlying logic for some of these functions is hidden, invisible, and not modifiable. These functions are supplied as part of the complete development toolset that includes the C/AL programming language. Some simple examples:

- **DATE2DMY:** Supply a date and, depending on how you call this function, it will return the integer value of the day, the month, or the year of that date.
- **STRPOS:** Supply a string variable and a string constant; the function will return the position of the first instance of that constant within the variable, or a zero if the constant is not present in the string contained in the variable.
- **GET:** Supply a value and a table, and the function will read the record in the table with a Primary Key equal to the supplied value, if one exists.
- **INSERT:** Add a record to a table.
- **MESSAGE:** Supply a string and optional variables; this function will display a message to the operator.

Such functions are the heart of the C/SIDE-C/AL tools. There are over 100 of them. On the whole, they are designed around the essential purpose of an NAV system: business and financial applications data processing. These functions are not modifiable; they operate according to their predefined rules. For development purposes, they act as language components.

A **trigger** is a combination of a defined event and a function that is performed when the event occurs. In the case of the built-in functions, NAV supplies the logic and the processing/syntax rules. In the case of triggers, as the developer, you supply the logic to enhance or even create the processing rules.

In addition to the pre-written "language component" functions, there are a large number of pre-written "application component" functions. The difference between these is that the code implementing the latter is visible and modifiable, though one should be extremely cautious about making such modifications.

An example of an application component function might be one to handle the task of processing a Customer Shipping Address to eliminate empty lines and standardize the layout based on user-defined setup parameters. This function would logically be placed in a Codeunit and thus made available to any routine that needs this capability. In fact, this function exists. It is called `SalesHeaderShipTo` and is located in the **Format Address** Codeunit. You can explore the following Codeunits for some functions you might find useful to use or from which to borrow logic. This is not an all-inclusive list, as there are many functions in other Codeunits which you may find useful in some future development project.

Some Codeunits containing functions you may find useful, either directly or as templates, are shown in the following table:

Object number	Name
1	ApplicationManagement
356	DateComprMgt
358	DateFilter-Calc
359	PeriodFormManagement
365	Format Address
397	Mail
412	Common Dialog Management
5052	AttachmentManagement
5054	WordManagement
6224	XML/COM Management

The pre-written application functions generally have been provided to address the needs of the NAV developers working at Microsoft. But you can use them too. Your challenge will be to find out that they exist and to understand how they work. There is very little documentation of these "application component" functions.

One significant aspect of these application functions is the fact that they are written in C/AL and their construction is totally exposed. In theory, they can be modified, though that is not advisable. If you decide to change one of these functions, you should make sure your change is compatible with *all* existing uses of that function.



A useful "trick" to find all the locations of use for a function (if you aren't using the better option of the *Developer's Toolkit for Microsoft Dynamics NAV*) is to add a dummy calling parameter to the function (temporarily) and then compile all objects in the system. You will get errors in all objects that call the changed function (don't forget about having all Automation and OCX functions registered before compiling and don't forget to remove the dummy calling parameter when you're done with testing). This technique works not only for Microsoft created functions, but also for functions created as part of a customization or add-on.

We can also create our own functions for any needed purpose. There are several reasons for creating new functions. The most common reason is to create a single, standardized instance of the logic to perform a specific task. Just about any time you need to use the same logic in more than one place, you should be considering creating a callable function. If you need to create a customized variation on one of NAV's existing functions, rather than change the original function, you should copy the original into your own codeunit and modify it there, as needed.

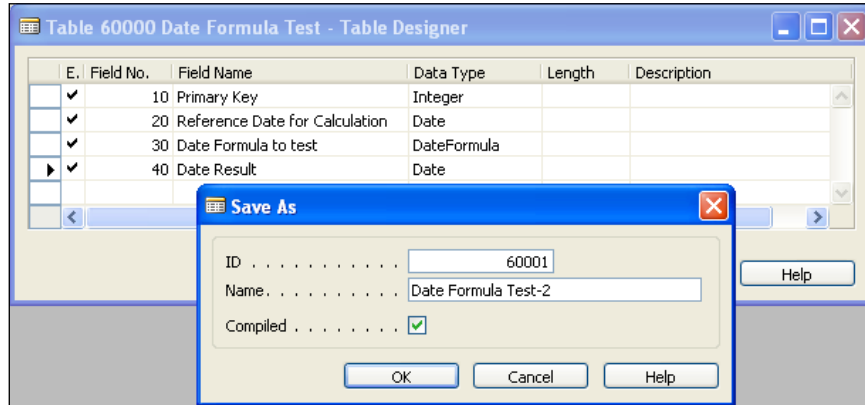
Another occasion when you should be creating functions is when you're modifying standard NAV processes. Whenever more than one line of code is needed for the modification, it may be a good idea to create the modification as a function. Thus the modification to the standard process can be limited to a call to the new function.

Though that approach is a great concept, it's often difficult to implement in practice. If, for example, you're not just adding logic but want to revise the way existing logic works, sometimes it's confusing to implement the change through just a call and an external (to the mainline process) function. Perhaps a more realistic approach is to set the threshold at a higher level such as 10 lines or 25 lines of new or changed code before creating such a function.

If a new function will be used in several objects, then it should be housed in an external (that is, modification-specific) codeunit. If it is only for use in a single object, then the new function can be resident in that object. This latter option allows direct access to the global variables within the object being modified.

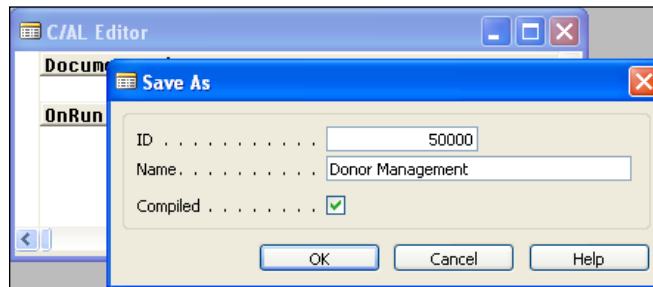
Let's take a quick look at how a function can be created. We're going to add a new codeunit to our C/AL application, Codeunit 50000. As this is where we will put any callable functions that we need for our Donor-oriented application, we will simply call it Donor Management. In that Codeunit, we're going to create a function to calculate a new date based on a given date. If that seems familiar, it's the same thing we did in Chapter 3, *Data Types and Fields for Data Storage and Processing*, to illustrate how a **DateFormula** data type works. This time, our focus is going to be on the creation of the function.

Our first step is to copy **Table 60000**, which we created for testing, and then save it as table **60001**. As a reminder, we do that by opening Table 60000 in the Table Designer, then selecting **File | Save As**, changing the object number to 60001 and the **Name** to **Date Formula Test-2** (see the following screenshot), and then exiting and compiling.



Once that's done, change the Version List to show that this table has been modified. If you used the coding of **PN** for Programming NAV and **.06** for a Chapter 6 change, then the Version List change would be to add **PN .06** to whatever was there previously.

We create our new Codeunit by simply clicking on the **Codeunit** button, then on the **New** button, and then choosing **File | Save As**, and entering the Object ID of **50000** and **Name** as **Donor Management**.



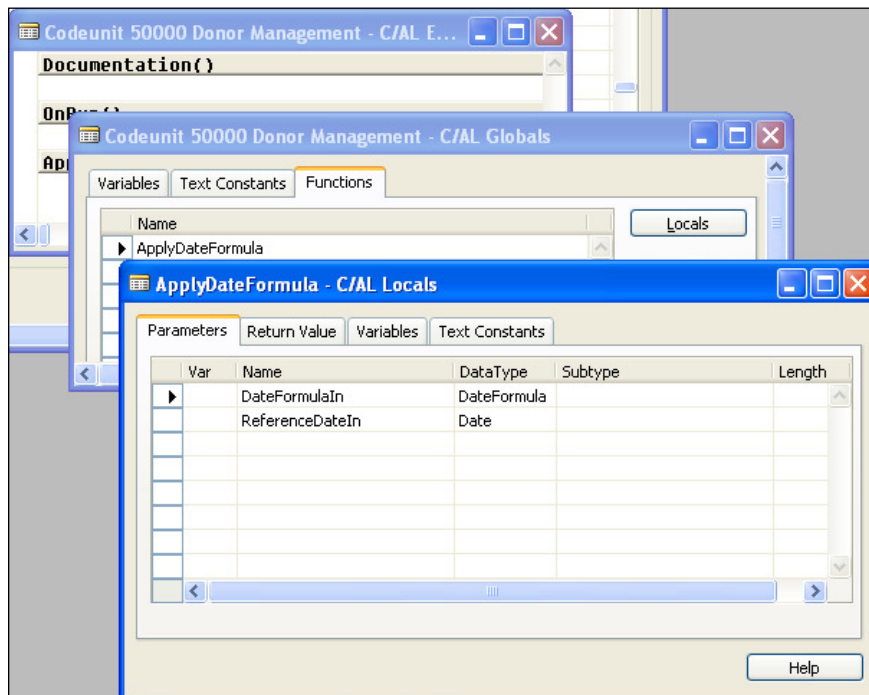
Now comes the hard part—designing our new function. When we had the function operating as a local function inside the table where it was called, we didn't have to worry much about passing data back and forth. We simply used the data fields that were already present in the table and treated them as global variables (which, in effect, they were). Now that our function will be external to the object from which it's called, we have to pass data values back and forth. Here's the basic calling structure of our function:

```
Output = Function (Input Parameter1, Input Parameter2)
```

In other words, we need to feed two values into our new callable function and accept one value back on completion of the function's processing.

Our first step is to click on **View | C/AL Globals**, then the **Functions** tab. Enter the name of the new function following the guidelines for good names, (ApplyDateFormula), and then click on the **Locals** button. This will allow us to define all the variables that will be local to the new function. The first tab on the **Locals** screen is **Parameters**, our input variables.

In keeping with good naming practice, we will define two input parameters, as shown in the following screenshot:

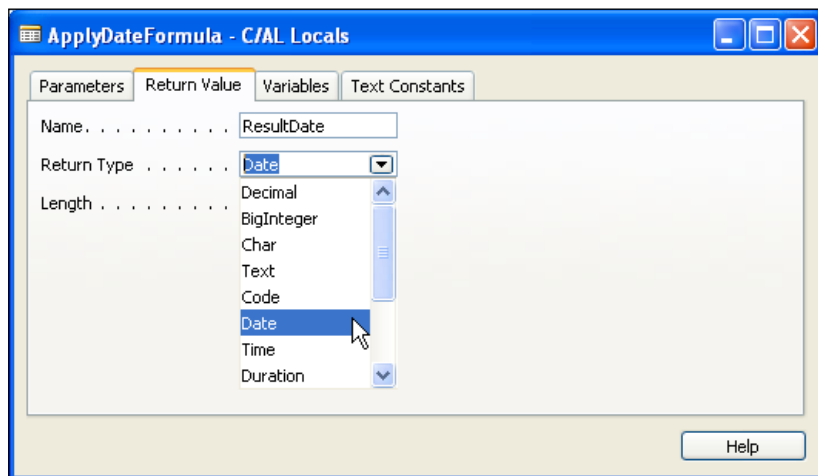




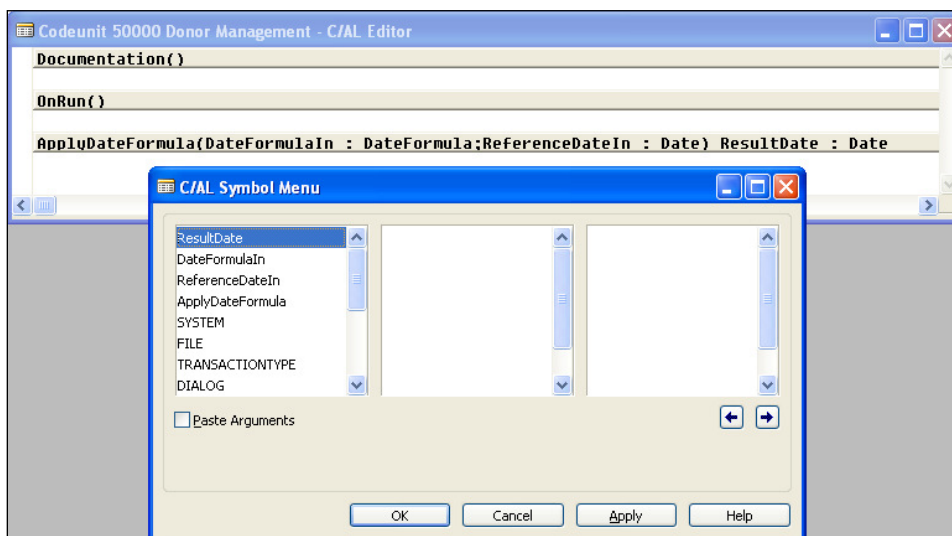
An important note regarding the **Var** column in the leftmost column of the **Parameters** tab form:

If we checkmark the **Var** column, then the parameter is passed by reference to the original calling routine's copy of that variable. That means, when the called function (in this case, the function in the Codeunit) changes the value of an input parameter, the original variable value in the calling object gets changed. As we've specified the input parameter passing here with the **Var** column unchecked, changes in the value of that input parameter will be local to the copy of the data passed in to this function and will not affect the calling routine. Checking the **Var** column on one or more parameters is a way to effectively have more than one result passed back to the calling routine. Parameter passing with the **Var** column checked is also faster, especially when passing complex data types (for example, records).

Select the **Return Value** tab and define our output variable as shown in the following screenshot:

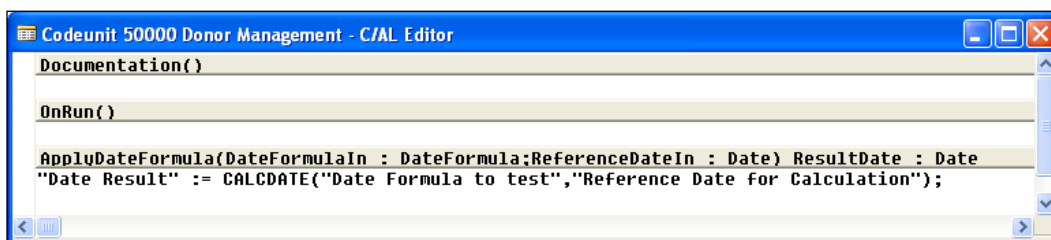


Once we complete the definition of our function and its local variables, we can exit by using the *Esc* key and the results will be saved. One way to view the effect of what we have just defined is to view the **C/AL Symbol Menu**. From the **Codeunit Designer** screen, with your new Codeunit 50000 in view and your cursor placed in the code area for our new function, click on **View | C/AL Symbol Menu** (or just press *F5*) and you will see the following image:

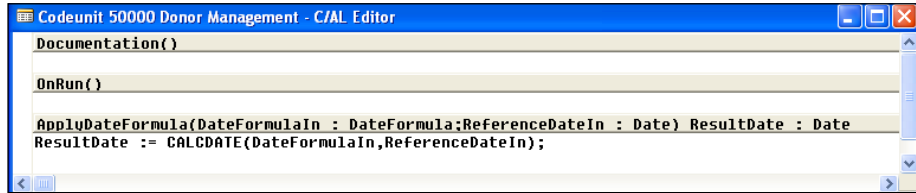


You can see that our `ApplyDateFormula` function has been defined with two parameters and a result. Now press *Esc* or select **OK**, move your cursor to the `OnRun()` trigger code area and again press *F5* to view the **C/AL Symbol Menu**. You won't see the two parameters and result. Why? Because those are local variables, which only exist in the context of the function and are not visible outside the function. We'll make more use of the C/AL Symbol Menu a little later, as it is a very valuable C/AL development tool. But right now we need to finish our new function and integrate it with our test Table 60001.

Move your cursor back to the code area for your new function. Then click on the menu item **Window | Object Designer | Table** button, then on Table 60001 | **Design**, and press *F9*. That should take you to the **C/AL Code** screen for Table 60001. Highlight and cut the code line from the local `CalculateNewDate` function. Admittedly, this will not be a particularly efficient process this time, but hopefully it will make the connection between the two instances of functions easier to envision. Using the **Window** menu, move back to our Codeunit function and paste the line of code we just cut from Table 60001. You should see the following on screen:



Edit that line of code so the variable names match those shown in our function trigger above. This should give you the following display:



Press *F11* to check to see if you have a clean compile. If you get an error, do the traditional programmer thing. Find it, fix it, recompile. Repeat until you get a clean compile. Then exit and **Save** your modified **Codeunit 50000**.

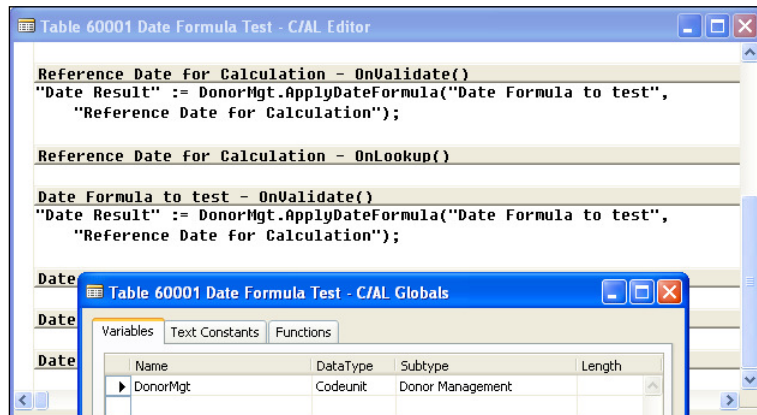
Finally, we must return to our test Table **60001**, to complete the changes necessary to use the external function rather than the internal function. The two lines of code that called the internal function `CalculateNewDate` must be changed to call the external function. The syntax for that call is:

```
Global/LocalVariable :=  
    Local/GlobalObjectName.FunctionName(Parameter1, Parameter2, ...).
```

Based on that, the new line of code should be:

```
"Date Result" := DonorMgt.ApplyDateFormula("Date Formula to test",  
    "Reference Date for Calculation");
```

Copy that line of code in place of the old function calls as shown in the following screenshot. To finish your housekeeping for this change, you should go to **View | Globals | Functions** tab and delete the now unused local function. Now go to the **Variables** tab of all functions needing to use a function from the Donor Management Codeunit and add the variable locally as shown in the following screenshot (it's always good practice to define variables as local unless global access is required):



If all has gone well, you should be able to save and compile this modified table. When that step works successfully, then **Run** the table and experiment with different Reference Dates and Date Formulas, just as you did back in Chapter 3, *Data Types and Fields for Data Storage and Processing*. You should get the same results for the same entries.

You might ask "Why couldn't I just replace the logical statement in the existing local function with a call to the new external function?" The answer is "You could". The primary reason for not doing that is the fact that the program would then be making a two-level call, possibly a less efficient procedure. On the other hand, having a single localized external call might create a structure that is easier to understand or easier to maintain. Plus this approach follows the guideline of using local variables whenever feasible. The choice is subjective. Such a decision comes down to a matter of identifying the best criteria on which to judge the available design options, then applying those criteria.

You should now have a good feeling for the basics of constructing both internal and external functions and some of the optional design features available to you for building functions.

Basic C/AL syntax

C/AL syntax is relatively simple and straightforward. The basic structure of most C/AL statements is essentially similar to what you learned with other programming languages. C/AL is modeled on Pascal and tends to use many of the same special characters in the same fashion as does Pascal. Some examples are discussed in the following sections.

Assignment and punctuation

Assignment is represented with a colon followed by an equal sign, the combination being treated as a single symbol. The evaluated value of the expression is to the right of the assignment symbol is assigned to the variable on the left side.

```
ClientRec."Phone No." := '312-555-1212';
```

All statements are terminated with a semi-colon (see the preceding line as an example) in the same manner as in Pascal. Multiple statements can be placed on a single program line, but that makes your code hard for human beings to read.

Fully qualified data fields are prefaced with the name of the record variable of which they are a part (see the preceding code line as an example where the record variable is named `ClientRec`). Essentially, the same rule applies to fully qualified function references; the function name is prefaced with the name of the object in which they are defined.

Single quotes are used to surround string literals (see the preceding code line for a phone number string).

Double quotes are used to surround an identifier (for example, variable and function name) that contains any characters other than numerals or upper and lower case letters. For example, the `Phone No.` field name in the preceding code line is constructed as "Phone No." because it contains a space and a period). Other examples would be "Post Code" (contains a space), "E-Mail" (contains a dash), and "No." (contains a period).

Parentheses are used much the same as in other languages, to indicate sets of expressions to be interpreted according to their parenthetical groupings.

Brackets [] are used to indicate the presence of subscripts for indexing of array variables. On occasion, a text string can be treated as an array of characters and you can use subscripts with the string name to access individual character positions within the string, but not beyond the terminating character at the end of the string. For example, `Address[1]` represents the first or leftmost character in the `Address` variable contents.

Statements can be continued on multiple lines without any special punctuation. As the C/AL code editor limits lines to 132 characters long, this capability is often used. The following example shows two instances that are interpreted exactly in the same manner by the compiler:

```
ClientRec."Phone No." := '312' + '-' + '555' + '-' + '1212';
ClientRec."Phone No." := '312' +
    '-' + '555' +
    '-' + '1212';
```

Wildcards

A **Wildcard** is a character that can be used in place of one or many other characters. There are three characters identified wild cards in C/AL. They are the question mark (?), the asterisk (*) and the 'at' symbol (@).

The question mark is a wildcard representing one character. If you search in NAV for a match for the string `a??ke`, you will be rewarded with results such as the following: `appke`, `aaake`, `abake`, `adeke`, `afike`, `azoke`, `a37ke`, `a%#ke`, and many more possibilities.

The asterisk is a wildcard representing a string of zero or more characters. If you search a field in NAV for the string `a*` you will get all the instances with strings starting with the letter `a`. If you search for the string `a*e`, you will get all the strings that start with the letter `a` and end with the letter `e` and have anything in between, including all the possibilities shown for our `?` search.

The 'at' symbol (`@`) is a limited functionality wild card. It is used as a modifier to a search or filter rather than a character replacement like `?` and `*`. It causes the case of the characters in the subject data to be ignored (that is, either upper or lower case matches will be successful). For example, if you search for `@aB`, then `ab`, `AB`, `aB` and `Ab` all will be included in the results.

Be very cautious when using wildcards in your code. They can lead to unexpected results, especially when data encoding rules change and, on some occasions, can cause severe performance degradation.

Expressions

Expressions in C/AL are made up of four elements: constants, variables, operators, and functions. Actually you could include a fifth element, **expressions**, because an expression may include a subordinate expression within it. As you become more experienced in coding C/AL, you will find that the capability of nesting expressions can be both a blessing and a curse, depending on the specific use and "readability" of the result.

You can create complex statements that will conditionally perform important control actions. These allow you to create a code statement that operates in much the way that a person would think about the task. You can also create complex statements that are very difficult for a human to understand. These are tough to debug and sometimes almost impossible to deal with in a modification.

One of your responsibilities over the time will be to learn to tell the difference so that you can write code that makes sense in operation, but is also easy to read and understand.

According to the **C/SIDE Reference Guide Help**, a C/AL Expression is a group of characters (data values, variables, arrays, operators, and functions) that can be evaluated with the result having an associated data type. We just looked at two code statements that accomplish the same result, namely that of assigning a literal string to a text data field. In each of these, the right side of the assignment symbol (that is, to the right of `:=`) is an expression. These statements are repeated below:

```
ClientRec."Phone No." := '312-555-1212';
ClientRec."Phone No." := '312' + '-' + '555' + '-' + '1212';
```

Operators

Now to review C/AL operators grouped by category. Depending on the data types you are using with a particular operator, you may need to know the type conversion rules (that is, what the allowed combinations of operator and data types are in an expression). The **C/SIDE Reference Guide Help** provides good information on type conversion rules. Search for the phrase **Type Conversion**.

Before we review the operators that can be categorized, let's discuss some operators that don't fit well in any of the categories. These include the following:

Other Operators	
Symbol	Evaluation
.	Member of: Fields in Records Controls in Forms Controls in Reports Functions in Objects
()	Grouping of elements
[]	Indexing
::	Scope
..	Range

- The symbol represented by a single dot or period doesn't have a given name in the NAV documentation, so we'll call it the Member symbol or Dot operator (as it is referred to in the MSDN Visual Basic Developer documentation). It indicates that a field is a member of a table (TableName.FieldName) or that a control is a member of a page (PageName.ControlName) or report (ReportName.ControlName) or that a function is a member of an object (ObjectName.FunctionName).
- Parentheses () and Brackets [] could be considered operators based on the effect their use has on the results of an expression. We discussed their use in the context of parenthetical grouping and indexing earlier. Parentheses are also used to enclose the parameters in a function call:

```
ObjectName.FunctionName (Param1, Param2, Param3) ;
```

- The Scope operator is a two character sequence " : ", two colons in a row. The Scope operator is used to allow C/AL code to refer to a specific Option value using the text descriptive value rather than the integer value that is actually stored in the database. For example, in our C/AL database Donor table, we have an Option field defined that is called Status with Option string values of Inactive and Active. Those values would be stored as integers 0 and 1, but we would like to use the strings to refer to them in code, so that our

code would be more self-documenting. The Scope operator allows us to refer to `Status::Inactive` (rather than 0) and `Status::Active` (rather than 1). These constructs are translated by the compiler to 0 and 1, respectively. If you want to type fewer characters when entering code, just enter enough of the Option string value to be unique, then let the compiler automatically fill in the rest when you next save and compile the object.

- The Range operator is a two character sequence ". . ", two dots in a row. This operator is very widely used in NAV, not only in C/AL code (including CASE statements and IN expressions), but also in filters entered by users. The English lower case alphabet can be represented by the range `a . . z`; the set of single digit numbers by the range `-9 . . 9` (that is, *minus 9 dot dot 9*); all the entries starting with the letter "a" (lower case) by `a . . a*`. Don't underestimate the power of the range operator. For more information on filtering syntax, refer to the Microsoft Dynamics Classic NAV Help's *Entering Criteria in Filters* section.

Arithmetic operators and functions

The Arithmetic operators include the following:

Arithmetic Operators		
Symbol	Action	Data Types
+	Addition	Numeric, String (concatenation), Date, Time
-	Subtraction	Numeric, Date, Time
*	Multiplication	Numeric
/	Division	Numeric
DIV	Integer Division (provides only the integer portion of the quotient of a division calculation)	Numeric
MOD	Modulus (provides only the integer remainder of a division calculation)	Numeric

As you can see in the **Data Types** column, these operators can be used on various data types. Numeric includes Integer, Decimal, Boolean, and Character data types. Text and Code are both String data.

Sample statements using DIV and MOD follow where `BigNumber` is an integer containing 200:

```
DIVIntegerValue := BigNumber DIV 60;
```

The contents of `DIVIntegerValue` after executing the preceding statement would be 3.

```
MODIntegerValue := BigNumber MOD 60;
```

The contents of `MODIntegerValue` after executing the preceding statement would be 20.

The syntax for these `DIV` and `MOD` statements is:

```
IntegerQuotient := IntegerDividend DIV IntegerDivisor;  
IntegerModulus := IntegerDividend MOD IntegerDivisor;
```

Boolean operators

Boolean operators only operate on expressions that can be evaluated as Boolean. They are as follows:

Boolean Operators	
Symbol	Evaluation
NOT	Logical NOT
AND	Logical AND
OR	Logical OR
XOR	Exclusive Logical OR

The result of an expression based on a Boolean operator will also be Boolean.

Relational operators and functions

The Relational operators are listed in the following screenshot. Each of these is used in an expression of the format:

```
Expression RelationalOperator Expression
```

For example: `(Variable1 + 97) > ((Variable2 * 14.5) / 57.332)`

Relational Operators	
Symbol	Evaluation
<	Less than
>	Greater than
<=	Less than or Equal to
>=	Greater than or Equal to
=	Equal to
<>	Not Equal to
IN	IN Valueset

We will spend a little extra time on the `IN` operator, because this can be very handy and is not documented elsewhere. The term **Valueset** in the **Evaluation** column for `IN` refers to a list of defined values. It would be reasonable to define a Valueset as a

container of a defined set of individual values, or expressions, or other Valuesets. Some examples of IN as used in the standard NAV product code are as follows:

```
GLEntry."Posting Date" IN [0D,WORKDATE]
Description[I+2] IN ['0'..'9']
"Gen. Posting Type" IN ["Gen. Posting Type"::Purchase,
                        "Gen. Posting Type"::Sale]
SearchString IN ['', '=>']
No[i] IN ['0'..'9']
"FA Posting Date" IN [01010001D..12319998D]
```

Here is another example of what IN used in an expression might look like:

```
TestString IN ['a'..'d', 'j', 'q', 'l'..'p'];
```

If the value of TestString were a or m, then this expression would evaluate to TRUE (Yes). If the value of TestString were z, then this expression would evaluate to FALSE (No).

Precedence of operators

When expressions are evaluated by the C/AL compiler, the parsing routines use a predefined precedence hierarchy to determine what operators to evaluate first, what to evaluate second, and so forth. That precedence hierarchy is provided in the C/SIDE Reference Guide, but for convenience the information, it is repeated here:

C/AL Operator Precedence Hierarchy		
Sequence	Symbols	
1	. [] () :: 	Member (Fields in Records, etc) Indexing Parenthetical Grouping Scope
2	NOT + -	Unary instances of: Logical Not Positive value Negating value
3	* / DIV MOD AND XOR	Multiplication Division Integer division Modulus Logical AND Logical Exclusive OR
4	+ - OR	Addition or Concatenation Subtraction Logical OR
5	> < >= <= <> IN	Greater than Less than Greater than or equal to Less than or equal to Not equal to IN Valueset
6	..	Range

Some basic C/AL

It's time for us to learn some more of the standard functions provided for our convenience by C/SIDE. We will focus on those most frequently found useful.

MESSAGE, ERROR, CONFIRM, and STRMENU functions

There is a group of functions in C/AL called **dialog functions**. The purpose of these functions is to allow for communications (that is, dialog) between the system and the user. There are eleven different dialog functions available. At least three of those are easy to use as tools in testing and debugging. In order to make it easier for us to proceed with our next level of C/AL development work, we're going to take time now to learn about those three dialog functions.

In each of these functions, data values can be inserted through use of a substitution string. The substitution string is the % (percent sign) character followed by the number 1 through 10, located within a message text string. That could look like the following:

```
MESSAGE('A message + a data element to display = %1',OrderAmount);
```

If the OrderAmount value was \$100.53, the output from the preceding would be:

```
A message + a data element to display = $100.53
```

You can have up to ten substitution strings in one dialog function. In all cases, the use of substitution strings and their associated display values is optional. You can also use any one of the dialog functions simply to display a completely predefined text message with nothing variable. Use of a Text Constant (accessed through **View | C/AL Globals** in the **Text Constants** tab) is recommended as it makes maintenance and multilanguage enabling easier.

MESSAGE function

MESSAGE is the most commonly used dialog function. It is easy to use for the display of transient data and can be placed almost anywhere in your C/AL code. All it requires of the user is acknowledgement that the message has been read. The disadvantage of messages is that they are not displayed until either the object completes its run or pauses for some other external action. Plus, if you should inadvertently create a situation that generates hundreds or thousands of messages, there is no graceful way to terminate their display once they begin displaying.

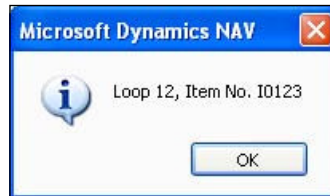
It's common to use MESSAGE as the poor man's trace tool. You can program the display of messages to only occur under particular circumstances and use them to view either the flow of processing (by outputting simple unique codes from different points in your logic) or to view the contents of particular data elements through multiple processing cycles.

MESSAGE has the following syntax: MESSAGE (String [, Value1] , ...), where there are as many ValueX entries as there are %X substitution strings (up to ten).

Here is a sample debugging message:

```
MESSAGE('Loop %1, Item No. %2',LoopCounter,"Item No.");
```

The output would look like the following (when the counter was 12 and the Item No. was I0123):



ERROR function

ERROR is formatted almost exactly like MESSAGE. Of course the function name is different and the behavior is different. When an ERROR function is executed, the execution of the current process terminates, the message is immediately displayed and the database remains unchanged as though the process calling the ERROR function had not run at all.



Sometimes you can use the ERROR function in combination with the MESSAGE function to assist in repetitive testing. MESSAGE functions can be placed in code to show what is happening with an ERROR function placed just prior to where the process would normally complete. Because the ERROR function rolls back all database changes, this technique allows you to run through multiple tests against the same data without any time-consuming backup and restoration of your test data. This isn't likely to be an original intended purpose of this function, but it turns out to be a very useful one.

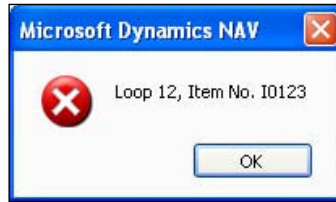
ERROR has the following syntax:

ERROR (String [, Value1] , ...) where there are as many ValueX entries as there are %X substitution strings (up to nine).

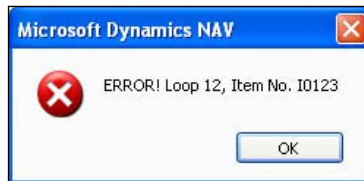
If the preceding MESSAGE was an ERROR function instead, the code line would be:

```
ERROR('Loop %1, Item No. %2',LoopCounter,"Item No.");
```

The output would look like the following screenshot:



Except for the big X in a bold red circle, you couldn't tell this was an ERROR message, although your process would terminate, which would be a clue. You could increase the ease of ERROR message recognition by including the word ERROR in your message, like the following:



Even in the best of circumstances, it is difficult for the system to communicate clearly with the users. Sometimes our tools, in their effort to be flexible, make it too easy for developers to take the easy way out and communicate poorly or not at all. In fact, an ERROR statement of the form ERROR(' ') will terminate the run and roll back all data processing without even displaying any message at all. An important part of your job, as a developer, is to ensure that your system communicates clearly and completely.

CONFIRM function

A third dialog function is the CONFIRM function. A CONFIRM function call causes processing to stop while the user responds to the dialog. In a CONFIRM, you would likely include a question in your text because the function provides **Yes** and **No** button options. The application logic can then be conditioned on the user's response.

You can also use `CONFIRM` as a debugging tool to control the path the processing will take. Display the status of data or processing flow and then allow the operator to make a choice (**Yes** or **No**) that will influence what happens next. This is exactly what `CONFIRM` is designed for in normal processing. But execution of a `CONFIRM` function will also cause any pending `MESSAGE` outputs to be displayed before the `CONFIRM` function displays. Consequently, combined with `MESSAGE` and `ERROR`, creative use of `CONFIRM` can add to your debugging/diagnostic toolkit.

`CONFIRM` has the following syntax:

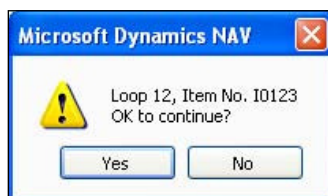
```
BooleanValue := CONFIRM(String [, Default] [, Value1] ,...)
```

where `Default` choice is `TRUE` or `FALSE` and there are as many `ValueX` entries as there are `%X` substitution strings (up to ten).

If you just code `CONFIRM(String)`, the `Default` choice will be `False`. Note that `True` and `False` appear onscreen as **Yes** and **No** (a feature that is consistent throughout NAV for C/AL Boolean values but not for RDLC report controls – see C/Side Reference Guide Help's *How to: Change the Printed Values of Boolean Variables*).

A `CONFIRM` function call with a similar content to the preceding examples might look like this for the code and the display:

```
CONFIRM('Loop %1, Item No. %2\OK to continue?',  
TRUE, LoopCounter, "Item No.");
```



In typical usage, the `CONFIRM` function is part of, or is referred to, by a conditional statement that uses the Boolean value returned by the `CONFIRM` function.

An additional feature is the use of the backslash (`\`) which forces a new line in the displayed message. This works throughout NAV screen display functions; to display a backslash, you must put two of them in your message text string, that is, `\\`.

STRMENU function

A fourth dialog function is the STRMENU function. A STRMENU function call also causes processing to stop while the user responds to the dialog. The advantage of the STRMENU function is the ability to provide several choices, rather than just two. A common use is to provide an option menu in response to the user pressing a command button.

STRMENU has the following syntax:

```
IntegerValue := STRMENU(StringVariable of Options separated by commas  
[, OptionDefault][, Instruction])
```

where the OptionDefault is an integer representing which of the options will be selected by default when the menu displays. If you do not provide an OptionDefault, the first option listed will be used as the default. Instruction is a text string which will be displayed above the list of options.

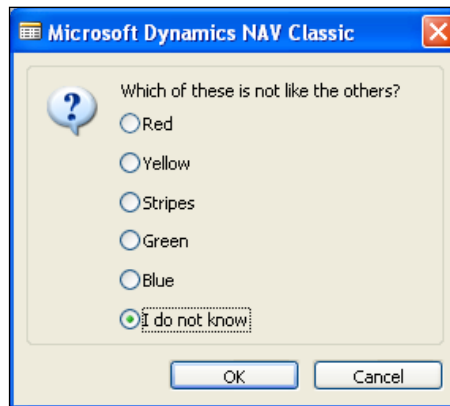
If the user responds **Cancel** or presses the *Esc* key, the value returned by the function is 0.

Use of the STRMENU function eliminates the need to use a Page object when asking the user to select from a limited set of options. The STRMENU can also be utilized from within a report or Codeunit when calling a Page would restrict processing choices.

If you phrase your instruction as a question rather than just explanation, then you can use STRMENU as a multiple choice inquiry to the user.

Here is an example of STRMENU with the instruction phrased as a question:

```
OptionNo := STRMENU('Red,Yellow,Stripes,Green,Blue,I do not know',6,  
    'Which of these is not like the others?');
```



Setting the default to 6 caused the sixth option (**I do not know**) to be the active selection when the menu is displayed.

SETCURRENTKEY function

The `SETCURRENTKEY` function behaves considerably differently when using the C/SIDE database than when using the SQL Server database. The explanation that follows focuses on the C/SIDE database behavior so that you will have more understanding of how the underlying database affects table processing.

When running on the Classic database, the `SETCURRENTKEY` function allows you to select the specific key to be used for subsequent processing, thus defining the sort order to be used. The syntax is:

```
[BooleanValue :=] Record.SETCURRENTKEY(Fieldname1,[Fieldname2], ... )
```

The `BooleanValue` is optional. If you do not specify it and no matching key is found, a runtime error will occur. This may not be a bad thing, as generally your key specification in code is fixed (not variable) and you would want to know during initial testing that you had not specified an existing key. In addition, if keys are later changed or disabled, you will want to make sure that either you have allowed for that in your error handling or that you have designed runtime error handling to identify a problem and stop processing until it is corrected.

If the key structure you specify is a partial structure, for example, only one field, and that structure matches multiple keys (that is, there are multiple keys that start with that field), C/AL may not select the key you intended. Therefore, you should provide a complete key specification.

When NAV is running on SQL Server, `SETCURRENTKEY` only determines the order in which the data will be presented to the processing, but the actual key choice for executing the query on the database is made by the SQL Server Query Analyzer. For this reason, it is very important in a SQL Server environment to make sure that the data and resources the Query Analyzer uses are well maintained. This includes maintaining the statistics that are used by the Query Analyzer to choose the best key, as well as making sure efficient key options have been defined. The indexes that are defined in SQL Server do not have to be the same as those defined in the C/AL table definition (for example, you can add additional keys in SQL Server and not in the C/AL, you can disable keys in SQL Server but leave enabled in C/AL, and so on.), but it is good practice over the long term to keep them the same. Even when the differences operate without problem, the mismatch between the application system and the underlying database system makes maintenance and upgrades more challenging.

SETRANGE function

The SETRANGE function provides the ability to set a simple range filter on a field. SETRANGE syntax is as follows:

```
Record.SETRANGE(Fieldname [,From-Value] [,To-Value]);
```

Prior to applying its range filter, the SETRANGE function removes any filters that were previously set for the defined field (filtering functions are defined in more detail in the next Chapter). If SETRANGE is executed without any From or To values, it will clear all the filters on the field. This is a common use of SETRANGE.



If SETRANGE is executed with only one value, that one value will act as both the From and To values.

Some examples of the SETRANGE function in code are as follows:

Filter to get only donors with ID from 100 through 499 or from the variable values LowVal through HiVal:

```
Donor.SETRANGE("Donor ID",100,499);  
Donor.SETRANGE("Donor ID",LowVal,HiVal);
```

Clear the filters on Donor ID:

```
Donor.SETRANGE("Donor ID");
```

Filter to allow all Donors added up through the end of a particular Campaign, that is, using the field "CAN Campaign"."End Date":

```
Donor.SETRANGE("Date Added",0D,"CAN Campaign"."End Date");
```

GET function

GET is the basic data retrieval function in C/AL. GET retrieves a single record, based on the Primary Key only. GET has the following syntax:

```
[BooleanValue :=] Record.GET ( [KeyFieldValue1]  
                                [,KeyFieldValue2] ,...)
```

The parameter(s) for the GET function are the Primary Key value (or values, if the Primary Key consists of more than one field).

Assigning the `GET` function result to `BooleanValue` is optional. If the `GET` function is not successful, that is, no record is found, and the statement is not handled by an `IF` statement, passed as a parameter or assigned to a Boolean variable, the process will terminate with a runtime error. Typically, therefore, the `GET` function is encased in an `IF` statement structured something like the following:

```
IF Customer.GET(NewCustNo) THEN ...
```



GET data retrieval is not constrained by filters. If there is a matching record in the table, GET will retrieve it.

FIND

The `FIND` family of functions is the general purpose data retrieval function in C/AL. It is much more flexible than `GET`, therefore more widely used. `GET` has the advantage of being faster as it operates only on an unfiltered direct access via the Primary Key, looking for a single uniquely keyed entry. The general purpose `FIND` function has the following syntax:

```
[BooleanValue :=] RecordName.FIND ( [Which] ).
```

The SQL Server specific members of the `FIND` family have slightly different syntaxes, as we shall see shortly.

Just as with the `GET` function, assigning the `FIND` function result to a Boolean value is optional. But in almost all of the cases, `FIND` is embedded in a condition that controls subsequent processing appropriately. Either way, it is important to structure your code to handle the instance where the `FIND` is not successful.

`FIND` differs from `GET` in several important ways:

- `FIND` operates under the limits of whatever filters are applied on the subject field.
- `FIND` presents the data in the sequence of the key which is currently selected.
- When used in the context of the Classic C/SIDE database, `FIND` uses the key which is currently selected for the actual data reads. When `FIND` is used with SQL Server, the index used for the data reading is controlled by the SQL Server Query Analyzer, unless index hinting is turned on.
- There are versions of the `FIND` function designed specifically for use with the SQL Server database. This allows C/AL coding to be optimized for SQL Server compatibility.

Following are the various forms of `FIND`:

- `FIND ('-')`: Finds the first record in a table that satisfies the defined filter and current key. Generally not an efficient option for SQL Server as it reads a set of records when many times only a single record is needed.
- `FINDFIRST`: Finds the first record in a table that satisfies the defined filter and defined key choice. Conceptually equivalent to the `FIND ('-')` but much better for SQL Server as it reads a single record, not a set of records.
- `FIND ('+')`: Finds the last record in a table that satisfies the defined filter and defined key choice. Often not an efficient option for SQL Server as it reads a set of records when many times only a single record is needed. The exception is when a table is to be processed in reverse order. Then it is appropriate to use `FIND ('+')` with SQL Server.
- `FINDLAST`: Finds the last record in a table that satisfies the defined filter and current key. Conceptually equivalent to the `FIND ('+')` but often much better for SQL Server as it reads a single record, not a set of records.
- `FINDSET`: The efficient way to read a set of records from SQL Server for sequential forward processing. `FINDSET` allows defining the standard size of the read record cache as a setup parameter, but normally defaults to reading 50 records (table rows) for the first server call. The syntax includes two optional parameters: `FINDSET ([ForUpdate] [, UpdateKey])`. The first parameter controls whether or not the read is in preparation for an update and the second parameter is `TRUE` when the first parameter is `TRUE` and the update is of key fields.

FIND ([Which]) options and the SQL Server alternates

Let's review the options of the `FIND` function using the syntax:

```
[BooleanValue :=] RecordName.FIND ( [Which] ).
```

The `[Which]` parameter allows the specification of which record is searched for relative to the defined key values. The defined key values are the set of values currently in the fields of the active key in the memory-resident record of table `RecordName`.

The following first table lists the `Which` parameter options and prerequisites. The second table lists the `FIND` options which are specific to SQL Server. In both cases, the results are always relative to the selected set (that is, they respect applied filters).

FIND "which" parameter	FIND action	Search and primary key value prerequisite before FIND
=	Match the search key values exactly	All must be specified
>	Read the next record with key values larger than the search key values	All must be specified
<	Read the next record with key values smaller than the search key values	All must be specified
>=	Read the first record found with key values equal to or larger than the search key values	All must be specified
<=	Read the next record with key values equal to or smaller than the search key values	All must be specified
-	Read the first record in the selected set. If used with SQL Server, reads a set of records	No requirement
+	Read the last record in the selected set. If used with SQL Server, reads a set of records	No requirement

SQL Server FIND option	FIND action	Search and primary key value prerequisite before FIND
FINDFIRST	Read the first record in a table based on the current key and filter. Used only for access to a single record, not in a read loop.	All must be specified
FINDLAST	Read the last record in a table based on the current key and filter. Used only for access to a single record, not in a read loop.	All must be specified
FINDSET	Read the record set specified. Syntax is Record.FINDSET ([ForUpdate] [,UpdateKey]) Set ForUpdate = True if data to be updated Set ForUpdate = True and UpdateKey = True if a key field is to be updated If no parameter specified, both default to False	All must be specified

The `FIND (' - ')` function is quite often used with the C/SIDE database as the first step in the course of reading a set of data, for example, reading all the sales invoices for a single customer. In such a case, the `NEXT` function is used to trigger all the data reads after the sequence is initiated with a `FIND (' - ')`. For a SQL Server installation, `FINDSET` should be used rather than `FIND (' - ')`. `FINDSET` only works for reading forward, not in reverse.



Chapter 11, *Optimizing for SQL Server*, of the course materials for Microsoft Course 80055, *C/SIDE Solution Development In Microsoft Dynamics® NAV* contains an extended description of the various uses of the `FINDSET` command.

The typical C/SIDE database read loop is as follows:

```
IF MyData.FIND('-') THEN
  REPEAT
    Processing logic here
  UNTIL MyData.NEXT = 0;
```

The same processing logic optimized for SQL Server is as follows:

```
IF MyData.FINDSET THEN
  REPEAT
    Processing logic here
  UNTIL MyData.NEXT = 0;
```

We will discuss the REPEAT-UNTIL control structure in more detail in the next chapter. Essentially, it does what it says; "**repeat** the following logic **until** the defined **condition** is **true**". In the case of the FIND-NEXT read loop, the NEXT function provides both the definition of how the read loop will advance through the table and provides the exiting condition.

When DataTable.NEXT = 0, that means there are no more records to be read. We have reached the end of the data, based on the filters and other conditions that apply to our reading process.

The specific syntax of the NEXT function is DataTable.NEXT(Step). DataTable is the name of the table being read. Step defines the number of records NAV will move ahead (or back) per read. The default Step is 1, meaning NAV moves ahead one record at a time, reading every record. A Step of 0 works the same as a step of 1. If the Step were 2, NAV would move ahead two records at a time and the process would only be presented with every other record.

Step can also be negative, in which case NAV moves backwards through the table. This would allow you to do a FIND('+') for the end of the table, then a NEXT(-1) to read backwards through the data. This is very useful if, for example, you need to read a table sorted by date and want to access the most recent entries first. FIND('+') is the appropriate function to use for either database when the objective is to read backwards from the end of the data set.

BEGIN-END compound statement

In C/AL, there are instances where the syntax rules only allow for use of a single statement. But your design may require the execution of several code statements.

C/AL provides at least two ways to address this need. One method is to have the single statement as a call to a function that contains multiple statements.

However, inline coding is often more efficient to run and significantly easier to understand. So C/AL provides a syntax structure to define a **Compound Statement** or **Block** of code. A compound statement containing any number of statements can be used in place of a single code statement.

A compound statement is enclosed by the reserved words `BEGIN` and `END`. The compound statement structure looks like this:

```
BEGIN
    <Statement 1>;
    <Statement 2>;
    ..
    <Statement n>;
END
```

IF-THEN-ELSE statement

`IF` is the basic conditional statement of most programming languages. It operates in C/AL similar to other languages. The basic structure is: `IF` a conditional expression is true, `THEN` execute Statement-1 `ELSE` (if condition not true) execute Statement-2. The `ELSE` portion is optional. The syntax is:

```
IF <Condition> THEN <Statement-1> [ ELSE <Statement-2> ]
```

Note that the statements within the `IF` do not have terminating semicolons unless they are contained in a `BEGIN - END` framework. As with other languages, `IF` statements can be nested so that you have conditionals dependent on the evaluation of other conditionals. Obviously one needs to take care with such constructs, as it is easy to end up with convoluted code structures that are difficult to debug and difficult for the next developer that works on this system to understand. In the next chapter, we will review the `CASE` statement which can make some complicated conditionals much easier to understand.


As you work with NAV C/AL code, you will see that often the `<Condition>` is really an expression built around a standard C/AL function. This approach is often used when the standard syntax for the function is Boolean value, function expression. Some examples are as follows:

- `IF Customer.FIND('-') THEN... ELSE...`
- `IF CONFIRM('OK to update?',TRUE) THEN... ELSE...`
- `IF TempData.INSERT THEN... ELSE...`
- `IF Customer.CALCFIELDS(Balance,Balance(LCY)) THEN...`

Indenting code

As we have just discussed BEGIN-END compound statements and IF conditional statements, which also are compound (that is, containing multiple expressions), this seems a good time to discuss indenting code.

In C/AL, the standard practice for indenting subordinate, contained, or continued lines is relatively simple. Always indent such lines by two characters except where there are left and right parentheses to be aligned.

 To indent a block of code by two characters at a time, select it and click on the *Tab* key. To remove the indentation *one* character at a time, select the code and click on *Shift+Tab*.

In the following examples, the parentheses are not required in the simplest instances, but they don't cause any problem and can make the code easier to read.

Some examples are:

```
IF (A <> B) THEN
    A := A + Count1
ELSE
    B := B + Count2;
```

Or:

```
IF (A <> B)
THEN
    A := A + Count1;
```

Or:

```
IF (A <> B)
THEN BEGIN
    A := A + Count1;
    B := A + Count2;
    IF C > (A * B) THEN
        C := A * B;
END
ELSE
    B := B + Count2;
```

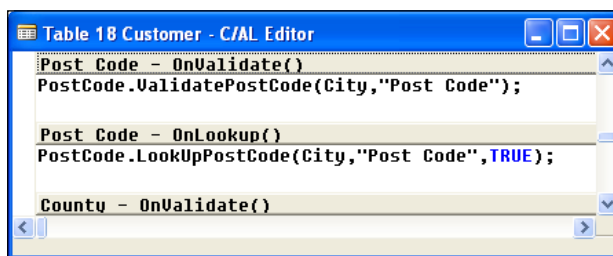
Some simple coding modifications

We're going to actually add some C/AL code to objects we've created for our ICAN application.

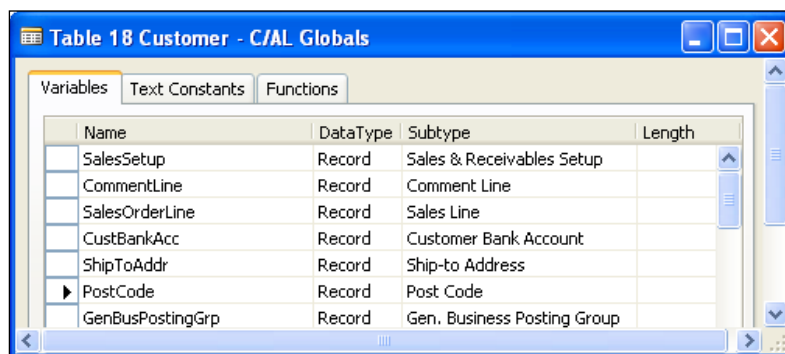
Adding a validation to a table

Let's start with some code in a **Validation** trigger of our Table 50000—Donor. When a record is added or updated, if the **Post Code** field is changed, we would like to update the appropriate address information.

As we modeled the address fields for our Donor record on the standard Customer table, let's first look at the Customer table to see how **Post Code** validation is handled there. We can access that code through the Table Designer. **Object Designer | Table | select Table 18—Customer | Design | select Field 91—Post Code | F9** is the route there. What we see in the standard code is the following:



Looking at this C/AL code, we can see that the **OnValidate** trigger (as well as the **OnLookup** trigger) contains calls to function routines in another object identified as `PostCode`. To find out what object `PostCode` actually is, we need to look in the Global working storage area.

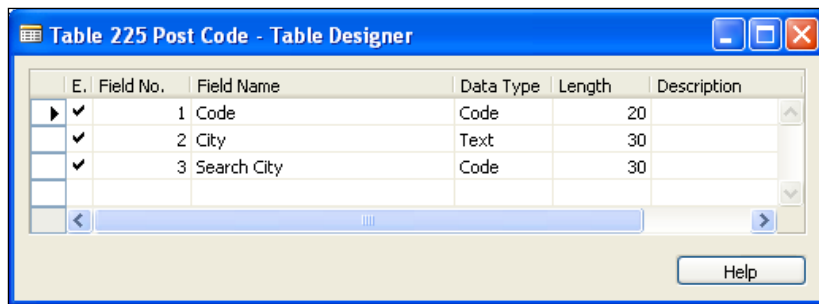


Now we see that `PostCode` is a reference to the **Record** (that is, table) **Post Code**. This is sort of like a treasure hunt at a birthday party. Now we follow that clue to the next stop, the Post Code table and the `ValidatePostCode` function that is used in the Customer Post Code validation triggers. To learn as much as we can about how this function works, how we should call it, and what information is available from the Post Code table (table 225), we will look at several things:

- The Post Code table field list
- The C/AL code for the functions in which we are interested
- The list of functions available in the Post Code table
- The calling and return parameters for the `ValidatePostCode` function

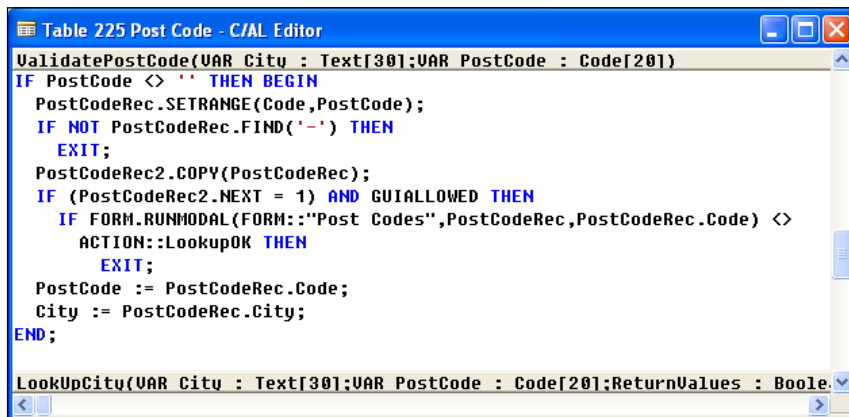
Screenshots for all those areas are as follows:

First, the field list in Table 225—Post Code:



E.	Field No.	Field Name	Data Type	Length	Description
✓	1	Code	Code	20	
✓	2	City	Text	30	
✓	3	Search City	Code	30	

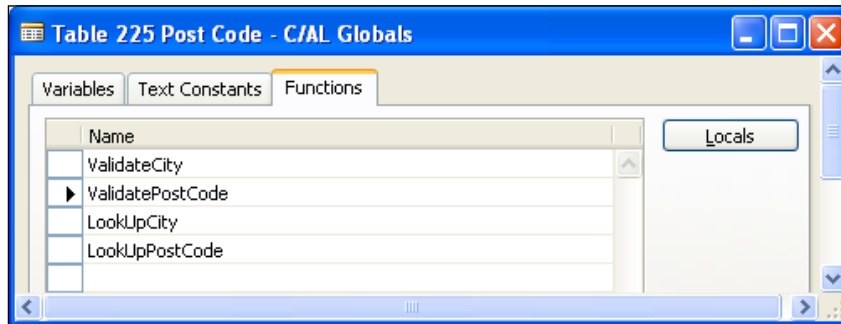
Next, the C/AL code for the **ValidatePostCode** function:



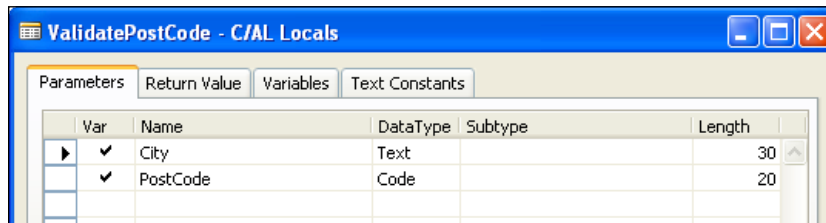
```
ValidatePostCode(VAR City : Text[30];VAR PostCode : Code[20])
IF PostCode <> '' THEN BEGIN
    PostCodeRec.SETRANGE(Code,PostCode);
    IF NOT PostCodeRec.FIND('-') THEN
        EXIT;
    PostCodeRec2.COPY(PostCodeRec);
    IF (PostCodeRec2.NEXT = 1) AND GUIALLOWED THEN
        IF FORM.RUNMODAL(FORM::"Post Codes",PostCodeRec,PostCodeRec.Code) <>
            ACTION::LookupOK THEN
            EXIT;
    PostCode := PostCodeRec.Code;
    City := PostCodeRec.City;
END;

LookUpCity(VAR City : Text[30];VAR PostCode : Code[20];ReturnValues : Boolean)
```

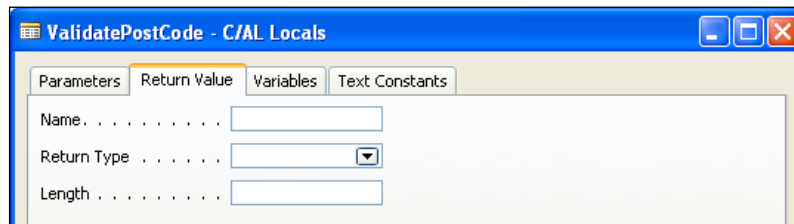
Now, the list of callable functions available within the Post Code table:



Finally, a look at the calling **Parameters** for the **ValidatePostCode** function:



Followed by this, image of the **Return Value** for the **ValidatePostCode** function:

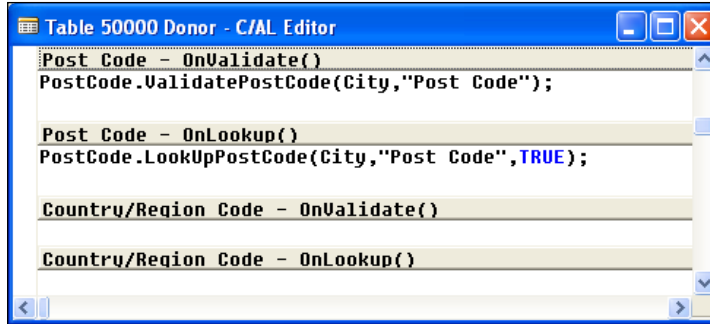


Doing some analysis of what we have uncovered, we can conclude the following:

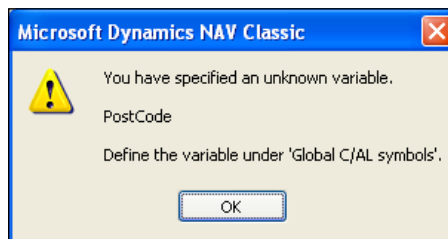
- The Post Code table in the standard product only contains the Post Code and the related City (as City and Search City). There is no field for State/Province/Region/Country. Therefore, if we wanted that data to be available as part of a Post Code validation as well, we would have to customize the Post Code table and its functions accordingly. Such a modification would generally not be a good idea due to the variety of address structures that exist across the globe.

- The **ValidatePostCode** function call uses two calling **Parameters**, one for the `City` field and one for the `Post Code` field. There is no **Return Value**. The function avoids the need for a **Return Value** by passing both the **Parameters** "by Reference" (not "by Value") as you can tell by the checkmark in the **Var** column. As a result, the function simply updates the **Parameter** `City` in the function code, which is referencing the actual data element in the object where the call originated. This interpretation is reinforced by studying the **ValidatePostCode** function C/AL code as well.

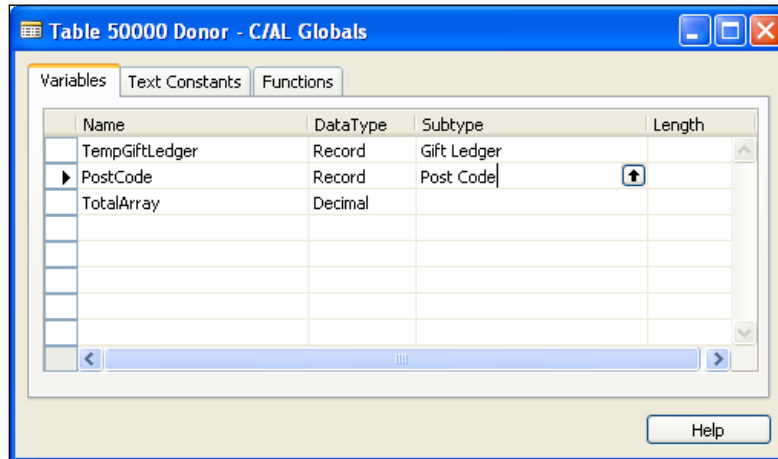
At the moment, we don't want to modify the Post Code table and functions, we conclude that we can accomplish our goal by simply copying the code from the **Post Code OnValidate** trigger in the Customer table into the equivalent trigger in our Donor table. As long we are doing that, we might as well copy the code for the **Post Code OnLookup** trigger also. The result looks like the following screenshot:



If you press *F11* at this point, you will get an error message indicating that the variable `PostCode` has not been defined.



Obviously we need to attend to this. The answer is shown in the next screenshot in the form of the `PostCode` Global Variable definition (inserted following the previous Record variable, keeping like variable types together).



After you save this change (simply moving focus from the new line of code to another line on the form or closing the form), press *F11* again. You should get no reaction other than a brief cursor blink when the object is compiled. The proper test is to run Page 50001 – Donor List, select the Action menu option to add a new Donor, enter at least the **Donor ID**, then click on the Post Code field and choose an entry from the **Post Code** table. The result should be the population of the **Post Code** field and the **City** field.

We've accomplished our goal. The way we've done it may seem disappointing to a programmer. It didn't feel like we really designed a solution or wrote any code. To some extent that's a correct evaluation. What we did was consider where else in NAV the same problem may have been addressed. We found that solution, figured out how it worked, then cloned it into our object, and AHA! We were done.

Obviously this approach doesn't work every time. But every time it does work is a small triumph of efficiency. The structure of the solution is totally consistent with the standard product, we have reused existing code constructs, and we have minimized the debugging effort and chances of production problems. In addition, our modifications are likely to work well even if the standard base application function changes in a future version.

Adding code to enhance a report

Our organization, ICAN, has decided to value Gifts of Services in order to satisfy some reporting requirements of a large foundation that we hope may provide grants for operating funds. So far, all that is tracked is the hours of service. As this is a demonstration system, not a full-scale production system, we're going to keep the design simple.

To accomplish the goal, we will do two things. One is to add logic and a reporting column to Report 50002—Gifts by Donor to show the calculated value of Gifts of Services Hours. The other is to set up a new table to contain the rate values.

To calculate a financial value, we need to have a value per hour to use in our calculation. For that purpose (and others that may arise), we will create a new ICAN Rate table and List page for data maintenance. The only data in our new table at this point will be a rate per hour for Gifts of Services, but we will design our code to allow for future expansion of the table's usage. Obviously a more sophisticated solution might value a gift of lawn mowing at a different hourly rate than a gift of computer repair services (for example).

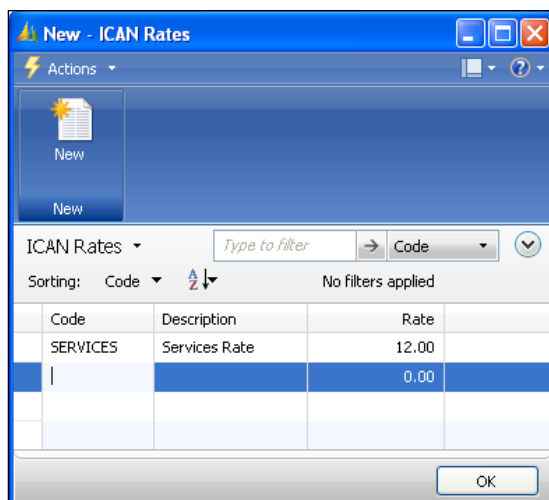
The ICAN Rate table will be a simple table with a structure of fields consisting of **Code** (the Primary Key), **Description**, and **Rate**. The resulting table definition looks like the following:

	E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>		10	Code	Code	10	
<input checked="" type="checkbox"/>		20	Description	Text	30	
<input checked="" type="checkbox"/>		30	Rate	Decimal		

Next, we need to build a List page to view and maintain the new ICAN Rate table. Since we've been through this process before (Chapter 4, *Pages – Tools for Data Display*), we'll just show the result in the Page Designer.

Name	Caption	Type	SubType	SourceExpr
<Control1000000000>	ICAN Rates	Container	ContentArea	
<Control1000000001>	<Control1000000001>	Group	Repeater	
<Code>	<Code>	Field		"Code"
<Description>	<Description>	Field		"Description"
<Rate>	<Rate>	Field		"Rate"

Once we save, compile, and run this Page, the List for the ICAN Rate table will display. Initially, there will be no data in the table, so the first action is to add a record with a Code of SERVICES. Once that is done, the ICAN Rate List should look similar to the following screenshot (especially if your rate happens to be 12.00 per hour).

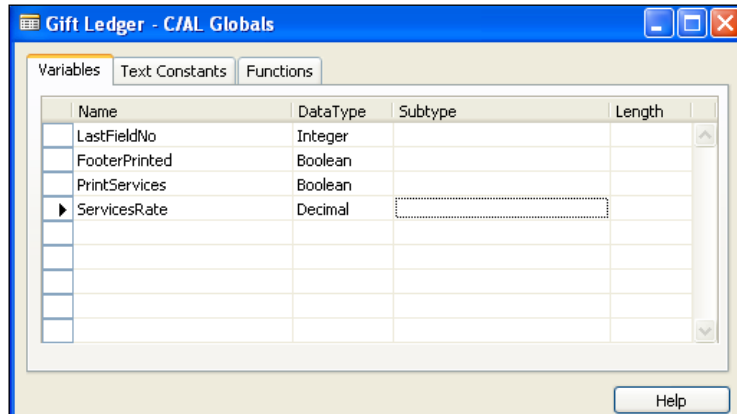


Now that we have the preparatory work done, we can make the report changes that are the primary goal for this modification. Working with Report 50002—Gifts by Donor, we need to do the following:

- Add access to the new ICAN Rate table
- Read the correct record in ICAN Rate table
- Add logic to calculate the value of Services (applying this only to SERVICES Gifts)

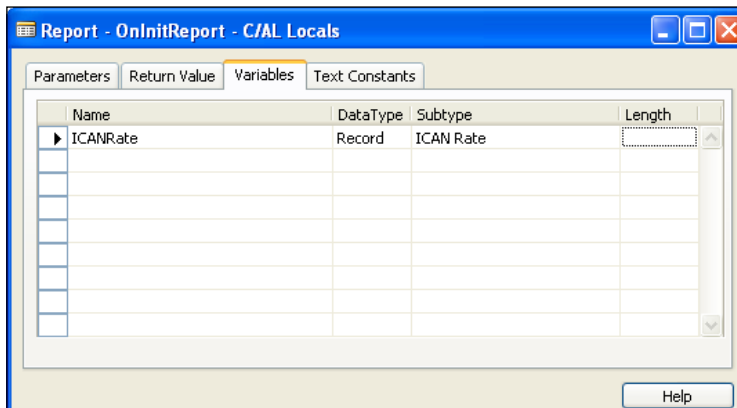
Depending on how we want to report the results, we could add a new column to the report for the Est. Value of Services or we could print the new calculated value in the same column as now contains the Est. Value for non-services gifts. As we've already had the experience of adding a column to this report in Chapter 5, *Reports*, we will start with the easier approach of using the existing column.

We begin by adding a Decimal Global Variable to the Report for the Services Rate value. Note that the other Global Variables already there were created by the Report Wizard when we originally generated this report.



Now let's add a Local Variable to the report **OnInitReport** trigger to allow access to the ICAN Rate table. We use a Local Variable for a couple of good reasons. First, as a general rule, code should use local variables whenever possible. Second, in the new NAV 2009 environment, it's more efficient for a record variable to be local (less internal processing is required that way). We use the **OnInitReport** trigger because it will allow reading the ICAN Rate table to occur at the earliest possible point after the object is instantiated.

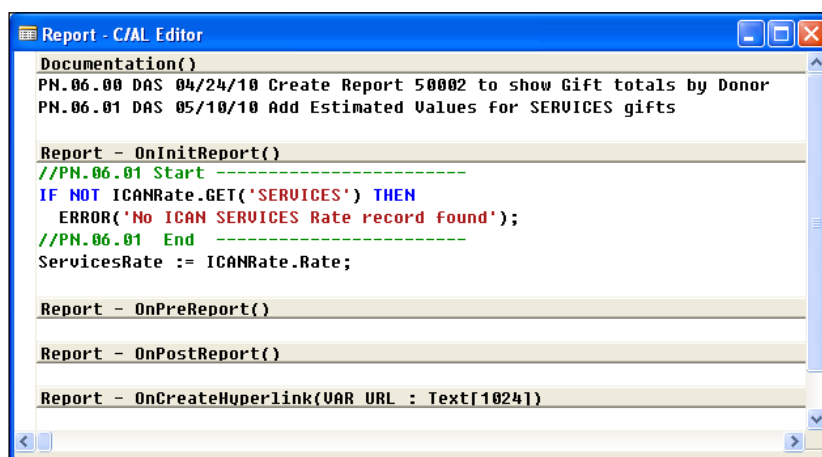
Return to the **Report Designer DataItem** definition screen. Focus on the first blank line and access the **C/AL Editor** (by pressing *F9*) showing the report triggers. Focus on the **OnInitReport** trigger and click on **View | C/AL Locals**. Now click on the **Variables** tab and define the local variable for **ICANRate** with a result like that in the next image:



The next step is to create the logic to read the record from the ICAN Rate table. As that read only needs to happen once (only one record has a rate for SERVICES), it should be done at the beginning of the report, in a trigger that is only executed once. That's why we are using the **OnInitReport** trigger. Because we are using a SQL Server database, we want to use an efficient SQL Server method to read that record. With only one record, obviously any of the **FIND** options would work, but if we want to use a **FIND** option, we should use **FINDFIRST**. However, as the simple table design will force there to be only one record with a Code of SERVICES, we can use the **GET** function.

A successful read is followed by storing the Rate data in the ServicesRate global variable. That code is shown in the next screenshot. The code shown embeds the **GET** function in an error checking **IF** statement. If you don't want to generate your own error message, you can simply use the following code and let the system supply the error message if no SERVICES record is found:

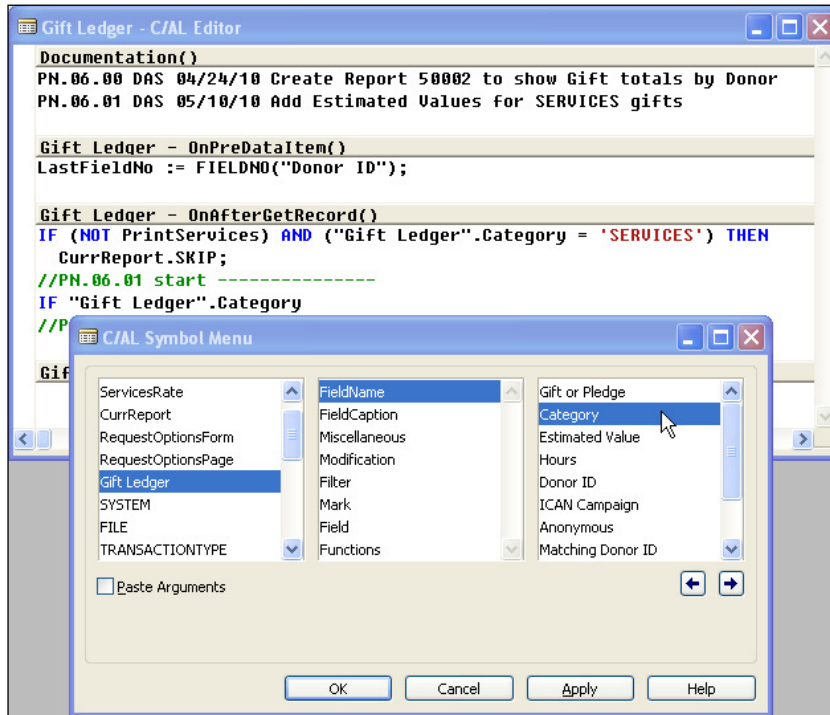
```
ICANRate.GET('SERVICES');
```



The last development work we need to do is to code the logic that will calculate the value of Services for all such records. As we are not updating the data table, we can use the internal record as temporary storage for the results of our calculation. We will use the same field that holds value for material gifts. Pseudo-logic for the code is:

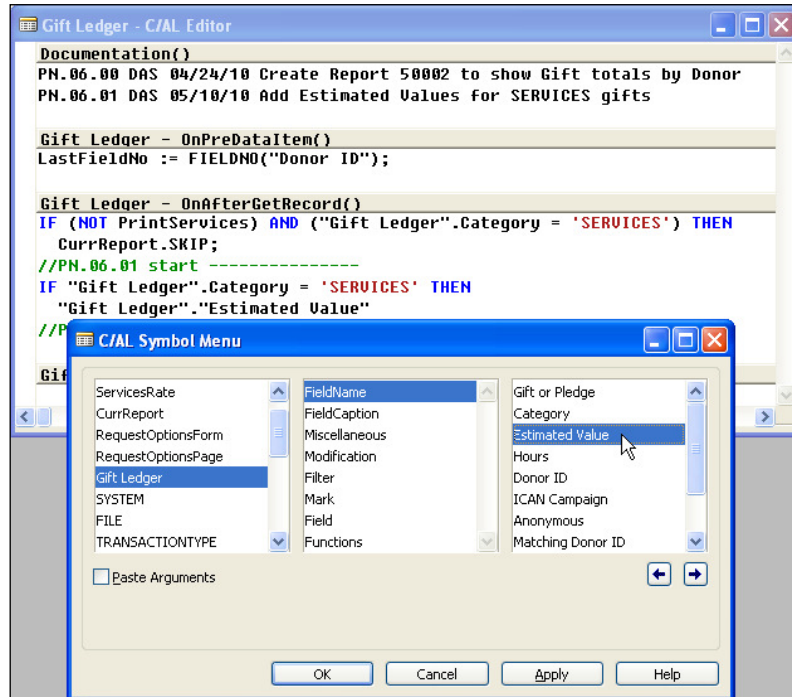
```
IF this is a SERVICES record, then calculate Estimated Value = Hours *
Rate.
```

That should be simple, a single `IF` statement. We can take advantage of tools built into C/SIDE to make our C/AL coding easier. For each data variable that we need in our code, we can look up the variable in the C/AL Symbol Menu (accessed through the *F5* key), double-click on the variable, and it will be pasted into our C/AL code line at the point where the cursor is. The first example for this statement is illustrated in the following image where the variable "Gift Ledger".Category has just been pasted into our code line:

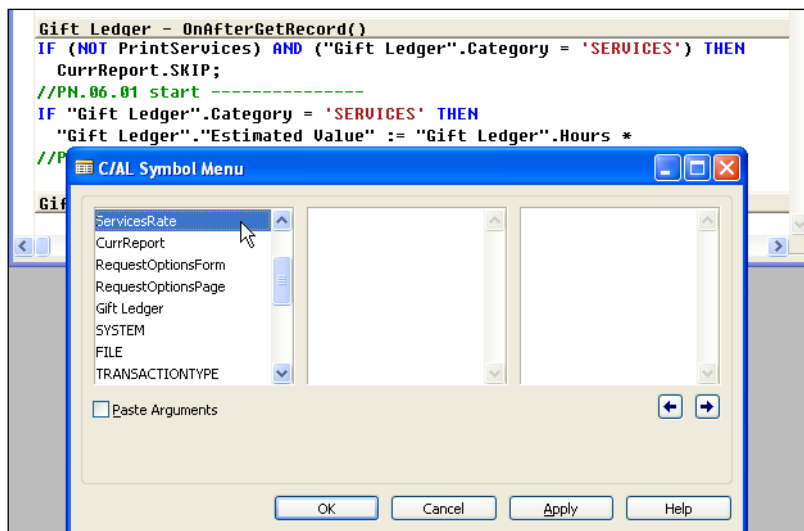


We are only computing estimated values for Services records, so that the `IF` statement checks for Category equal to 'SERVICES'.

We use the same technique as we did a moment ago to paste in the target field for our calculation, "Gift Ledger"."Estimated Value".



Next, we enter the assignment operator ($:=$) and paste in the "Gift Ledger".Hours field followed by the $*$ operator. Finally, we go back to the C/AL Symbol Menu to help us finish our calculation statement, accessing the **ServicesRate** field from the global variables.



The final code looks like the following:

```
Gift Ledger - OnAfterGetRecord()
IF (NOT PrintServices) AND ("Gift Ledger".Category = 'SERVICES') THEN
    CurrReport.SKIP;
//PN.06.01 start -----
IF "Gift Ledger".Category = 'SERVICES' THEN
    "Gift Ledger"."Estimated Value" := "Gift Ledger".Hours * ServicesRate;
//PN.06.01 end -----
Gift Ledger - OnPostDataItem()
```

At this point, we might notice that many of the data fields are referencing "Gift Ledger" fields and, as Gift Ledger is the default record reference in this Gift Ledger—OnAfterGetRecord() trigger, we don't have to have those explicit qualifiers. They don't hurt anything, but they may make the code harder for you to read (this is a subjective judgment). If you remove those qualifiers, the final code will look like the following (and work exactly the same as the code with qualifiers).

```
Gift Ledger - OnAfterGetRecord()
IF (NOT PrintServices) AND (Category = 'SERVICES') THEN
    CurrReport.SKIP;
//PN.06.01 start -----
IF Category = 'SERVICES' THEN
    "Estimated Value" := Hours * ServicesRate;
//PN.06.01 end -----
Gift Ledger - OnPostDataItem()
```

Now that we've completed our initial set of report modifications, it's time to test. Exit and save, compiling the report. Then use the **Run** command to test. Remember to select the option to print Services entries.

After you have a working report at this level, you might want to do some more advanced work on your own. Perhaps the Services values should be in a separate column? Maybe you should add a report option that prints only Services gifts, just as you now can print everything except Services gifts? Looking at the report which shows value of services but not the rate or the hours on which that value is based, perhaps those pieces of information should also be printed?

If you want a good test of what you have learned about Reports to this point, all those changes would be good ones with which to experiment. If you are feeling particularly brave, you might want to add a Services Rate field to the Category table rather than the Rate table, so that there could be multiple kinds of Services with different rates. But if you do that, then you may need to redesign the places where we have used the literal 'SERVICES' to select particular data in or out of processing. It's interesting how quickly a few small changes can expand the scope of work significantly. Nevertheless, making some additional modifications to allow for multiple rates for gifts of Services would be good practice.

In almost every modification, there are a number of different ways to accomplish essentially the same result. Some of those paths would be significantly different to the developer, but nearly indistinguishable to the user. Some might not even matter to the next developer who has to work on this report.

What is important is that the result works reliably, provides the desired output, operates with reasonable speed, and does not cost too much to create or maintain. If all those goals are met, most of the other differences are generally not particularly important.

Summary

Thought is the blossom; language the bud; action the fruit behind it
— Ralph Waldo Emerson

In this chapter, we covered topics including Object Designer navigation, as well as more specific navigation of individual Object (Table, Page, Report, and so on) Designers.

We covered a number of C/AL language areas in relative detail including functions and how they may be used, variables of various types (both development and system), basic C/AL syntax, and discussion of C/AL expressions and operators. Some of the essential C/AL functions that we covered included dialogs for communication with the user, SETRANGE filtering, GET and FIND, and related functions, BEGIN-END for code structures, plus IF-THEN for basic process flow control.

Finally, we got some hands-on experience by adding validation code to a table and adding code to enhance a generated report.

In the next chapter, we will expand our exploration and practice in the use of the tools of C/AL.

Review questions

1. All NAV objects can contain C/AL code. True or False?
2. All NAV object development work starts from the Object Designer. True or False?
3. Choose two object types that have Wizards to "jump start" their development:
 - a. Pages
 - b. XMLports
 - c. Tables
 - d. Reports
4. If an object type has a Wizard, you must start with the Wizard before proceeding to the object Designer form. True or False?
5. Objects can be exported in several formats. Choose three:
 - a. fob
 - b. text
 - c. .NET
 - d. XML
 - e. gif
6. Which object export format should be used to transmit updates to client sites?
 - a. fob
 - b. text
 - c. .NET
7. Object numbers and names are so flexible that you can (and should) choose your own approach to numbering and naming. True or False?
8. Whenever possible, the controlling logic for managing data should be resident within the tables. True or False?

9. One setting defines how parameters are passed to functions, whether a parameter is passed by reference or by value. Choose that setting identity:
 - a. DataType
 - b. Subtype
 - c. Var
 - d. Value
10. When an `ERROR` statement is executed, the user is given the choice to terminate processing, causing rollback, or to ignore the error and continue processing. True or False?
11. Choice of the proper version of the `FIND` statement can make a significant difference in processing speed. True or False?
12. When a function is needed for a customization, it is always better to create your own within your custom code than to call an existing function that exists in standard code. True or False?

7

Intermediate C/AL

If you always do what you always did, you always get what you always got – Anon
You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program – Alan Perlis

In the last chapter, we learned enough C/AL to create a basic operational set of code. In this chapter, we will learn more about C/AL functions and pick up a few good habits along the way. The C/AL functions represent a significant portion of knowledge that you will need on a day-to-day basis, as you are getting started as a professional NAV 2009 Developer.

Our goal is to understand more complex C/AL statement types, to be able to competently manage I/O, to create moderately complex program logic structures, and to understand data filtering and sorting as handled in NAV and C/AL. Since the functions and features in C/AL are designed for business and financial applications, you can do a surprising amount of work with a relatively small number of language constructs.

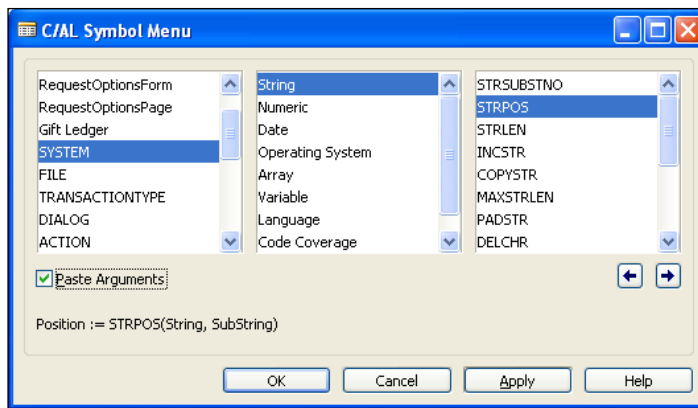
Keep in mind that anything discussed in this chapter will not relate directly to those portions of NAV objects which contain no C/AL (for example, MenuSuites and Visual Studio Report Designer report layouts).

Some C/AL development tools

All NAV development is done in C/AL and all C/AL development is done in C/SIDE. As an Integrated Development Environment, C/SIDE contains a number of tools designed to make our C/AL development effort easier. Among these is the C/AL Symbol Menu which we had utilized in our report modifications at the end of the previous chapter.

C/AL Symbol Menu

When you are in one of the Object Designers, the **C/AL Symbol Menu** is accessed from either the menu option **View | C/AL Symbol Menu** or by pressing *F5*. The default three-column display has variables and function categories in the left column. If the entry in the left column is a system function or a variable of function type, then the center column contains subcategories for the highlighted left-column entry. The right column contains the set of functions that are a part of the highlighted center-column entry. In a few cases (such as BLOB fields), there is additional information displayed in the columns further to the right. These columns are accessed through the arrows displayed just below the rightmost display column, as shown in the following screenshot:



The **C/AL Symbol Menu** is a very useful multi-purpose tool for the developer. You can use it as a quick reference to see what C/AL functions are available to you, to access **Help** on those functions, and to view what other systems would refer to as the Symbol Table. You can also use the **C/AL Symbol Menu** to paste variable names or mini code structures into your code.

The reference use is very helpful when you are starting as a C/AL developer. It is a guide to the inventory of available code tools with some especially handy features.

The first one is the general syntax for the highlighted function shown at the bottom left of the screen, as shown in the previous screenshot. You have also quick and focused access to **C/SIDE Reference Guide Help**. When you focus on an entry in the right (third) column and press *F1*, you will be taken directly to the **Help** for that function. If focus is in the left or centre column, pressing *F1* may just bring up the general **C/SIDE Reference Guide Help** rather than a specific entry.

The second use of the C/AL Symbol Menu is as a symbol table. The symbol table for your object is visible in the left column of the **C/AL Symbol Menu** display. The displayed symbol set (that is, variable set) is context sensitive. It will include

all system-defined symbols, all your **Global** symbols, and **Local** symbols from the function that had focus at the time you had accessed the C/AL Symbol Menu. Though it would be useful, there is no way within the Symbol Menu to see all Local variables in one view. The Local symbols will be at the top of the list, but you have to know the name of the first Global symbol to determine the scope of a particular variable (that is, if it appears in the symbol list before the first Global, it's a Local variable, otherwise it's Global).

The third use for the C/AL Symbol Menu is as a code template with a paste function enabled. This function will be enabled if you have accessed the **C/AL Symbol Menu** from the **C/AL Editor**. Paste is initiated by pressing either the **Apply** button or the **OK** button. In both the cases, the element with focus will be pasted into your code. **Apply** will leave the Symbol Menu open and **OK** will close it (*double-clicking* on the element has the same effect as clicking on **OK**).

If the element with focus is a simple variable, then that variable will be pasted into your code. If the element is a function whose syntax appears at the lower left of the screen, the result of the paste action (that is, **Apply** or **OK** or *double-click*) depends on whether or not **Paste Arguments** (just below the leftmost column) is checked or not. If the **Paste Arguments** checkbox is not selected, then only the function itself will be pasted into your code. If the **Paste Arguments** checkbox is selected (as shown in the preceding screenshot), then the complete syntax string, as shown, will be pasted into your code. This can be a very convenient way to create a template to help you enter the correct parameters with the correct syntactical punctuation more quickly.

When you are in the C/AL Symbol Menu, you can focus on a column, click on a letter and jump to the next field in sequence in the column starting with that letter. This acts as a limited Search substitute, sort of an assisted browse.

Internal documentation

When you are creating or modifying software, you should always document what you have done. It is often difficult for developers to spend much time (that is, money) on documentation because most of them never enjoy doing it and the benefits are uncertain. A reasonable goal is to provide enough documentation so that a smart person following you can understand the reasons for what you have done. If you choose good variable names, the C/AL code will tend to be self-documenting. If you lay out your code neatly, by using indentation consistently and localizing logical elements in functions, then the flow of your code should be easy to read. Nevertheless, you should add comments to describe the functional reason for the change. It doesn't matter how easy it is to follow the logic if you don't know the business reason for what's being done.

In case of a brand-new function, a simple statement of purpose is all that is necessary. In case of a modification, it is extremely useful to have comments providing a functional definition of what the change is intended to accomplish, as well a description of what has been changed. If there is good external documentation of the change, the comments in the code should ideally refer back to this external documentation. In any case, the primary focus should be on the functional reason for the change, not just the technical reason. Any good programmer can study the code and understand what changed, but without the documentation describing why the change was made, the next person's task will be made much more difficult.

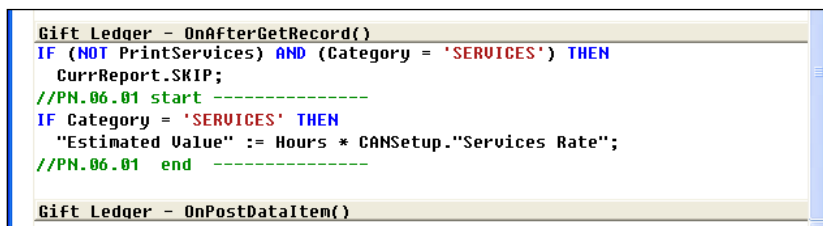
In the following example, the documentation is for a brand-new report. The comments are in the **Documentation** section, where there are no format rules, except for those you impose. This is a new report, which we created in a previous chapter. The comment is coded to indicate the organization making the change (we are crediting our book topic as "Programming NAV") and a sequence number for this change. In this case, we are using a two digit number (06) for the change, plus the version number of the change, 00, hence **PN.06.00**, followed by the initials of the developer (**DAS**) and the date of the change as shown in the following screenshot.

You can make up your own standard format that will identify the source and date of the work, but do have a standard and use it. When you add a new data element to an existing table, the **Description** property should receive the same modification identifier that you would place in the code comments.



When you make a subsequent change to an object, you should document that change in the Documentation trigger and also in the code, as described earlier. Inline comments can be done in two ways. The most visible way is to use a // character sequence (two forward slashes). The text that follows the slashes on that line will be treated as a comment by the compiler, that is, it will be ignored. If the comment spans two physical lines, the second portion of the comment must also be preceded by two forward slashes.

In the following screenshot we have used `//` to place comments inline in code to identify a change:



```

Gift Ledger - OnAfterGetRecord()
IF (NOT PrintServices) AND (Category = 'SERVICES') THEN
    CurrReport.SKIP;
//PN.06.01 start -----
IF Category = 'SERVICES' THEN
    "Estimated Value" := Hours * CANSetup."Services Rate";
//PN.06.01 end -----
Gift Ledger - OnPostDataItem()

```

Here we have made the modification version number **01**, resulting in a change version code of **PN.06.01**. In the following code, modifications are highlighted by bracketing the additional code with comment lines containing the modification identifier, and **start** and **end** indicators. The NAV-published standards do not include the dashed lines, as shown here, but doing something like that often makes it easier to spot modifications when you are scanning code rapidly. You can create your own standards, but be consistent in whatever style you use.

The second way to place a comment within code is to surround the comment with a matched pair of braces `{ }`. As braces are less visible than the slashes, you should always use `//` when your comment is relatively long. If you want to use `{ }`, it's a good idea to insert a `//` comment at the beginning and end of the material inside the braces, to make the existence of the comments more readily identifiable.

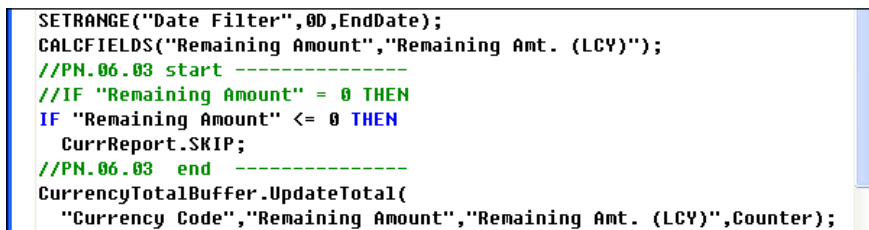
For example:

```

{ //PN.06.02 start deletion -----
//PN.06.02 replace validation with a call to an external function
...miscellaneous C/AL validation code
//PN.06.02 end deletion ----- }

```

When you delete code, you should always leave original statements in place, but commented out so that it is inoperative (an exception to this may apply if a source code control system is in use which tracks all changes). When you change the existing code, you should leave the original code in place, but commented out, with the new version being inserted as shown in the following screenshot:



```

SETRANGE("Date Filter",00,EndDate);
CALCFIELDS("Remaining Amount","Remaining Amt. (LCY)");
//PN.06.03 start -----
//IF "Remaining Amount" = 0 THEN
IF "Remaining Amount" <= 0 THEN
    CurrReport.SKIP;
//PN.06.03 end -----
CurrencyTotalBuffer.UpdateTotal(
    "Currency Code","Remaining Amount","Remaining Amt. (LCY)",Counter);

```


Don't forget to update the external version numbers located in the Version List field on the **Object Designer** screen.

From previous experience, you know that it is not the format of the internal documentation that is critical. What is critical is that the documentation exists in a consistent and reliable fashion which accurately describes the changes that have occurred. The ideal situation is to also have external documentation which defines at least the original requirements, validation specifications, and recommended operating procedures.

Computation and Validation utility functions

C/AL includes a number of utility functions designed to facilitate data computations and validation or initializing of field contents. The following are some of the Validation utility functions:

- TESTFIELD
- FIELDERROR
- VALIDATE
- ROUND
- TODAY, TIME, and CURRENTDATETIME functions
- WORKDATE functions

TESTFIELD

The TESTFIELD function is widely used in standard NAV code. With TESTFIELD, you can test a variable value and, if the test fails, issue an error message in a single statement. The syntax is as follows:

```
Record.TESTFIELD (Field, [Value] )
```

If a Value is specified and the field does not contain that value, an error condition is raised (that is, the process terminates) and the associated error message is issued. If no Value is specified, the condition evaluated is to compare the field contents to zero or blank. If no Value is specified and the field is zero or blank, then that is an error.

The advantage of TESTFIELD is ease of use and consistency in code. The disadvantage is that the error message, although not as hard to understand as some others, is not as informative as you might provide as a careful developer.

The following screenshot of TESTFIELD usage is from Table 27—Item. This code checks to make sure the field "Qty. per Unit of Measure" is equal to 1.

```

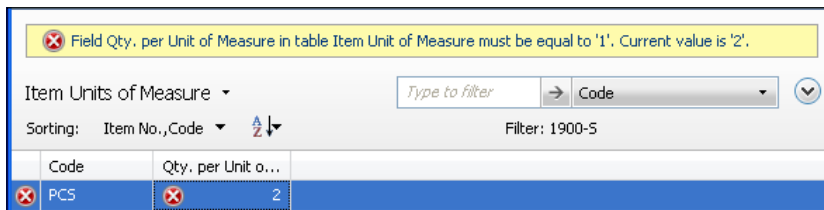
Base Unit of Measure - OnValidate()
TestNoOpenEntriesExist(FIELDCAPTION("Base Unit of Measure"));

"Sales Unit of Measure" := "Base Unit of Measure";
"Purch. Unit of Measure" := "Base Unit of Measure";
IF "Base Unit of Measure" <> '' THEN BEGIN
    ItemUnitOfMeasure.GET("No.", "Base Unit of Measure");
    ItemUnitOfMeasure.TESTFIELD("Qty. per Unit of Measure", 1);
END;
IF CurrFieldNo <> 0 THEN
    MODIFY(TRUE);

Base Unit of Measure - OnLookup()

```

An example of the error message generated when the Item "Qty. per Unit of Measure" is not equal to 1 follows:



FIELDERROR

Another function very similar to the TESTFIELD function is the FIELDERROR function. But where TESTFIELD performs a test and terminates with either an error or an OK result, FIELDERROR presumes that the test was already performed and the field failed the test. The FIELDERROR is designed to display an error message, and then trigger a runtime error, thus terminating the process. The syntax is as follows:

```
TableName.FIELDERROR(FieldName[,OptionalMsgText]);
```

If you include your own message text, for example, defining a Text Constant and using it in code:

```
Text004    the E-mail address format is not correct

Donor.FIELDERROR("E-mail",Text004);
```

You will see an error message from FIELDERROR similar to the following image:



The error message begins with the name of the field, which is the `FieldName` parameter in the function call (in this case **E-mail**), followed by your specified `MsgText` (**the E-mail address format is not correct**), which is in turn followed by the word **"in"**, the qualified name of the first table field (**Donor.Donor ID**) and the value in that field (**1098**). The **Refresh** referred to by the instructions means the *F5* key.

We use a Text Constant field, rather than an embedded text literal, so that the code can be multilingual.

If you don't include your own message text, your function call will look like this:

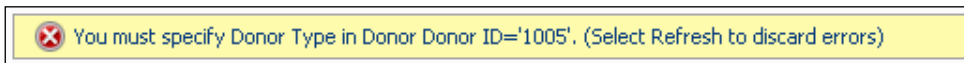
```
Donor.FIELDERROR ("Donor Type" );
```

If you don't specify your own message text, the default message comes in two flavors. The first instance is the case where the referenced field is not empty, such as in the following screenshot. In this case, the content of the field is **BUSINESS**.



The error message logic presumes that the error is due to a wrong value.

Following is another instance of a `FIELDERROR` function call with no message text supplied. In this case the field was empty. The resulting error message logic presumes that the error is the due to empty field, shown in the following screenshot:



VALIDATE

The syntax of the `VALIDATE` function is as follows:

```
Record.VALIDATE ( Field [, Value] )
```

`VALIDATE` will fire the `OnValidate` trigger of `Record.Field`. If you have specified a `Value`, then that `Value` is assigned to the field and the field validations are invoked.

If you don't specify a `Value`, then the field validations are invoked using the field value that already exists in the field. This function allows you to easily centralize your code design around the table, a definite advantage and one of NAV's strengths.

For example, if we were to code changing the Item "Base Unit of Measure" to another unit of measure, the code should make sure the change is valid. We should get an error if the new unit of measure has any quantity other than 1.

Making the unit of measure change with a simple assignment statement would not catch a quantity error.

```
Item."Base Unit of Measure" := NewUnitOfMeasure;
```

If this were coded with a simple `VALIDATE`, the assigned value would be checked and the error caught.

```
Item."Base Unit of Measure" := NewUnitOfMeasure;
Item.VALIDATE("Base Unit of Measure");
```

The simpler code, which does both the assignment and validation in a single statement is:

```
Item.VALIDATE("Base Unit of Measure",NewUnitOfMeasure);
```

ROUND

The `ROUND` function allows you to control the rounding precision for a decimal expression. The syntax for the `ROUND` function is as follows:

```
DecimalResult := ROUND (Number [, Precision] [, Direction] )
```

where `Number` is what is being rounded, `Precision` spells out the number of digits of decimal precision, and `Direction` indicates whether to round up, round down, or round to the nearest number. More specifically, some examples of `Precision` values are as follows:

Precision value	Rounding effect
100	To a multiple of 100
1	To an integer value
.01	To two decimal places (the US default)
0.01	Same as .01
.0001	To four decimal places

As noted, if no `Precision` value is specified, the US Localization will default to two decimal places, the standard for US currency. Default options in other localizations may differ.

The options available for the Direction value are shown in the following table:

Direction value (a text value)	Rounding effect
'='	Round to the nearest (mathematically correct)
'>'	Round up
'<'	Round down

The following statement:

```
DecimalValue := ROUND (1234.56789,0.001,'<')
```

would result in a DecimalValue containing 1234.567, whereas the statements:

```
DecimalValue := ROUND (1234.56789,0.001,'=')  
DecimalValue := ROUND (1234.56789,0.001,'>')
```

would each result in a DecimalValue containing 1234.568.

TODAY, TIME, and CURRENTDATETIME functions

TODAY retrieves the current system date as set in the operating system. TIME retrieves the current system time as set in the operating system. CURRENTDATETIME retrieves the current date and time in the DATETIME format, which is stored in UTC international time and then displayed in local time. The syntax is as follows:

```
DateField := TODAY;  
TimeField := TIME;  
DateTimeField := CURRENTDATETIME;
```

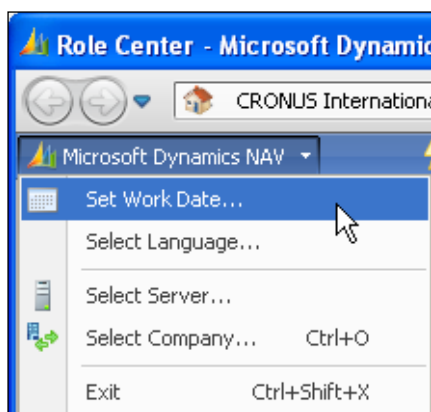
These are useful for date- and time-stamping transactions or for filling in default values in fields of the appropriate data type. For data entry purposes, the current system date can be entered by simply typing a letter T or the word **TODAY** in the date entry field (this is not a case-sensitive entry). NAV will automatically convert that entry to the current system date.

The NAV C/SIDE Server uses the date of January 1, 0000 as the earliest date (0D) and subsequent dates through December 31, 9999. The Microsoft Dynamics NAV undefined time (0T) is represented by the same value as an undefined date (0D). The undefined date in the RTC is represented by the earliest valid DATETIME in SQL Server, which is January 1, 1753 00:00:00.000.

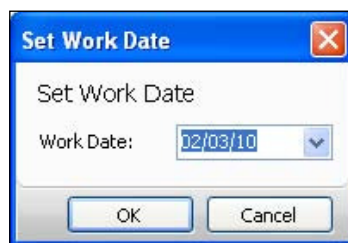
In order to assure compatibility with previous versions, NAV 2009 supports the date January 1, 0000, then the date range from January 1, 1753 through December 31, 9999. If a date greater January 1, 0000, but less than January 1, 1753 inadvertently shows up in the RTC, a runtime error will occur.

WORKDATE function

A useful feature of NAV is the Work Date. Many standard NAV routines default dates to **Work Date** rather than to the system date. When a user logs into the system, the Work Date is initially set equal to the System Date. But at any time, the operator can set the Work Date to any date by accessing **Set Work Date...** from the **Microsoft Dynamics NAV** menu (as shown in the next screenshot) and then entering the desired new Work Date.



The following screenshot shows the **Set Work Date** screen:



The syntax for using the `WorkDate` function in C/AL is as follows:

```
DateField := WORKDATE;
```

For data entry purposes, the current system date can be entered by the operator simply typing a letter **w** or **W** or the word **WORKDATE** in the date entry field. NAV will automatically convert that entry to the current Work Date.

Data conversion functions

Some data type conversions are handled in the normal process flow by NAV without any particular attention on part of the Developer (for example, Code to Text, Char to Text). Some data type conversions can only be handled through C/AL functions. Formatting is included as a data type conversion.

FORMAT function

The `FORMAT` function provides for the conversion of an expression of any data type (for example, integer, decimal, date, option, time, Boolean) into a formatted string. The syntax is as follows:

```
StringField := FORMAT( ExpressionToFormat [, OutputLength]
                        [, FormatString or FormatNumber] )
```

The formatted output of the `ExpressionToFormat` will be assigned to the output `StringField`. The optional parameters control the conversion according to a complex set of rules. These rules can be found in the **C/SIDE Reference Guide Help** file for the `FORMAT` function. Whenever possible, you should always apply `FORMAT` in its simpler form. The best way to determine the likely results of a `FORMAT` expression is to test it through a range of the values to be formatted. Make sure that you include the extremes of the range of possible values in your testing.

The optional `OutputLength` parameter can be zero (which is the default), a positive integer, or a negative integer. The typical `OutputLength` value is either zero, in which case, the defined format is fully applied, or it is a figure designed to control the maximum character length of the formatted string result.

The last optional parameter has two totally separate sets of choices. One set, represented by an integer `FormatNumber`, allows the choice of a particular predefined (that is, standard) format, of which there are four to nine choices depending on the `ExpressionToFormat` data type. The other set of choices allows you to build your own format expression.

The **C/SIDE Reference Guide Help** information for the `FORMAT` property provides a relatively complete description of the available tools from which you can build your own format expression. The `FORMAT` property **Help** also provides a complete list of the predefined format choices as well as a good list of example sample formats and the resulting formatted data.

Note that a `FORMAT` function which cannot be executed will result in a run-time error that will terminate execution of the process. Thus, to avoid production crashes, you will want to place a high importance on thoroughly testing any code where `FORMAT` is used.

EVALUATE function

The `EVALUATE` function is essentially the reverse of the `FORMAT` function. It allows you to convert a string field into the defined data type. The syntax of the `EVALUATE` function is as follows:

```
[ BooleanField := ] EVALUATE ( ResultField, StringToBeConverted [, 9]
```

The handling of a run-time error can be done by specifying `BooleanField` or including `EVALUATE` in an expression which will deal with an error (such as an `IF` statement). The `ResultField` data type will determine what data conversion the `EVALUATE` function will attempt. The data type of the `ResultField` must be one of the following: integer, Boolean, date, time, code, option, text constant, or GUID. The format of the data in `StringToBeConverted` must be compatible with the data type of `ResultField` otherwise a run-time error will occur.

The optional parameter, number 9, only applies for XMLport data exporting. Use of the optional number 9 parameter will convert C/SIDE format data types into XML standard data types. This is used to deal with the fact that several equivalent C/SIDE-XML data types are represented differently at the base system level (that is, "under the covers"). The C/SIDE data types include decimal, Boolean, datetime, date, time, integer and duration.

DATE functions

In order to convert numeric data to Date data types and Dates to numeric data, C/AL uses a series of `DATE` functions.

DATE2DMY function

`DATE2DMY` allows you to extract the sections of a date (Day, Month, and Year) from a Date field. The syntax is as follows:

```
IntegerVariable := DATE2DMY ( DateField, ExtractionChoice )
```


The fields `IntegerVariable` and `DateField` are just as their names imply. The `ExtractionChoice` parameter allows you to choose which value (Day, Month, or Year) will be assigned to the `IntegerVariable`. The following table provides the `DATE2DMY` extraction choices:

DATE2DMY extraction choice	Integer value result
1	2 digit day (1 - 31)
2	2 digit month (1 - 12)
3	4 digit year

DATE2DWY function

`DATE2DWY` allows you to extract the sections of a date (Day of the week, Week of the year, and Year) from a Date field. The syntax is as follows:

```
IntegerVariable := DATE2DWY ( DateField, ExtractionChoice )
```

The fields `IntegerVariable` and `DateField` are just as their names imply. The `ExtractionChoice` parameter allows you to choose which value (Day, Week, or Year) will be assigned to the `IntegerVariable`.

The following table provides the `DATE2DWY` extraction choices:

DATE2DWY extraction choice	Integer value result
1	2 digit day (1 - 7 for Monday - Sunday)
2	2 digit week (1 - 53)
3	4 digit year

DMY2DATE and DWY2DATE functions

`DMY2DATE` allows you to create a date from integer values (or defaults) representing the day of the month, month of the year, and the four-digit year. If an optional parameter (Month or Year) is not specified, the corresponding value from the **system date** is used. The syntax is as follows:

```
DateVariable := DMY2DATE ( DayValue [, MonthValue] [, YearValue] )
```

The only way to have the function use **Work Date** values for Month and Year is to extract those values and then use them explicitly. An example is as follows:

```
DateVariable := DMY2DATE (22, DATE2MDY (WORKDATE, 2) , DATE2MDY (WORKDATE, 3) )
```



This example also illustrates how expressions can be built up of nested expressions and functions. We have `WorkDate` within `DATE2MDY` within `DMY2DATE`.

`DWY2DATE` operates similarly to `DMY2DATE`; allowing you to create a date from integer values representing the day of the week (1 to 7, that is, Monday to Sunday), week of the year (from 1 to 53), and the four-digit year. The syntax is as follows:

```
DateVariable := DWY2DATE ( DayValue [, WeekValue] [, YearValue] )
```

An interesting result can occur for week 53 because it can span two years. In that case the year of the result will vary depending on the day of the week in the parameters (that is, the year of the result may differ from the year specified in the parameters).

CALCDATE function

`CALCDATE` allows you to calculate a date value assigned to a `Date` data type variable based on a `Date Expression` applied to a `Base` or `Reference Date`. If you don't specify a `BaseDateValue`, the current system date is used as the default date. Otherwise, the `BaseDateValue` can be supplied either in the form of a variable of data type `Date` or as a `Date` constant.

The syntax for `CALCDATE` is as follows:

```
DateVariable := CALCDATE ( DateExpression [, BaseDateValue] )
```

There are a number of ways in which you can build a `DateExpression`. The rules for the `CALCDATE` function `DateExpression` are similar to the rules for `DateFormula` described in Chapter 3.

If there is a `CW`, `CM`, `CP`, `CQ`, or `CY` (Current Week, Current Month, Current Period, Current Quarter, Current Year) parameter in an expression, then they will be evaluated based on the `BaseDateValue`. If you have more than one of these in your expression, the results are unpredictable.

If your `Date Expression` is stored in a `DateFormula` variable (or a `Text` or `Code` variable with the **DateFormula** property set to **Yes**), then the `Date Expression` will be language independent. If you create your own `Date Expression` in the form of a string constant within your inline `C/AL` code, surrounding the constant with `< >` delimiters as part of the string, that will make the constant language independent. Otherwise, the `Date Expression` constant will be language dependent.

Regardless of how you have constructed your `DateExpression`, it is important to test it carefully and thoroughly before moving on. Incorrect syntax will result in a runtime error. One easy way to test is by using a Report whose sole task is to evaluate your expression and display the result. If you want to try different Base Dates, you can use the Request Page, accept the Base Date as input, then calculate and display the `DateVariable` in the `OnValidate` trigger.

Some sample `CALCDATE` expression evaluations are as follows:

- `('<CM>', 031010D)` will yield 03/31/2010, that is, the last day of the Current Month for the date 3/10/2010.
- `('<-WD2>', 031011D)` will yield 03/08/2011, that is, the WeekDay #2 (the prior Tuesday) before the date 3/10/2011.
- `('<CM+1D>', BaseDate)` where `BaseDate` equals 03/10/10, will yield 04/01/2010, that is, the last day of the month of the Base Date plus one day (the first day of the month following the Base Date).

FlowField-SumIndexField functions

In the chapter on Fields, we discussed `SumIndexFields` and `FlowFields`. To recap briefly, `SumIndexFields` are defined in the screen where keys are defined. They allow very rapid calculation of values in filtered data. In most of the systems, the calculation of most types of group totals, periodic totals, and such, required passing all of the data to be totaled. SIFT technology allows a NAV system to respond almost instantly with such totals in any area where the `SumIndexField` was defined. In fact, use of SIFT totals combined with NAV's retention of detailed data supports totally flexible ad hoc queries of the form "What were our sales for red widgets between the dates of November 15th through December 24th?" And the answer is returned fast! `SumIndexFields` are the basis of `FlowFields` which have a **Method** of `Sum` or `Average`; such a `FlowField` must refer to a data element that is defined as a `SumIndexField`.

`SumIndexFields` are the basis of `FlowFields`; a `FlowField` must refer to a data element that is defined as a `SumIndexField`. When you access a record that has a `SumIndexField` defined, there is no visible evidence of the data sum that `SumIndexField` represents. When you access a record that contains `FlowFields`, the `FlowFields` are empty virtual data elements until they are calculated. When a `FlowField` is displayed in a page, it is automatically calculated by NAV; the developer doesn't need to do so. But in any other scenario, the developer is responsible for calculating `FlowFields` before they are used.

FlowFields are one of the key (all puns intended) areas where NAV systems are subject to significant processing bottlenecks, especially in SQL Server systems. In versions older than V5.0 SP1, the data used to calculate FlowFields (that is, the SumIndexFields) was maintained in separate tables. It was critical to minimize the number of FlowFields, to optimize the design of indexes used to maintain and access them, and to frequently optimize the operational indexes within SQL Server.

Beginning with V5.0 SP1 and NAV 2009, the FlowFields are now calculated with a SQL Server feature called "Indexed Views". This feature eliminates the need for the separate indexes to support FlowFields and is therefore much more efficient and easier to maintain. (For additional information, look up **SIFT** in the **C/SIDE Reference Guide** Help, especially **SIFT and Microsoft Dynamics NAV with Microsoft SQL Server**.)

Even with the improved SQL Server SIFT design, it is still critical that the Keys used for SumIndexField definition are designed with efficient processing in mind. Sometimes, as part of a performance tuning effort, it's necessary to revise or add new keys to improve FlowField performance. Note that even though you can manage indexes in SQL Server independent of the NAV key definitions, doing so will make your system more difficult to support in the long run because the SQL Server resident changes aren't visible within NAV.

In addition to being careful about the SIFT-key structure design, it is also important not to define any SumIndexFields that are not necessary. Each additional SumIndexField adds additional processing requirements and thus adds to the processing load of the system.

CALCFIELDS function

The syntax for CALCFIELDS is as follows:

```
[BooleanField := ] Record.CALCFIELDS ( FlowField1 [, FlowField2] ,...)
```

Executing the CALCFIELDS function will cause all the specified FlowFields to be calculated (that is, updated). Specification of the BooleanField allows you to handle any run-time error that may occur. Any runtime errors for CALCFIELDS usually result from a coding error or a change in a table key structure.

The FlowField calculation takes into account the filters (including FlowFilters) that are currently applied to the Record. After the CALCFIELDS execution, the included FlowFields can be used similarly to any other data fields. The CALCFIELDS must be executed for each cycle through the subject table.

Whenever the contents of a BLOB field are to be used, CALCFIELDS is used to load the contents of the BLOB field from the database into memory.

CALCSUMS function

The CALCSUMS function is conceptually similar to CALCFIELDS. But the CALCFIELDS operates on FlowFields and CALCSUMS differs by operating directly on the record and field where the SumIndexFields are defined. That difference means that you must specify the proper key plus any filters to apply when using CALCSUMS (the applicable key and filters to apply are already defined in the properties for FlowFields).

The syntax for CALCSUMS is as follows:

```
[ BooleanField := ] Record.CALCSUMS ( SIFTField1 [,SIFTField2] ,...)
```

Prior to this statement, you must have specified a key that has the SIFTFields defined. Before executing the CALCSUMS function, you also need to specify any filters that you want to apply to the Record from which the sums are to be calculated. The SIFTField calculations take into account the filters that are currently applied to the Record, but those filters must apply to key fields associated with the chosen SIFTfield.

Executing the CALCSUMS function will cause all the specified SIFTField totals to be calculated. Specification of the BooleanField allows you to handle any runtime error that may occur. Runtime errors for CALCSUMS usually result from a coding error or a change in a table key structure.

Before the execution of CALCSUMS, SIFTFields contain only the data from the individual record that was read. After the CALCSUMS execution, the included SIFTFields contain the totals that were calculated by the CALCSUMS function (these totals only affect the data in memory, not that on the disk). These totals can then be used the same as data in any field, but if you want to access the individual record's original data for that field, you must either save a copy of the record before executing the CALCSUMS or you must re-read the record. The CALCSUMS must be executed for each read cycle through the subject table.

CALCFIELDS and CALCSUMS comparison

In the Sales Header record, there are FlowFields defined for Amount and "Amount Including VAT". These FlowFields are all based on Sums of entries in the Sales Line table. The CalcFormula for Amount is `Sum("Sales Line".Amount WHERE (Document Type=FIELD(Document Type) , Document No.=FIELD(No.)))`. To calculate a TotalOrderAmount value while referencing the Sales Header table, the code can be as simple as:

```
TotalOrderAmount := "Sales Header".CALCFIELDS (Amount);
```

To calculate the same value from code directly referencing the Sales Line table, the required code is similar to the following (assuming a Sales Header record has already been read):

```
"Sales Line".SETRANGE("Document Type", "Sales Header"."Document Type");
"Sales Line".SETRANGE("Document No.", "Sales Header"."No.");
CALCSUMS(Amount);
TotalOrderAmount := Amount;
```

Flow control

The structures defined for flow control are discussed in the following subsections:

REPEAT-UNTIL control structure

REPEAT-UNTIL allows you to create a repetitive code loop REPEATing a block of code UNTIL a specific conditional expression evaluates to TRUE. In that sense, REPEAT-UNTIL defines a block of code, operating like the BEGIN-END compound statement structure which we covered in the previous chapter. In this case, the REPEAT tells the system to keep reprocessing the block of code, while the UNTIL serves as the exit doorman, checking if the conditions for ending the processing are true. Because the exit condition is not evaluated until the end of the loop, a REPEAT-UNTIL structure will always process at least once through the contained code.

REPEAT-UNTIL is very important in NAV because it is frequently part of the data input cycle with the FIND-NEXT structure, which will be covered shortly.

An example of the REPEAT-UNTIL structure to process data in a 10-element array is as follows:

```
LoopCount := 0;
REPEAT
    LoopCount := LoopCount + 1;
    TotCustSales := TotCustSales + CustSales[LoopCount];
UNTIL LoopCount = 10;
```

WHILE-DO control structure

A WHILE-DO control structure allows you to create a repetitive code loop DOing a block of code WHILE a specific conditional expression evaluates to TRUE. WHILE-DO is different from REPEAT-UNTIL, both in the possible need for a BEGIN-END structure to define a block of code and in the timing of the evaluation of the exit condition.

The syntax of the WHILE-DO control structure is as follows:

```
WHILE <Condition> DO <Statement>
```

The Condition can be any Boolean expression which evaluates to TRUE or FALSE. The Statement can be a simple statement or the most complex possible compound BEGIN-END statement. Most WHILE-DO loops will contain a BEGIN-END block of code. The Condition will be evaluated at the beginning of the loop. When it evaluates to FALSE, the loop will terminate, meaning that a WHILE-DO loop can be exited without processing.

An example of the WHILE-DO structure to process data in a 10-element array is as follows:

```
LoopCount := 0;
WHILE LoopCount < 10
DO BEGIN
    LoopCount := LoopCount + 1;
    TotCustSales := TotCustSales + CustSales[LoopCount];
END;
```

As, in many cases, the WHILE-DO is slower than REPEAT-UNTIL, only use WHILE-DO when the application logic demands it.

CASE-ELSE statement

The CASE-ELSE statement is a conditional expression very similar to IF-THEN-ELSE except that it allows for more than two choices of outcomes for the evaluation of the controlling expression. The syntax of the CASE-ELSE statement is as follows:

```
CASE <ExpressionToBeEvaluated> OF
    <Value Set 1> : <Action Statement 1>;
    <Value Set 2> : <Action Statement 2>;
    <Value Set 3> : <Action Statement 3>;
    ...
    ...
    <Value Set n> : <Action Statement n>;
    [ELSE <Action Statement n + 1>;
END;
```

The ExpressionToBeEvaluated must not be a record. The data type of the Value Set must be compatible with (that is, able to be automatically converted to) the data type of the ExpressionToBeEvaluated. Each Value Set must be an expression, a set of values or a range of values. The following example illustrates a typical instance of a CASE-ELSE statement:

```

CASE Customer."Salesperson Code" OF
  '2','5','9': Customer."Territory Code" := 'EAST';
  '6'..'8': Customer."Territory Code" := 'WEST';
  '3': Customer."Territory Code" := 'NORTH';
  '1'..'4': Customer."Territory Code" := 'SOUTH';
ELSE Customer."Territory Code" := 'FOREIGN';
END;

```

In this example, you can see several alternatives for the Value Set. The first line (EAST) Value Set is a list of values. If "Salesperson Code" is equal to '2' or '5' or '9', the value EAST will be assigned to Customer."Territory Code". The second line (WEST) Value Set is a range, any value from '6' through '8'. The third line (NORTH) Value Set is just a single value ('3'). Looking at the bulk of standard code, you will see that the single value is the norm for CASE structures. The fourth line (SOUTH) Value Set is again a range ('1'..'4'). If nothing in any Value Set matches ExpressionToBeEvaluated, then the ELSE clause will be executed.

An example of an IF-THEN-ELSE statement equivalent to the preceding CASE-ELSE statement is as follows:

```

IF Customer."Salesperson Code" IN ['2','5','9'] THEN
  Customer."Territory Code" := 'EAST'
ELSE IF Customer."Salesperson Code" IN ['6'..'8'] THEN
  Customer."Territory Code" := 'WEST'
ELSE IF Customer."Salesperson Code" = '3' THEN
  Customer."Territory Code" := 'NORTH'
ELSE IF Customer."Salesperson Code" IN ['1'..'4'] THEN
  Customer."Territory Code" := 'SOUTH'
ELSE Customer."Territory Code" := 'FOREIGN';

```

The following is a more creative, and somewhat less intuitive example of the CASE-ELSE statement. In this instance, ExpressionToBeEvaluated is a simple TRUE and the Value Set statements are all conditional expressions. The first line containing a Value Set expression that evaluates to TRUE will be the line whose Action Statement is executed. The rules of execution and flow in this instance are same as the previous example.

```

CASE TRUE OF
  Salesline.Quantity < 0:
  BEGIN
    CLEAR(Salesline."Line Discount %");
    CredTot := CredTot - Salesline.Quantity;
  END;
  Salesline.Quantity > QtyBreak[1]:
    Salesline."Line Discount %" := DiscLevel[1];

```



```
Salesline.Quantity > QtyBreak[2]:  
    Salesline."Line Discount %" := DiscLevel[2];  
Salesline.Quantity > QtyBreak[3]:  
    Salesline."Line Discount %" := DiscLevel[3];  
Salesline.Quantity > QtyBreak[4]:  
    Salesline."Line Discount %" := DiscLevel[4];  
ELSE  
    CLEAR(Salesline."Line Discount %");  
END;
```

WITH-DO statement

When you are writing code referring to fields within a record, the most specific syntax for field references is the fully qualified reference. When referring to the field **City** in the record **Customer**, use the reference `Customer.City`.

In many C/AL instances, the record name qualifier is implicit, that is, the compiler assumes a default record qualifier based on context within the code. This happens automatically for variables within a page that is bounded to a table. The bound table becomes the implicit record qualifier for fields referenced in the Page object. In a Table object, the table is the implicit record qualifier for fields referenced in the C/AL internal to that object. In Report and XMLport objects, the Data Item record is the implicit record qualifier for the fields referenced within Data Item-specific triggers (for example, `OnAfterGetRecord`, `OnAfterImportRecord`, and so on).

In all other C/AL code, the only way to have an implicit record qualifier is to use the `WITH-DO` statement. `WITH-DO` is widely used in the base product in Codeunits and processing Reports. The `WITH-DO` syntax is as follows:

```
WITH <RecordQualifier> DO <Statement>
```

Typically, the `DO` portion of this statement will be followed by a `BEGIN-END` code block, that is, the `Statement` will be a compound statement. The scope of the `WITH-DO` statement is terminated by the end of the `DO` statement.

When you execute a `WITH-DO` statement, `RecordQualifier` becomes the implicit record qualifier used by the compiler until the end of the statement or until it is overridden by a nested `WITH-DO` statement. Where fully qualified syntax would require the following form:

```
Customer.Address := '189 Maple Avenue';  
Customer.City := 'Chicago';
```

the `WITH-DO` syntax takes advantage of the implicit record qualification making the code easier to write, and hopefully easier to read, for example:

```
WITH Customer DO
BEGIN
  Address := '189 Maple Avenue';
  City := 'Chicago';
END;
```

Nested `WITH-DO` statements are valid, but not used. They are also not recommended because they can easily confuse the developer, resulting in bugs. The same comments apply to nesting a `WITH-DO` statement within a function where there is an automatic implicit record qualifier, such as in a table, bound Page, report, or Dataport. Of course, wherever the references to other record variables occur within the scope of a `WITH-DO`, you must include the specific qualifiers. This is particularly important when there are variables with the same name (for example, `City`) in multiple tables that might be referenced in the same set of C/AL logic.

Some developers maintain that it is always better to use fully qualified variable names to reduce the possibility of inadvertent reference errors. This approach also eliminates any possible misinterpretation of variable references by the next developer maintaining the code.

QUIT, BREAK, EXIT, SKIP, and SHOWOUTPUT functions

There is a group of C/AL functions that can be used to control the flow and affect the processing under different circumstances. Each acts to interrupt flow in different places and with different results. To get a full appreciation for how these functions are used, you need to review them in place in code in NAV 2009. Focus on the first four and only check out `SHOWOUTPUT` in the Classic reports so you can better understand the equivalent functionality in RTC reports.

QUIT function

The `QUIT` function is the ultimate processing interrupt for Report, Dataport, or XMLport objects. When a `QUIT` is executed, processing immediately terminates even for the `OnPostObject` triggers. `QUIT` is often used in reports to terminate processing when the report logic determines that no useful output will be generated by further processing.

The syntax of the QUIT function is as follows:

```
CurrReport.QUIT;  
CurrDataport.QUIT;  
CurrXMLport.QUIT;
```

BREAK function

The effect of a BREAK function depends on the context in which it executes. If the BREAK is within a loop structure such as a WHILE-DO or REPEAT-UNTIL loop, BREAK exits the loop as if the loop exit condition had been satisfied except the exit is at the point of the BREAK. If the BREAK function is not in a loop, then its execution will exit the host trigger. BREAK can only be used in Data Item triggers in Reports, Dataports, and XMLports. BREAK is often used to break or terminate the sequence of processing one segment of a report while allowing the overall processing to continue.

The BREAK syntax is one of the following:

```
CurrReport.BREAK;  
CurrDataport.BREAK;  
CurrXMLport.BREAK;
```

EXIT function

EXIT is used to end the processing within a C/AL trigger. EXIT works the same whether it is executed within a loop or not. EXIT can be used simply to end the processing of the trigger or to pass a return parameter from a local function. If EXIT is used without a return value then a default return value of zero is returned. The syntax for EXIT is as follows:

```
EXIT([<ReturnValue>])
```

EXIT could be considered as an acceptable substitute for the dreaded GOTO. EXIT is frequently used in functions to pass back a return value.

SKIP function

When executed, the SKIP function will skip the remainder of the processing in the current cycle in the current trigger. Unlike BREAK, it does not terminate processing in the trigger. It can be used only in the OnAfterGetRecord trigger of a Report, Dataport, or XMLport object. In many reports, when the results of processing in the OnAfterGetRecord trigger are determined not to be useful for output, the SKIP function is used to terminate just that single iteration of the trigger without interfering with any other processing.

The SKIP syntax is one of the following:

```
CurrReport.SKIP;  
CurrDataport.SKIP;  
CurrXMLport.SKIP;
```

SHOWOUTPUT function

SHOWOUTPUT can be used only in the OnPreSection trigger of Classic Report objects. SHOWOUTPUT has no direct corresponding function in VS RD Reports. To replicate similar capabilities in VS RD Reports, see the **C/SIDE Reference Guide Help for How to: Apply Conditional Visibility Controls**.

Input and Output functions

In the previous chapter, we learned about the basics of the FIND function. We learned about FIND ('-') to read the beginning of the selected records for the Classic Client, FINDSET to read a selected set of records and FIND ('+') to begin reading at the far end of the selected records for both clients. Now we will review additional functions that are generally used with FIND functions in typical production code. While designing the code by using the MODIFY and DELETE record functions, you need to consider the possible interactions with other users on the system. There might be someone else modifying and deleting records in the same table in which your application is working.

You may want to utilize the LOCKTABLE function to gain total control of the data briefly, while updating the data. You can find more information on LOCKTABLE in the online **C/AL Reference Guide Help**. Be aware that LOCKTABLE performs quite differently in the C/SIDE database from how it performs in the SQL Server database. While the C/SIDE database only supported table locking, the SQL Server database supports Record Level Locking. There are a number of factors that you should consider when coding data locking in your processes. It is worthwhile reading all of the **C/AL Reference Guide** material found by a Search on LOCKTABLE, particularly **Locking in Microsoft SQL Server**.

NEXT function with FIND or FINDSET

The syntax defined for the NEXT function is as follows:

```
IntegerValue := Record.NEXT ( ReadStepSize )
```

The full assignment statement format is rarely used to set an IntegerValue. In addition, there is no documentation for a non-zero IntegerValue. When IntegerValue goes to zero, it means that the read loop has reached the end of the selected record set.

If the `ReadStepSize` value is negative, the table will be read in reverse; if that value is positive (the default), then the table will be read forward. The size of the value in `ReadStepSize` controls which records should be read. For example, if `ReadStepSize` is 2 or -2, then every second record will be read. If `ReadStepSize` is 10 or -10, then every tenth record will be read. The default value is zero, in which case every record will be read (the same as if it were 1 or +1) and the read direction will be forward.

In a typical data input loop, the first read is a `FIND` or `FINDSET` function followed by a `REPEAT-UNTIL` loop. The exit condition is a `NEXT` expression similar to `UNTIL Record.NEXT = 0;`. The C/AL for `FINDSET` and `FIND(-)` are structured the same.

The full C/AL syntax would look like the following:

```
IF CustRec.FIND('-') THEN
REPEAT
    Block of C/AL logic
UNTIL CustRec.NEXT = 0;
```

INSERT function

The purpose of the `INSERT` function is to insert (that is, add) records into the table. The syntax for the `INSERT` function is as follows:

```
[BooleanValue :=] Record.INSERT ( [ TriggerControlBoolean ] )
```

If `BooleanValue` is not used and the `INSERT` function fails (for example, if inserting would result in a duplicate Primary Key), then the process will terminate with an error. Most of the time, a detected error should be handled in code rather than allowing process termination.

The `TriggerControlBoolean` value, a `TRUE` or `FALSE` entry, controls whether or not the table's `OnInsert` trigger fires when the `INSERT` occurs. The default value is `FALSE`. If you let the default `FALSE` control, you run the risk of not performing error checking that the table's designer assumed would be run when a new record was added.



If you are reading a table and you need to also `INSERT` records into that table, the `INSERT`s should be done to a separate instance of the table, using either a global or local variable. Otherwise, you run the risk of reading your new records as part of your processing (normally a very confusing action) and the risk of changing the sequence of your processing unexpectedly due to the introduction of new records into your data set.

MODIFY function

The purpose of the `MODIFY` function is to modify (that is, update) the existing data records. The syntax for `MODIFY` is as follows:

```
[BooleanValue :=] Record.MODIFY ( [ TriggerControlBoolean ] )
```

If `BooleanValue` is not used and `MODIFY` fails (for example, if another process changes the record after it was read by this process), then the process will terminate with an error statement. Any detected error should either be handled or should terminate the process. The `TriggerControlBoolean` value is a `TRUE` or `FALSE` entry, which controls whether or not the table's `OnModify` trigger fires when this `MODIFY` occurs. The default value is `FALSE`.

`MODIFY` cannot be used to cause a change in a Primary Key field. In that case, the `RENAME` function must be used.

Rec and xRec

In Table and Page objects, the system automatically provides you with the system variables `Rec` and `xRec`. After a record has been updated by `MODIFY`, `Rec` represents the current record data in process and `xRec` represents the record data before it was modified. By comparing field values in `Rec` and `xRec`, you can determine if changes have been made to the record in the current process cycle. `Rec` and `xRec` records have all the same fields in the same structure as the table to which they relate.

DELETE function

The purpose of the `DELETE` function is to delete existing data records. The syntax for `DELETE` is as follows:

```
[BooleanValue :=] Record.DELETE ( [ TriggerControlBoolean ] )
```

If the `BooleanValue` is not used and `DELETE` fails, the process will terminate with an error statement. You should handle any detected error or terminate the process, as appropriate, under the control of your C/AL code.

The `TriggerControlBoolean` value is a `TRUE` or `FALSE` entry, which controls whether or not the table's `OnDelete` trigger fires when this `DELETE` occurs. The default value is `FALSE`. If you let the default `FALSE` prevail, you run the risk of not performing error checking that the table's designer assumed would be run when a record was deleted.

MODIFYALL function

MODIFYALL is the high-volume version of the MODIFY function. If you have a group of records in which you wish to modify one field in all of them to the same new value, you should use MODIFYALL. MODIFYALL is controlled by the filters that apply at the time of invoking. MODIFYALL does not do any error checking, such as checking for an empty set or enforcing referential integrity.

The other choice for doing a mass modification would be to have a FIND-NEXT loop in which you modified each record one at a time. The advantage of MODIFYALL is that it allows the system to optimize processing for the volume update.

The syntax for MODIFYALL is as follows:

```
Record.MODIFYALL (FieldToBeModified,NewValue  
[,TriggerControlBoolean ] )
```

The TriggerControlBoolean value is a TRUE or FALSE entry, which controls whether or not the table's OnModify trigger fires when this MODIFY occurs. The default value is FALSE.

In a typical situation, a filter or series of filters would be applied to a table followed by the MODIFYALL function. A simple example where we are going to reassign all the Territory Codes for a particular Salesperson to NORTH is as follows:

```
CustRec.SETRANGE("Salesperson Code",'DAS');  
CustRec.MODIFYALL("Territory Code",'NORTH',TRUE);
```

DELETEALL function

DELETEALL is the high volume version of the DELETE function. If you have a group of records that you wish delete, use DELETEALL. The other choice would be to have a FIND-NEXT loop in which you delete each record one at a time. The advantage of the DELETEALL is that it allows the system to optimize processing for the volume deletion.

The syntax for DELETEALL is as follows:

```
Record.DELETEALL ( [,TriggerControlBoolean] )
```

The TriggerControlBoolean value is a TRUE or FALSE entry that controls whether or not the table's OnDelete trigger fires when this DELETE occurs. The default value is FALSE. If the TriggerControlBoolean value is TRUE, then the OnDelete trigger will fire for each record deleted. In that case, there is no speed advantage for DELETEALL versus the use of a FIND-DELETE-NEXT loop.

In a typical situation, a filter or series of filters would be applied to a table followed by the `DELETEALL` function, similar to the preceding example. Like `MODIFYALL`, `DELETEALL` respects the filters that have been set and does not do any referential integrity error checking.

Filtering

We have talked about the fact that the filtering capabilities built into NAV provide a significant additional level of power to the system. This power is available to the users and to the developer as well. It is true that other systems provide filtering of data for inquiry, reporting, or analysis. But a few other systems have filtering implemented as pervasively as does NAV nor do they have it tied to the detailed retention of historical data. The result of NAV's features is that even the most elementary implementation of NAV includes very powerful data analysis capabilities for end user use.

You as the developer should appreciate the fact that you cannot anticipate every need of any user, let alone anticipate every need of every user. For that reason, you should give the user as much freedom as you can. Wherever feasible, the user should be given the opportunity to apply their own filters so that they can determine the optimum selection of data for their particular situation. On the other hand, freedom, here as everywhere, is a double-edged sword. With the freedom to decide just how to segment one's data, comes the responsibility for figuring out what constitutes a good segmentation to address the problem at hand.

As you, the experienced systems designer and developer, presumably have considerable insight into good ways to analyze and present the data, it may be best for you to provide some predefined selections. In some cases, the data structure means a very limited set of options make sense (maybe just one). As a result, you should often provide one or more specific accesses to data (pages and/or reports), but then, if possible, also allow more sophisticated user access to manipulate the data creatively on their own.

When applying filters by using any of the options, be very conscious of the table key that will be active when the filter takes effect. In a table containing a lot of data, filtering on a field that is not represented very high in the currently active key may result in a poor (or very poor) response time for the user. In the same vein, in a system suffering from a poor response time during processing, you should first investigate the relationships of active keys to applied filters.

SETRANGE function

SETRANGE allows you to set a simple range filter on your data. The syntax is as follows:

```
Record.SETRANGE (Field [,LowValue] [,HighValue] );
```

If both the optional parameters are omitted, any filtering that was previously applied to `Record.Field` will be cleared. In fact, this is the recommended way for clearing filters on a single field.

If only one parameter is specified, it becomes both the high and low range values. In other words, you will be filtering on a single value in this field. If you specify both a low- and high-range value, the filter will be logically the same as: `LowValue` less than or equal to `Field` less than or equal to `HighValue`. If you happen to specify a `HighValue` that is less than the `LowValue`, you will exclude all data, resulting in selecting an empty set.

SETFILTER function

SETFILTER allows you to apply any Filter expression that could be created manually, including various combinations of ranges, C/AL operators, and even wild cards.

SETFILTER syntax is as follows:

```
Record.SETFILTER ( Field, FilterString [, FilterValue1], . . . );
```

`FilterString` can be a literal such as `'1000..20000'` or `'A*|B*|C*'`. Optionally, you can use variable tokens in the form of `%1`, `%2`, `%3`, and so forth, representing variables (but not operators) `FilterValue1`, `FilterValue2`, and so forth to be substituted in the filter string at runtime. This construct allows you to create filters whose data values can be defined dynamically at runtime. A new `SETFILTER` cancels any previous filtering on the field prior to setting the new filter.

A pair of `SETFILTER` examples follow:

```
CustRec.SETFILTER("Salesperson Code", 'KKS' | 'RAM' | 'CDS');  
CustRec.SETFILTER("Salesperson Code", '%1|%2|%3', SPC1, SPC2, SPC3);
```

If `SPC1` equals `'KKS'`, `SPC2` equals `'RAM'`, and `SPC3` equals `'CDS'`, these two examples would have the same result. Obviously the second option allows flexibility not provided by the first option.

COPYFILTER and COPYFILTERS functions

These functions allow copying the filters of a single field or all the filters on a record (table) and applying those filters to another record. The syntaxes follow:

```
FromRecord.COPYFILTER (FromField, ToRecord.ToField)
ToRecord.COPYFILTERS (FromRecord)
```

Note that the `COPYFILTER` structure begins with the `FromRecord` variable while that of `COPYFILTERS` begins with the `ToRecord` variable.

GETFILTER and GETFILTERS functions

These functions allow you to retrieve the filters on a single field or all the filters on a record (table) and assign the result to a text variable. The syntaxes are as follows:

```
ResultString := FilteredRecord.GETFILTER (FilteredField)
ResultString := FilteredRecord.GETFILTERS
```

The text contents of the `ResultString` will contain an identifier for each filtered field and the currently applied value of the filter. `GETFILTERS` is often used to retrieve the filters on a table and print them as part of a report heading. The `ResultString` will look similar to the following:

```
Customer:: No.: 10000..999999, Balance: >0
```

MARK function

A mark on a record is an indicator that disappears when the current session ends and which is only visible to the process, that is, setting the mark. The `MARK` function sets the mark. The syntax is as follows:

```
[BooleanValue := ] Record.MARK ( [SetMarkBoolean] )
```

If the optional `BooleanValue` and assignment operator (`:=`) is present, the `MARK` function will give you the current Mark status (`TRUE` or `FALSE`) of the `Record`. If the Optional `SetMarkBoolean` parameter is present, the `Record` will be Marked (or unmarked) according to that value (`TRUE` or `FALSE`). The default for `SetMarkBoolean` is `FALSE`. The `MARK` functions are a little tricky to use, so it should be used carefully and only when a simpler solution is not readily available. MARKing records can cause significant performance problems, so use this feature sparingly.

CLEARMARKS function

CLEARMARKS clears all the marks from the specified record (that is, from the particular instance of the table in this instance of the object). The syntax is as follows:

```
Record.CLEARMARKS
```

MARKEDONLY function

MARKEDONLY is a special filtering function that can apply a mark-based filter.

The syntax for MARKEDONLY is as follows:

```
[BooleanValue := ] Record.MARKEDONLY ( [SeeMarkedRecordsOnlyBoolean] )
```

If the optional BooleanValue parameter is defined, it will be assigned a value TRUE or FALSE to tell you whether or not the special MARKEDONLY filter is active. Omitting the BooleanValue parameter, MARKEDONLY will set the special filter depending on the value of SeeMarkedRecordsOnlyBoolean. If that value is TRUE, it will filter to show only marked records; if that value is FALSE, it will remove the marked filter and show all records. Though it may not seem logical, there is no option to see only the unmarked records. The default value for SeeMarkedRecordsOnlyBoolean is FALSE.

RESET function

This function allows you to RESET (that is, clear) all filters that are currently applied to a record. The syntax is as follows:

```
FilteredRecord.RESET;
```

RESET also sets the current key back to the Primary Key, removes any marks, and clears all internal variables in the current instance of the record.

Filter Groups

Filter Groups are (not surprisingly) groups of filters that are applied. Filter Groups are numbered from 0 to 255. Several of the Filter Groups are utilized by NAV for internal usage. You should be careful not to use these groups most of the time. See the **C/SIDE Reference Guide Help** for FILTERGROUP for more information.

The FILTERGROUP function is used to change or determine the current active filtergroup. Its syntax is:

```
[CurrentFilterGroup] := Record, FILTERGROUP ( [NewFilterGroupNo] )
```

Using just the `Record, FILTERGROUP ([NewFilterGroupNo])` portion sets the active Filter Group.

You can change the active Filter Group to which filter commands will apply until the next Filter Group change. When filters are applied, the combination of all filters set in all groups is applied. One use of a Filter Group would be to assign a filter which the user cannot see is present or change. The code could change the Filter Group, set a special filter, and then return the active Filter Group to its original state. This could be used to apply special application-specific permissions to a particular system function, such as filtering out access to the business owner's expense accounts to anyone expect the outside auditor.

InterObject communication

There are several ways for communicating information between objects during NAV processing.

Communication via data

The most widely used and simplest communication method is through data tables. For example, the table `No. Series` is the central control for all document numbers. Each object that assigns numbers to a document (for example, `Order`, `Invoice`, `Shipment`, and so on) accesses the `No. Series` table for the next number to use, and then updates the `No. Series` table so that the next object needing to assign a number to the same type of document will have the updated information.

Communication through function parameters

When an object calls a function in another object, information is generally passed through the calling and return parameters. The calling and return parameter specifications were defined when the function was originally coded. The generic syntax for a function call is as follows:

```
[ReturnValue := ] FunctionName ( [ Parameter1 ] [ ,Parameter2 ] ,...)
```

The rules for including or omitting the various optional fields are specific to the local variables defined for each individual function. As a developer, when you design the function, you define the rules and thereby determine just how communications with the function will be handled. It is obviously important to define complete and consistent parameter passing rules prior to beginning a development project.

Communication via object calls

Sometimes you need to create an object which in turn calls other objects. You may simply want to allow the user to be able to run a series of processes and reports but only enter the controlling parameters once. Your user interface object is to be responsible for invoking the subordinate objects after having communicated setup and filter parameters.

There is a significant set of standard functions designed for various modes and circumstances of invoking other objects. Examples of these functions are `SETTABLEVIEW`, `SETRECORD`, and `GETRECORD` (there are others as well). There are also instances where you will need to build your own data passing function.

In order to properly manage these relatively complex processes, you need to be familiar with the various versions of `RUN` and `RUNMODAL` functions. You will also need to understand the meaning and effect of a single instance or multiple instances of an object. Briefly, key differences between invoking a page/form or report object from within another object via `RUN` versus `RUNMODAL` are as follows:

- `RUN` will clear the instance of the invoked object every time the object completes, which means that all of the internal variables are initialized. This clearing behavior does not apply to a codeunit object; state will be maintained across multiple calls to `RUN`.
- `RUNMODAL` does not clear the instance of the invoked object, so internal global variables are not re-initialized each time the object is called. The object can be re-initialized by using `CLEAR(Object)`.
- `RUNMODAL` does not allow any other object to be active in the same user session while it is running, whereas `RUN` does.

Covering these topics in more detail is too advanced for this book, but once you have mastered the material covered here, you should study the information in the **C/SIDE Reference Guide** Help and reference manuals relative to this topic.

Using the new knowledge

It's time to use some of the knowledge that we have gained in this and preceding chapters. We'll do that in a practical manner by further developing the ICAN system.

A development challenge for you

We are going to create a new report on giving by Donors. One of the ways that ICAN rewards Donors each year is by giving them a status Recognition Level based on the amount they have donated in the previous year. One measurement the Fund Raising Manager uses is to evaluate the YTD level of giving of each Donor relative to their Recognition Level achieved the previous year. Our new report will assist in that review.

Our first task is to create a rough report layout for concept purposes.

Donor Recognition Level Giving Analysis report layout					
[Report Headings-----]					
[Column Headings-----]					
Donor ID	Donor Name	Address City Region, Post Code	Country	Estimated Gift Value	Past Recognition Level
Totals by Recognition Level:					
Recognition Level Descr	Count of Gifts	Total Value of Gifts			

This layout is a starting point for creating a first version of the report. This is a version that, once we've got the basics working, we will enhance on our own for a later self-directed exercise.

Creating more ICAN test data

In our specific case, we are working with a testing database which just has a few records in it that we've created along the way. For this report, we would like to have a larger set of Gift Ledger data, but without the pain of manually entering a significant volume of test data. Therefore, our first step will be to create a Processing Report that will read the existing Gift Ledger, then copy those records several times back into the same table, thus giving us a larger volume of test data. After we've cloned the initial data, if we wish to provide more variety to the contents of various data fields, it will be much easier to step through the Gift Ledger and make changes.

"Donor Giving" report design

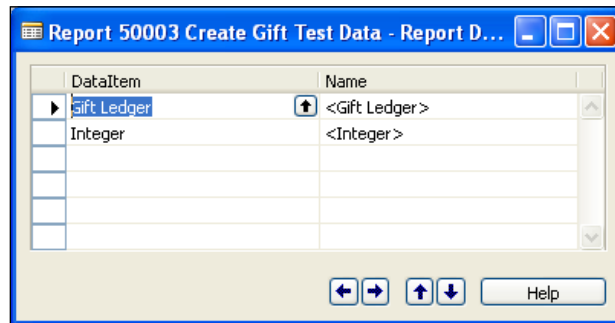
The design we will use is relatively simple. First, we will read the existing data into a temporary table. Then we will copy that data back into the permanent table in the database as many times as it takes to create the desired data volume.

We will define two Data Items. One will be the Gift Ledger table. Our code will let the Report Read loop walk us through the existing data which we will copy into the temporary table. For the second Data Item, we will use the system table Integer. This table is simply the sequence of numbers 1, 2, 3, 4, and so on. Based on the size of the incoming dataset and the desired size of the dataset to be created, we will filter the Integer table to the number of data copies we want to create. If you have the urge to work on creating this on your own before we work on it together, now is the time to go do that.

Beginning development

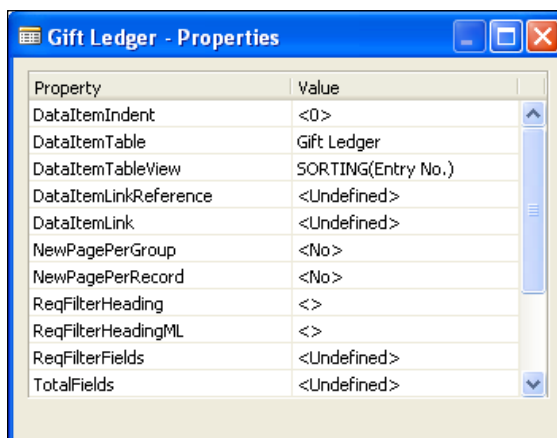
The first thing to do is start up a new blank report and define the DataItems. An aspect of this routine that is different from previous ones that we have reviewed is that the second DataItem is not subordinate to the first DataItem. In other words, the processing flow for this routine will be to step all the way through the Gift Ledger table, then step through the defined portion of the Integer table.

As soon as we get our DataItems defined, we should save and compile the shell of the new report, assigning it to Report ID 50003 with the **Name** of Create Gift Test Data. Then, as our development efforts move along, we will save and compile on a regular basis. This is the safest way to check our work and to back it up in case C/SIDE should terminate unexpectedly.

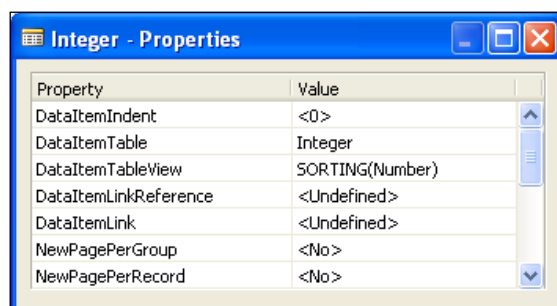


Eliminating the Request form/page

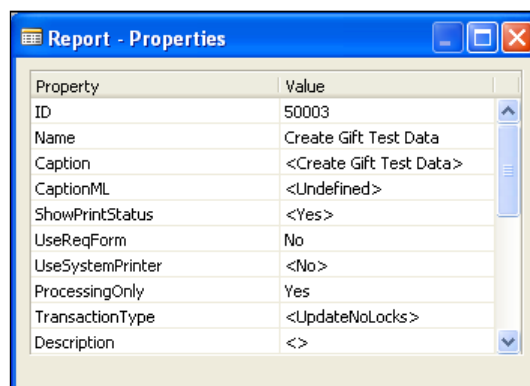
Because this is (probably) a one-time use routine which we will run within the Classic Client (that is, under developer control), we will set it up so it doesn't display any Request screen options at all. We can do this by specifying the sort keys to be used in the Data Items.



The Integer table has only one field. That field, Number, is also its key.

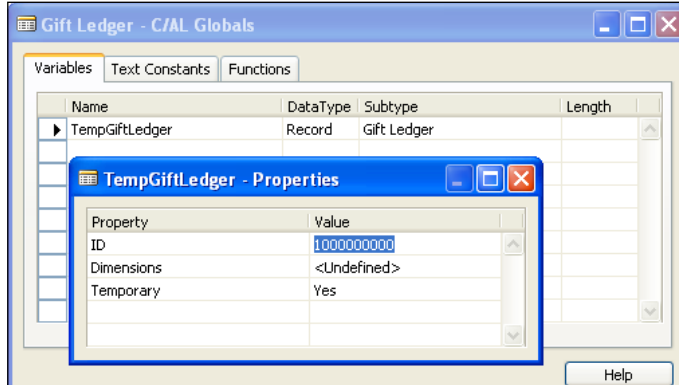


The final piece of eliminating the Request Form/Page is to set the appropriate option in the Report Properties, **UseReqForm**, to **No**. We will also set the **ProcessingOnly** option to **Yes** as shown below:



Working storage definition

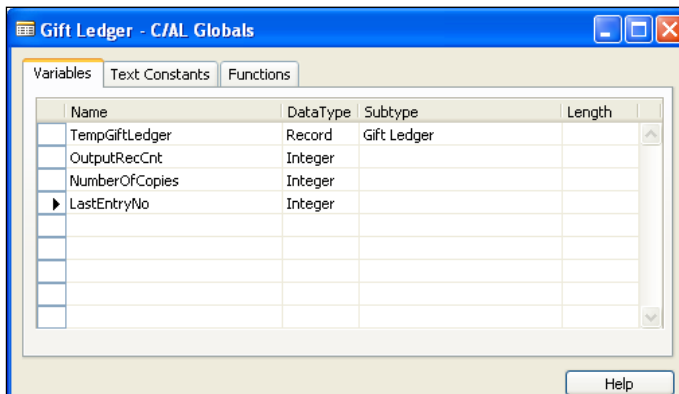
As we know we want to use a temporary table to store a copy of the data because it exists before our processing. Our Global Variable is **TempGiftLedger**. Once defined, we access its properties and set the **Temporary** property to **Yes**.



Following the definition of the temporary table, we consider what other Global Variables we may need. At this point we realize that we will need at least the following:

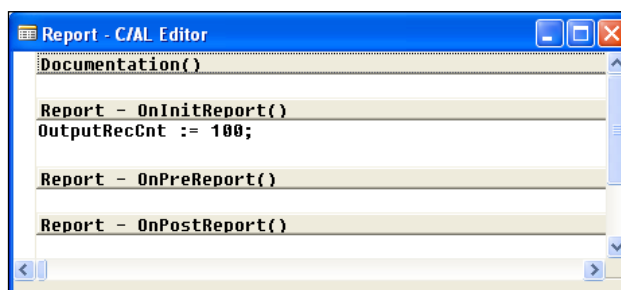
- A count of the output records we want to generate
- An integer that represents how many times we will need to copy the input dataset to generate the target number of output records
- Noting that the Primary Key for the output table is **Entry No.**, we will define an integer variable to increment and assign as **Entry No.** to the next record being written out

Those are defined as shown in the the following C/AL Globals screenshot:



Defining the C/AL code

Based on the fact that this is a one-time use process, we will simply define the minimum number of new output records that we want to generate. The following screenshot shows it being defined as quantity 100. If this routine is to be used multiple times or as a model, then this quantity should be entered through a Request Option Form/Page, not specified in the code. If you want to create a set of Gift Ledger test data that is different than 100 more than the original set, you should change this number accordingly.

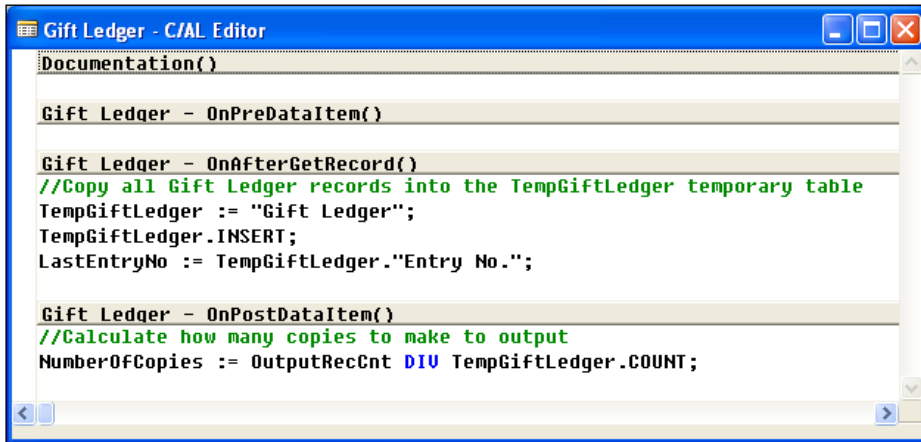


The first step in processing is to copy all of the Gift Ledger records from the permanent table into the temporary table. We will take advantage of the Data Item read loop to read the Gift Ledger. In the **OnAfterGetRecord** trigger, we simply assign in the incoming record to the TempGiftLedger record. In a rather inelegant piece of code, we save the Entry No. of each record as it passes by, so that we will have the number for the last record when we exit the loop.

When the incoming Gift Ledger has been completely passed, control will pass to the **OnPostDataItem** trigger. There we will calculate the number of copies of the input data to write out to give us at least the number of additional records we specified. This calculation could have been more sophisticated (by using both DIV and MOD) to give us an exact number, but then the output loop would have to break in the middle of a data pass.

If the number of records in the Gift Ledger is 12 and the target OutputRecCnt is 100, then this formula would be interpreted as **100 DIV 12 = 8 = NumberOfCopies**. This calculation could have just as easily been placed at the OnPreDataItem trigger; it doesn't matter where in the dataflow as the values used don't change during processing.

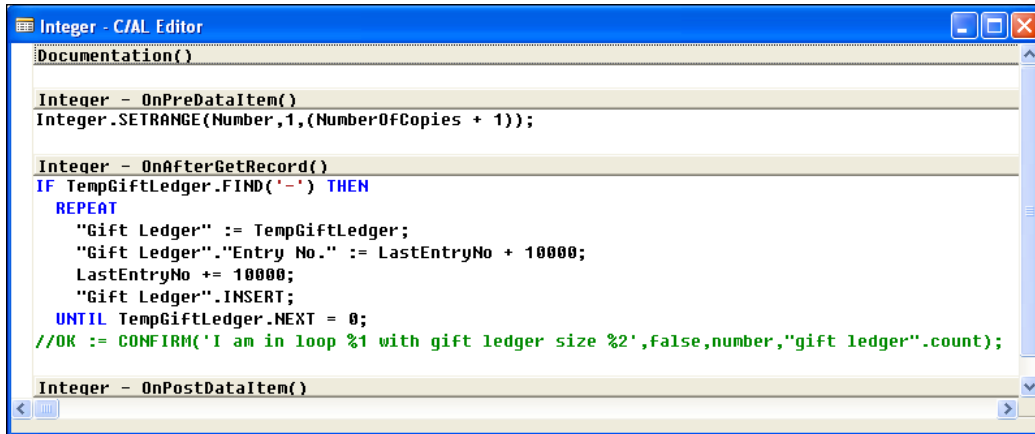
The simpler approach is illustrated, but after the initial version of this object is working, it would be a good exercise for you to enhance the code to output the exact number of additional records specified by `OutputRecCnt`.



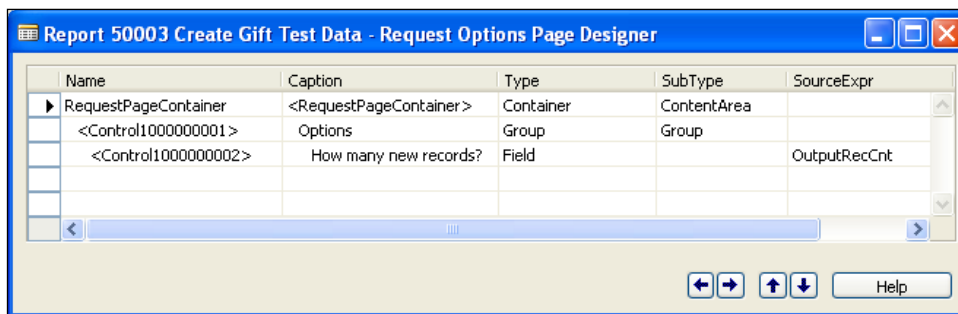
Once the original data has been stored, we can now move to the reading/processing loop we've designed to be driven by the System table, `Integer`. The first step is to set the limit on the `Integer` table based on the `NumberOfCopies` figure we calculated previously. That limit is set by doing a `SETRANGE` on the `Integer` key field, `Number`, to be one more than the `DIV` value of `NumberOfCopies`. Our cycle through the `Integer` table will be a simple counting of 1, 2, 3, 4, ...`NumberOfCopies`+1.

In each loop through `Integer`, we want to write out a full copy of the Gift Ledger data stored in our temporary table, `TempGiftLedger`. That process requires use of a basic `FIND-NEXT` loop to process the temporary table. The processing loop consists of copying the saved record to the output record, updating the `Entry No.` field and `INSERT`ing the new data in the permanent table as shown in the following image.

During the debugging of a process like this one, it's often very helpful to have some information display during the processing to let us know what's happening. The `CONFIRM` statement that has been commented out (preceded with two slashes) provides just such tracking information. This particular statement, if activated, would show which `Integer` loop was just processed and what the number of records are in the original Gift Ledger table. In this case, the developer response to the `CONFIRM` statement wouldn't matter. Processing continues. But it would obviously be easy to extend this code to allow the developer to terminate processing at this point as an option. Note that the `CONFIRM` statement, as coded, requires a Boolean variable (here that is the variable `OK`) which needs to be defined in the Local (or, if necessary, Global) variables.



If we were to modify this object to allow the user to enter the `OutputRecCnt` value, we would create a Request Options Page for that entry, as illustrated in the following Page Designer screenshot:

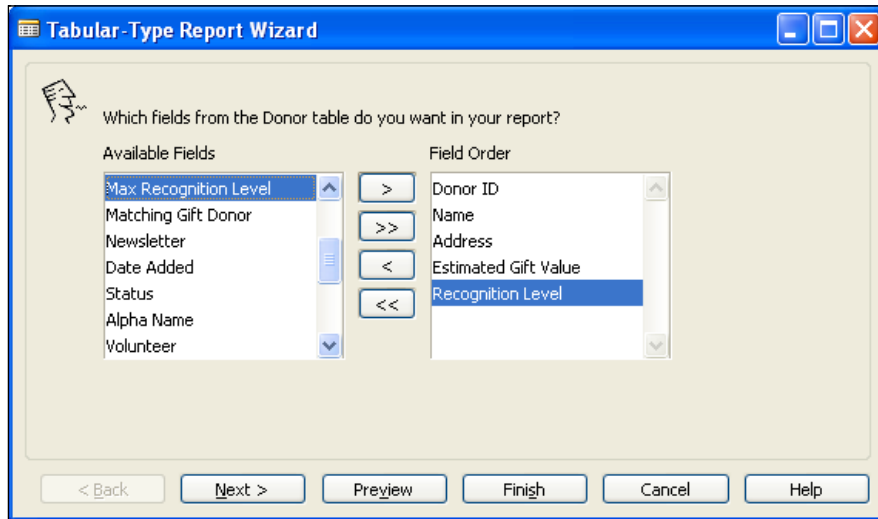


Developing the Donor Recognition Status report

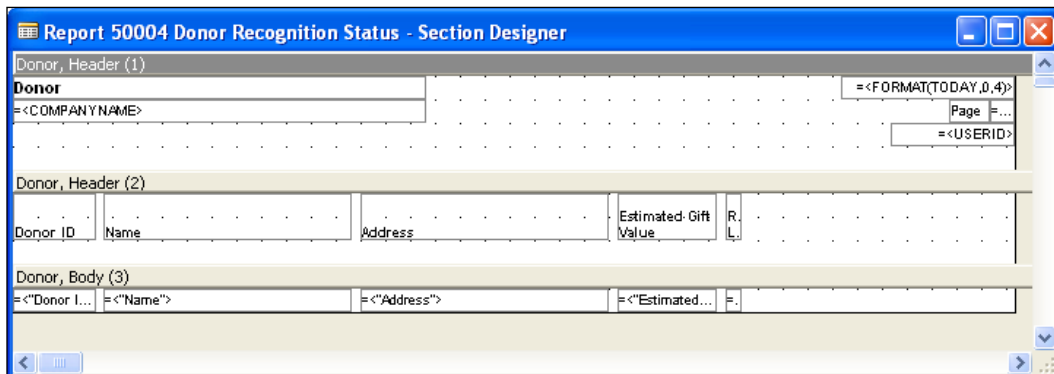
As with other production reports, it makes sense to start this process by using the Report Designer Wizard.

Using the Report Wizard

Our new report will be based on the Donor table and we choose to create a tabular-type Report using the Wizard. By using the original report design layout as our guide, we can choose the matching data fields from the Donor table. The following screenshot illustrates that set of choices:



As we proceed from the preceding screen by clicking on **Next >**, we choose to have our data sorted by **Donor**, not to be Grouped, to have a List Style report, then Finish the Wizard processing. As soon as we get into the Report Designer, we open up the Section Designer (**View | Sections**) to see something like the following image. We have a good start on a layout that agrees with our original Word document rough draft layout.

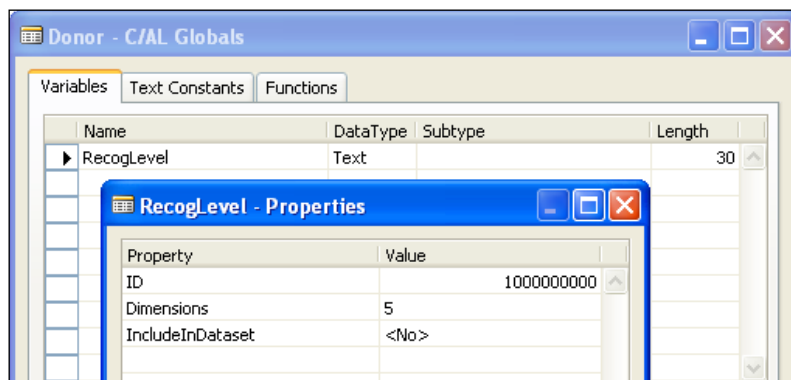


Beginning the C/AL coding for the report

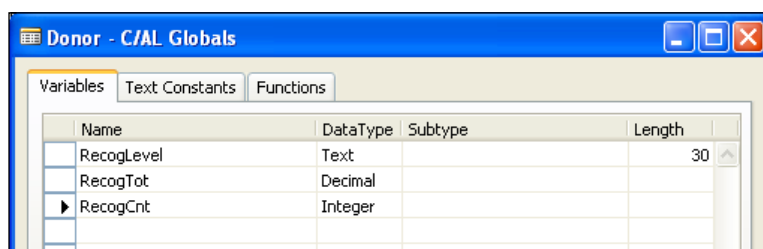
With a reasonable layout beginning in hand, we can now start on the construction of the C/AL side of the report. These tasks may not be done in a particularly logical order other than the sequence in which we think about what is needed.

The first thing that catches our attention is the need for places to store the five levels of totals that we are going to offer at the end of our report. We want three columns of information in those totals, the Description of each possible value for the Recognition Level data element, the count of the number of gifts received at each Level and the total value of the gifts received at each Level.

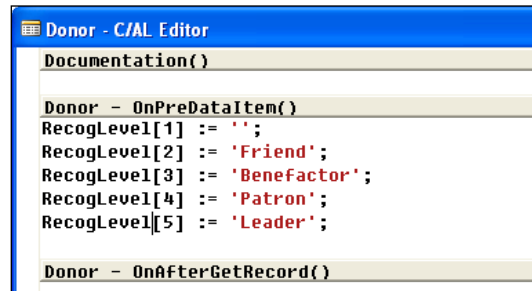
There are several ways in which we could design the data structure and supporting logic to collect our data for these totals. Let's choose a simple method, defining three Global variables, each defined with the **Dimensions** property set to 5 (five), the number of total lines we need. Each of these variables will now be a one by five array. Each element will need to be referenced with the variable named in subscripted format (for example, **RecogLevel[1]** for the first member of the array).



The result is the definition of the three dimensioned variables (RecogLevel, RecogCnt, and RecogAmt) that we know we need.



Simply because it helps us keep track of what we're doing, next we write the code to load the **RecogLevel** description array. Because the Recognition Level field in the Donor table is an Option field, we check that Option definition to determine what the available values are for the Recognition Level. We load the **RecogLevel** array variables with those values, as shown in the following image. This C/AL code is placed in the **OnPreDataItem** trigger because it only needs to be executed once.

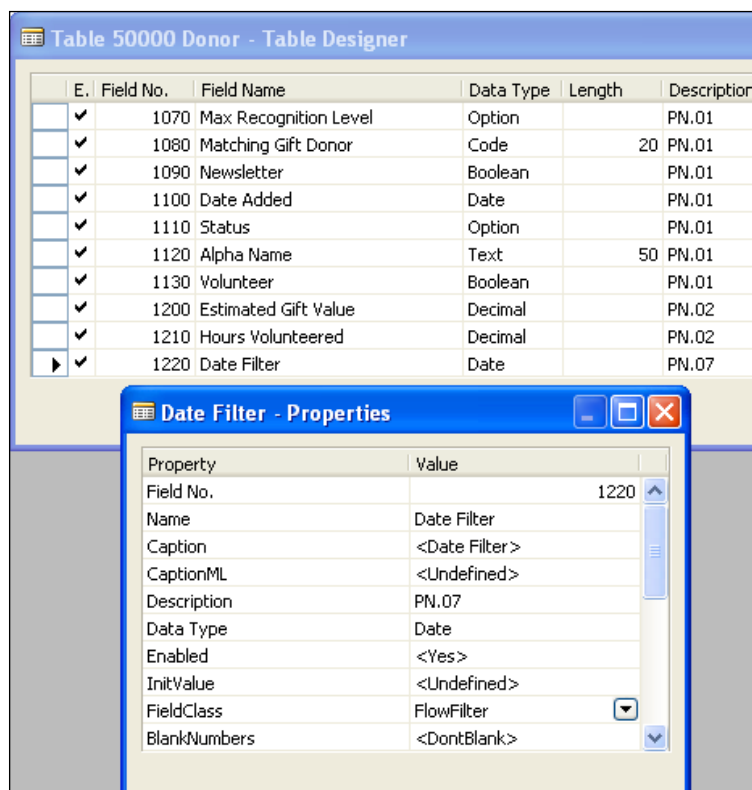


If we wanted to be more clever (and more general), we would write code to loop through the Recognition Level options, by using the **FORMAT** function to create the text strings to be stored. This approach would continue working if the set of defined options were later changed, plus it will support multilanguage use. You might want to try this approach as a self-directed exercise.

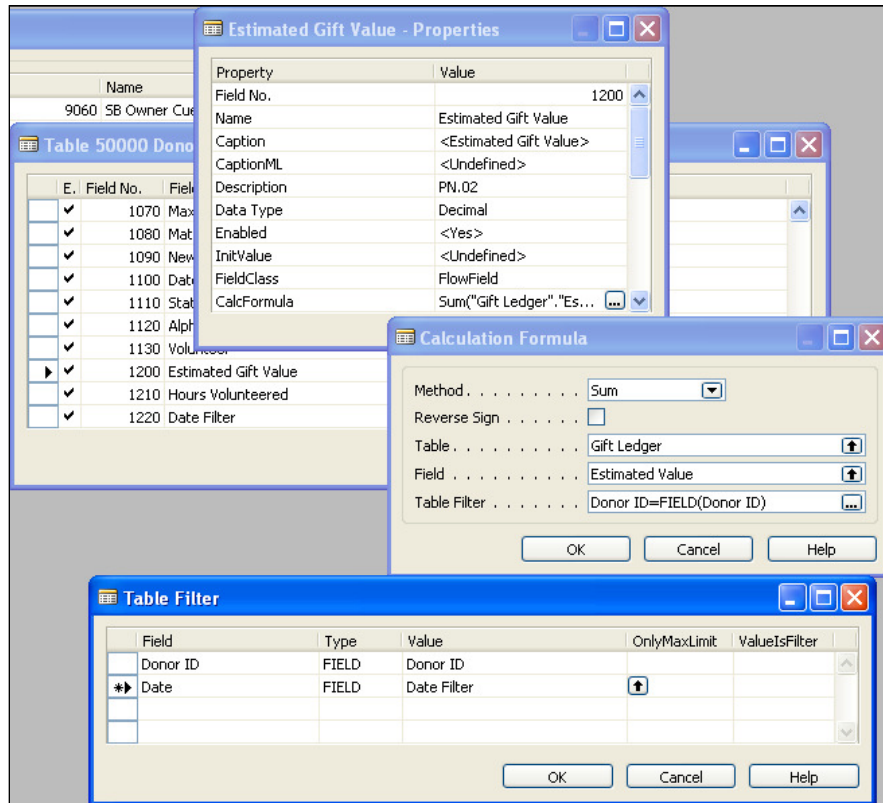
Retrofitting date filtering capability

One of our original report design goals is to allow this report to apply date filters on the data. As we start considering the next step of the coding, that of reading the Donor records and calculating the Gift totals (the **Estimated Gift Value** field), we realize that we don't have a way to filter the subordinate Gift Ledger table through the Donor table. We could calculate the totals by directly accessing the Gift Ledger, but we prefer the approach that is more like standard NAV structure. That involves accessing the Customer Ledger through the Customer table.

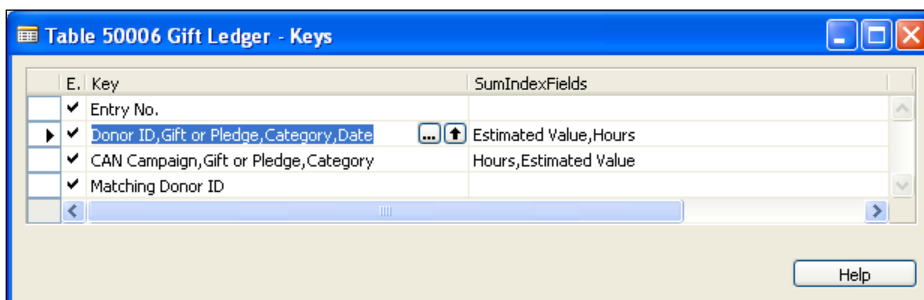
To support this type of filtering, we need to add a Date Filter field of **FieldClass** FlowFilter to the **Donor** table as shown in the next screenshot:



Then we need to add the Date Filter field to the Donor. "Estimated Gift Value" **FlowField** definition. This allows a Date Filter value to be applied to the calculation of the Estimated Gift Value. The result is an Estimated Gift Value for the defined period of time.

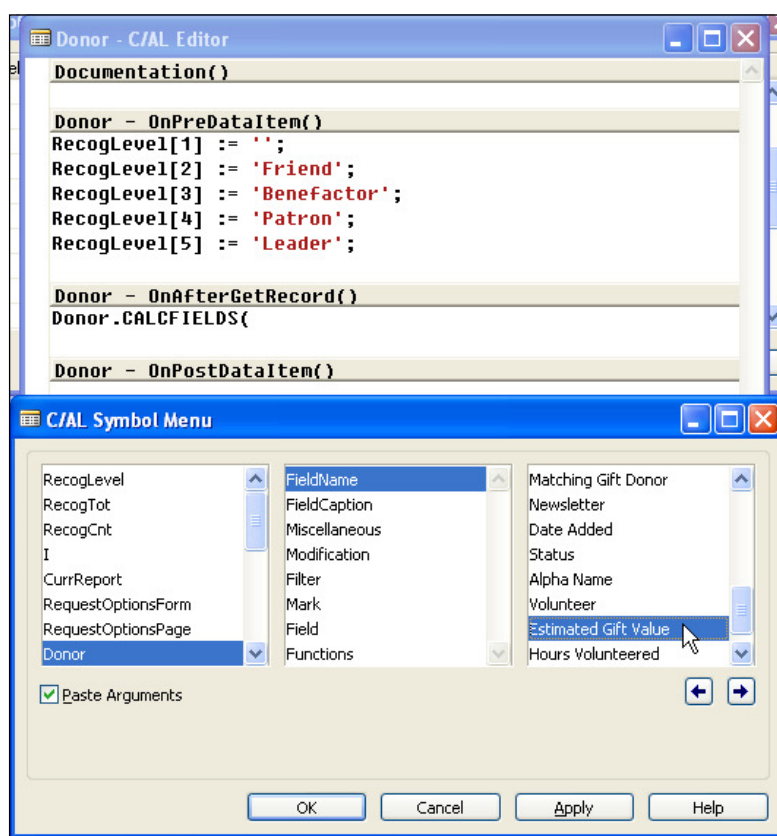


Finally, we need to add the "Gift Ledger".Date field, on which the Date Filter is to be applied, to the key that the Estimated Value SIFT total is tied.



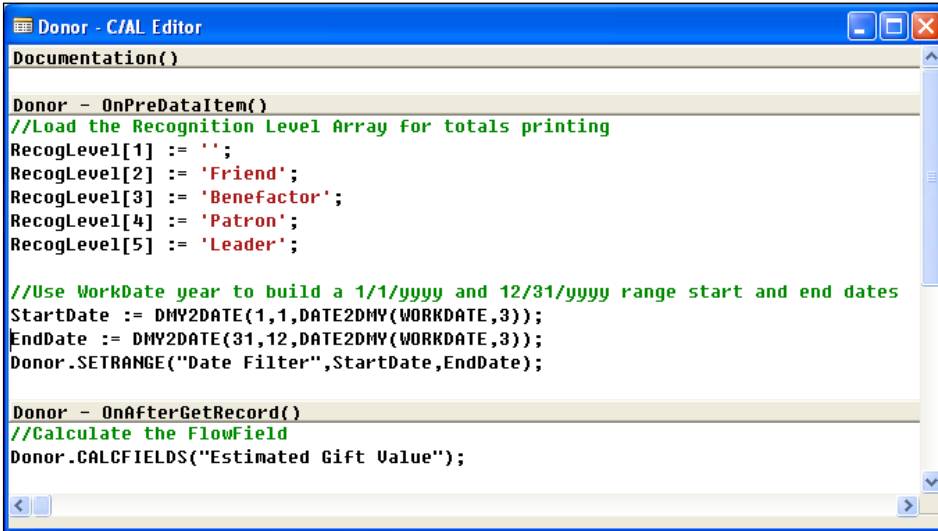
Adding code for CALCFIELDS processing

We return to the C/AL coding for our report, specifically to the CALCFIELDS line of code for which we just did a lot of preparation. One tool that makes writing C/AL code much easier is the C/AL Symbol Menu. In this case, we want to do the CALCFIELDS on the gift value field in the Donor record. By using the C/AL Symbol Menu, we can just point and click to get the right field name and spelling. The field name is always pasted in fully qualified (in this case that would be Donor."Estimated Gift Value"). In situations where the full qualification (that is, table name qualifier) isn't required, you can edit the pasted value down to just the field name (in this case "Estimated Gift Value").



As we complete the CALCFIELDS line of code, we realize that we need to calculate the values that are going to make up the date filter that will limit the Gift Ledger entries to be included in the calculated values. As the date filter values only need to be calculated once for the report, we decide we can put that code in the `OnPreDataItem` trigger.

This report is going to be based on annual giving data, that is, from January 1 to December 31. We will use the Work Date as the source of the year. That way, the user can just change the Work Date to report the analysis based on a different calendar year, but by default it will be based on the year of the current Work Date. The C/AL code in the following screenshot shows various Date Functions being used to build the StartDate and EndDate values. Then the Date() Filter can be assigned the date range values. StartDate and EndDate will also need to be defined as Date variables in C/AL Globals.



```
Donor - C/AL Editor
Documentation()

Donor - OnPreDataItem()
//Load the Recognition Level Array for totals printing
RecogLevel[1] := '';
RecogLevel[2] := 'Friend';
RecogLevel[3] := 'Benefactor';
RecogLevel[4] := 'Patron';
RecogLevel[5] := 'Leader';

//Use WorkDate year to build a 1/1/yyyy and 12/31/yyyy range start and end dates
StartDate := DMY2DATE(1,1,DATE2DMY(WORKDATE,3));
EndDate := DMY2DATE(31,12,DATE2DMY(WORKDATE,3));
Donor.SETRANGE("Date Filter",StartDate,EndDate);

Donor - OnAfterGetRecord()
//Calculate the FlowField
Donor.CALCFIELDS("Estimated Gift Value");
```

If you want to allow for the incredibly small probability that the WORKDATE could change between the two lines of code in which it is used, a "for sure" version of the EndDate calculation would be:

```
EndDate := DMY2DATE(31,12,DATE2DMY(StartDate,3));
```

Adding code to print addresses

Looking back at our draft report layout, we see that our report includes printing addresses. Address lines one and two are simple, but the third line of the address information is a build up of three fields from the data record. As is often the case, there are multiple ways to handle this. The following code allows for any of the fields to be blank and still build a properly formatted third address line.

```
//Build the third address line from individual fields
IF "Country/Region Code" <> '' THEN
    AddressLine3 := ' ' + "Country/Region Code";
IF "Post Code" <> '' THEN
    AddressLine3 := "Post Code" + AddressLine3;
IF "State/Province" <> '' THEN
    IF AddressLine3 <> '' THEN
        AddressLine3 := "State/Province" + ' ' + AddressLine3
    ELSE AddressLine3 := "State/Province";
```

To calculate the number of cash gifts in the desired date range, we decide to take the direct approach of counting the number of records in the properly filtered Gift Ledger. We define a C/AL Local (to the **Donor - OnAfterGetRecord()** trigger) for the Gift Ledger and apply the filters for the Donor, Gifts only, the Date range, and a Gift value greater than zero (as the non-cash gifts don't have an Estimated Gift Value).

We can code to calculate the record count and add that to the appropriate totals array entry, using the Donor record's Recognition Level option value to create the proper subscript. Because the first option value in an option field is stored as 0 (zero), we add 1 (one) to the option to get a usable subscript. We calculated the "Estimated Gift Value" FlowField earlier in our code. Now we add it to the `RecogTot["Recognition Level" + 1]` array element in the same manner.

```
//Filter for money Gifts for this Donor in the date range
GiftLedger.SETCURRENTKEY("Donor ID","Gift or Pledge",Category,Date);
GiftLedger.SETRANGE("Donor ID","Donor ID");
GiftLedger.SETRANGE(Date,StartDate,EndDate);
GiftLedger.SETFILTER("Estimated Value",>0);
//Add the Number of Gifts Count to the Count total for Donor's Recognition Level
RecogCnt[("Recognition Level" + 1)] += GiftLedger.COUNT;

//Add this Donor's Total Gift Amount to the total for Donor's Recognition Level
RecogTot[("Recognition Level" + 1)] += "Estimated Gift Value";
```

We should be done, or at least nearly done, with the C/AL coding. We have to turn our attention to the Sections of this report. Remember that all data elements that are going to be available for use by the Visual Studio Report Designer must be defined in the Sections Designer of the Classic Report Designer. All the fields that are defined in the Sections at this point are those that were created by the Report Designer Wizard.

There are a number of fields that we want to report which are not yet defined in Sections. Those include the additional two address lines and all the total lines. So our next step is to add those as Controls in the Section Designer. Even though we are not designing our report with the goal of running it in the Classic Client, it's almost as easy in this instance to add the remaining fields laid out so that they will give an acceptable report result for the Classic Client as well as for the Role Tailored Client.

One feature of using the Sections Designer that will prove handy here is the ability to add a Data Item Footer Section to the report. The Header Section was created by the Wizard, but as we didn't specify any totaling by the Wizard, a Footer Section was not created. You can create a Data Item Footer Section within the Section Designer by selecting **Edit | New**, specifying the lowest level **Data Item** (in this case **Donor**, the only Data Item), and choosing the **Footer** option. That will add a Donor Footer as you see in the next image. Once the new Footer Section exists, Label Controls can be added for the three total columns, accompanied by appropriately placed Textbox Controls for our totaling array data elements. The Label Controls will need their Caption properties defined and the Textbox Controls will need their SourceExpr properties defined properly. After the basic report has been successfully tested, you may want to return and do some more formatting. More likely, you will focus that formatting effort on the VS RD layout because that is where the capability exists to create a much better looking report and that is where you plan to run the report for production use.

Report 50004 Donor Recognition Status - Section Designer

Donor, Header (1)

Donor	=<COMPANYNAME>	=<FORMAT(TODAY,0,4)>
		Page =...
		=<USERID>

Donor, Header (2)

Donor ID	Name	Address	Estimated Gift Value	Recognition Level
----------	------	---------	----------------------	-------------------

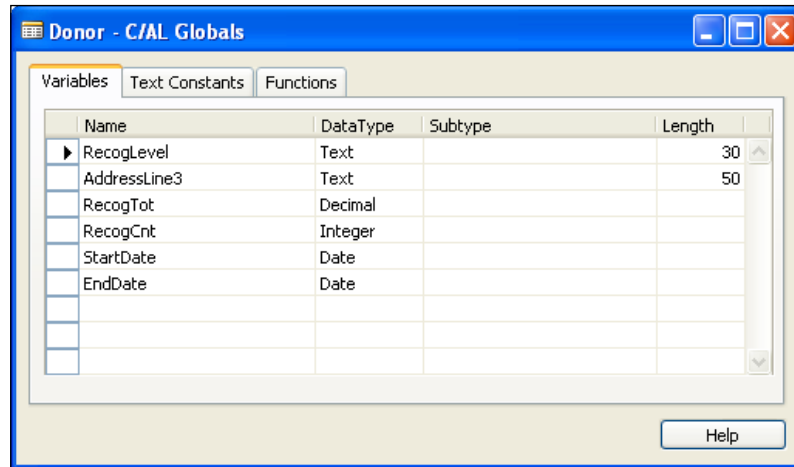
Donor, Body (3)

=<Donor I...>	=<Name>	=<Address>	=<Estimated...>	=<Recognition Le...>
		=<City>		
		=<Address Line3>		

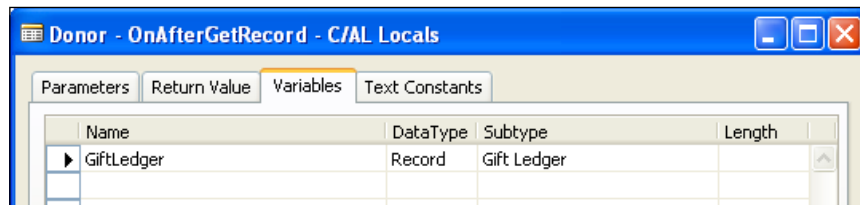
Donor, Footer (4)

Recognition Level	Cash Gift Count	Cash Gift Total
=<RecogLevel[1]>	=<RecogCnt[1]>	=<RecogTot[1]>
=<RecogLevel[2]>	=<RecogCnt[2]>	=<RecogTot[2]>
=<RecogLevel[3]>	=<RecogCnt[3]>	=<RecogTot[3]>
=<RecogLevel[4]>	=<RecogCnt[4]>	=<RecogTot[4]>
=<RecogLevel[5]>	=<RecogCnt[5]>	=<RecogTot[5]>

Just for review purposes, we'll take a quick look at the list of variables that we have defined as C/AL Globals and Locals. The Globals are shown following screenshot:



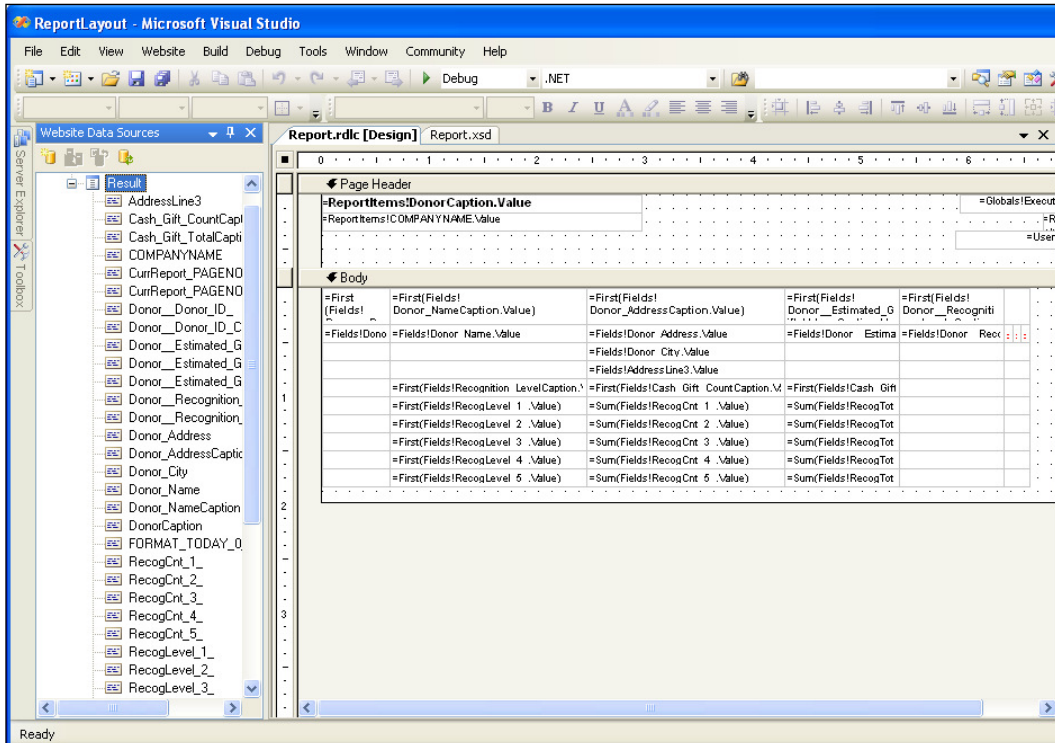
Then the Local variable that was defined for the **OnAfterGetRecord** trigger:



Once again, as we've done on a regular basis during the construction of our report, we will save and compile. But this time, we are going to **Run** and **Preview** the report using the **Run** button on the Object Designer Form. The resulting output should look similar to the following:

Donor				
CRONUS International Ltd.				
Donor ID	Name	Address	Estimated Gift Value	Recognition Level
1001	Juan O'Hara	1123 Riviera Way Duluth MN 55701 US	0.00	Friend
1002	Karen Johnson Fashions, LTD	19W243 Rue De La Paz Nice 06000 FR	0.00	Friend
1003	Hans Statenbacker Research	131-A Koelnerstrasse Solingen 42699 GE	0.00	Benefactor
1004	John Jones	915 Msta View Fargo ND 58103 US	100.00	Friend
1005	Marchen Smits	10314 S. Hoyne Ave Columbus OH 43201 US	547.50	Benefactor
1006	Schmidt Haus AG	Einsiedlerplatz 14 Vienna 1050 AT	0.00	Benefactor
Recognition Level		Cash Gift Count	Cash Gift Total	
		0	0.00	
Friend		1	100.00	
Benefactor		2	547.50	
Patron		0	0.00	
Leader		0	0.00	

Now that we have a basic working report in the Classic Client, let's transform the report for our target production environment, the Role Tailored Client. Once again, highlight the report object, **Design** it, then select **Tools | Create Layout Suggestion**. After a few moments of processing, we should see our report layout presented in the VS RD, looking much like the following screenshot.



Later, when your time permits, you can come back and work on making it more flexible and more attractive, perhaps adding some new data elements or graphical features. But now, we just want to test our new report in the RTC. We exit, save, save again, and compile. Use the Windows Run command:

```
DynamicsNAV:////runreport?report=50004.
```


The result of the RTC test of our new report, shown in the following image, looks very much like the Classic Client test. But now it's time for your creativity to be applied. You should look for ways to make this a better report and, in the process, learn more about C/AL coding and VS RD report development.

Print Preview

Donor Recognition Status

1 of 1 100%

Donor 4/30/200

CRONUS International Ltd.

Donor ID	Name	Address	Estimated Gift Value	Recognition Level
1001	Juan O'Hara	1123 Riviera Way Duluth MN 55701 US	0.00	Friend
1002	Karen Johnson Fashions, LTD	19W243 Rue De La Paz Nice 06000 FR	0.00	Friend
1003	Hans Statenbacker Research	131-A Koelnerstrasse Solingen 42699 GE	0.00	Benefactor
1004	John Jones	915 Vista View Fargo ND 58103 US	100.00	Friend
1005	Marchen Smits	10314 S. Hoyne Ave Columbus OH 43201 US	547.50	Benefactor
1006	Schmidt Haus AG	Einsiedlerplatz 14 Menna 1050 AT	0.00	Benefactor
Recognition Level		Cash Gift Count	Cash Gift Total	
		0	0.00	
Friend		3	300.00	
Benefactor		4	1,095.00	
Patron		0	0.00	
Leader		0	0.00	

Summary

In this chapter, we covered a number of practical tools and topics regarding C/AL coding and development. We started with reviewing methods and then we dived into a long list of functions that you will need on a frequent basis.

We began this chapter by covering development assisted by use of the C/AL Symbol Menu, followed by a discussion of development documentation. Then we covered a variety of selected data-centric functions, including some for computation and validation, data conversion, and date handling. Next, we reviewed functions that affect the flow of logic and the flow of data, including FlowFields and SIFT, Processing Flow Control, Input and Output, and Filtering.

In the next chapter, we will move from the details of the functions to the broader view of C/AL development including models of code usage in the standard product code, integration into the standard NAV code, and some debugging techniques.

Review questions

1. The C/AL Symbol Menu can be used for several of the following purposes. Choose three:
 - a. Find applicable functions
 - b. Paste highlighted entries directly into code
 - c. Test coded functions
 - d. Use entries as a template for function syntax and arguments
 - e. Translate text constants into a support language
2. Documentation cannot be integrated into in-line C/AL code. True or False?
3. The TESTFIELD function can be used to assign new values to a variable. True or False?
4. The VALIDATE function can be used to assign new values to a variable. True or False?
5. The WORKDATE value can be set to a different value from the System Date. True or False?
6. Which three of the following are valid date related NAV functions?
 - a. DATE2DWY
 - b. CALCDATE
 - c. DMY2DATE
 - d. DATE2NUM
7. A FlowField requires two of the following. Which two?
 - a. A record key
 - b. SQL Server database
 - c. A SumIndexField
 - d. A Decimal variable
8. Which of the following is **not** a valid C/AL flow control combination? Choose one:
 - a. REPEAT - UNTIL
 - b. DO - UNTIL
 - c. CASE - ELSE
 - d. IF - THEN

9. Which of the following functions should be used within an `OnAfterGetRecord` trigger to end processing just for a single iteration of the trigger? Choose one:
 - a. `EXIT`
 - b. `BREAK`
 - c. `QUIT`
 - d. `SKIP`
10. Which of the following formats of `MODIFY` will cause the table's `OnModify` trigger to fire? Choose one:
 - a. `MODIFY`
 - b. `MODIFY (TRUE)`
11. `SETRANGE` is often used to clear all filtering from a single field. True or False?
12. `RESET` is used to clear the current sort key setting from a record. True or False?

8

Advanced NAV Development Tools

The universe is full of magical things, patiently waiting for our wits to grow sharper – Eden Phillpotts

The only way of discovering the limits of the possible is to venture a little way past them into the impossible – Arthur C. Clarke

Having studied the foundation basics of C/AL, followed by a review of intermediate functions and structures, it's time to get into more advanced topics. As NAV is so flexible and suitable for addressing many different problem types, advanced NAV topics range far and wide.

We have three key goals in this chapter. One is to gain an overall view of NAV as an application software system. We're not going to study its functional operation, but gain a basic understanding of the process flow of the system. We also want to have a good understanding of the structural "style" of the software, so that our enhancements are designed for a better fit.

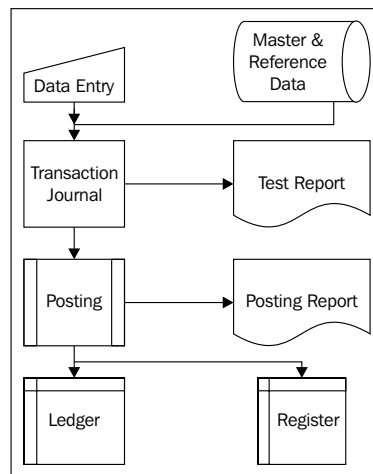
The second goal is to review some of the higher level components of the NAV system that are available as resources to us, as designers and developers, allowing us to accomplish more at less cost. These resources include standard objects that we can call, features that we can build on, and tools that allow us to build or integrate totally new functionality.

The third goal of this chapter is to learn about many of the debugging tools and techniques available to the NAV developer. As it has been pointed out, "Without programmers, there are no bugs." As we are all developers and therefore a primary source of bugs, we need to be knowledgeable about the tools we can use to stamp out those bugs. Fortunately, NAV has a good arsenal of such tools.

NAV process flow

Primary data such as sales orders, purchase orders, production orders, financial transactions, job transactions, and so on flow through the NAV system as follows:

- **Initial Setup:** Entry of essential Master data, reference data, control and setup data. Much of this preparation is done when the system (or a new application) is first set up for production use.
- **Transaction Entry:** Transactions are entered into a `Journal` table; data is preliminarily validated as it is entered, master and auxiliary data tables are referenced as appropriate. Entry can be manual keying, an automated transaction generation process, or an import function which brings transaction data in from another system.
- **Validate:** Provide for additional test validations of data prior to submitting the batch to Posting.
- **Post:** Post the Journal Batch, completing transaction data validation, adding entries as appropriate to one or more Ledgers, including perhaps a register and a document history.
- **Utilize:** Access the data via Forms and/or Reports of various types as appropriate. At this point, total flexibility exists. Whatever tools are available and are appropriate for users' needs should be used. There are some very good tools built into NAV for data manipulation, extraction, and presentation. In the past, these capabilities were considered good enough to be widely accepted as full **Online Analytical Processing (OLAP)** tools.
- **Maintenance:** Continue maintenance of Master data, reference data, and setup and control data, as appropriate. The loop returns to the beginning of this data flow sequence.



The preceding image provides a simplified picture of the flow of application data through a NAV system. Many of the transactions types have additional reporting, more ledgers to update, or even auxiliary processing. However, this is the basic data flow followed whenever a Journal and Ledger table are involved.

Data preparation

Prepare all the Master data, reference data, and control and setup data. Much of this preparation is done initially, when an application is first set up for production usage.

Naturally, this data must be maintained as new Master data becomes available, as various system operating parameters change, and so on. The standard approach for NAV data entry allows records to be entered that have just enough information to define the primary key fields, but not necessarily enough to support processing. This allows a great deal of flexibility in the timing and responsibility for entry and completeness of new data.

This system design philosophy allows initial and incomplete data entry by one person, with validation and completion to be handled later by someone else. For example, a sales person might initialize a new customer entry with name, address, and phone number, saving the entry with just the data entered to which they have access. At this point, there is not enough information recorded to process orders for this new customer.

At a later time, someone in the accounting department can set up posting groups, payment terms, and other control data that should not be controlled by the sales department. This additional data may make the new customer record ready for production use. As in many instances data comes into an organization on a piecemeal basis, the NAV approach allows the system to be updated on an equally piecemeal basis providing a flexible user friendliness many accounting-oriented systems lack.

Transactions entry

Transactions are entered into a `JOURNAL` table; data is preliminarily validated as it is entered, master and auxiliary data tables are referenced as appropriate.

NAV uses a relational database design approach that could be referred to as a "rational normalization". NAV resists being constrained by the concept of a normalized data structure, where any data element appears only once. The NAV data structure is normalized so long as that principle doesn't get in the way of processing speed. Where processing speed or ease of use for the user is improved by duplicating data across tables, NAV does so.

At the point where Journal transactions are entered, a considerable amount of data validation takes place. Most, if not all, of the validation that can be done is done when a Journal entry is made. These validations are based on the combination of the individual transaction data plus the related Master records and associated reference tables (for example lookups, application or system setup parameters, and so on). Here also you find the philosophy of allowing entries, which are incomplete and not totally ready for processing, to be made.

Testing and Posting the Journal batch

Any additional validations that need to be done to ensure the integrity and completeness of the transaction data prior to being Posted are done either in pre-Post routines or directly in the course of the Posting processes. The actual Posting of the Journal batch occurs when the transaction data has been completely validated. Depending on the specific application function, when Journal transactions don't pass muster during this final validation stage, either the individual transaction is bypassed while acceptable transactions are Posted, or the entire Journal Batch is rejected until the identified problem is resolved.

The Posting process adds entries as appropriate to one or more Ledgers and sometimes a document history. When a Journal Entry is Posted to a Ledger, it becomes a part of the permanent accounting record. Most data cannot be changed or deleted once it is resident in a Ledger except by a subsequent Posting process.

During the Posting process, Register tables are also updated showing what transaction entries (by ID number) were posted when and in what batches. This adds to the transparency of the NAV application system for audits and analysis.

In general, NAV follows the standard accounting practice of requiring Ledger corrections to be made by Posting reversing entries, rather than deletion of problem entries. The overall result is that NAV is a very auditable system, a key requirement for a variety of government, legal, and certification requirements for information systems.

Accessing the data

The data in a NAV system can be accessed via Pages and/or Reports of various types as appropriate, providing total flexibility. Whatever tools are available to the developer or the user, and are appropriate, should be used. There are some very good tools in NAV for data manipulation, extraction, and presentation. Among other things, these include the SIFT/Flowfield functionality, the pervasive filtering capability (including the ability to apply filters to subordinate data structures), and the Navigate function. NAV 2009 added the ability to create page parts for graphing,

with a wide variety of predefined graph page parts included as part of the standard distribution. You can create your own chart parts as well, but that discussion is outside the scope of this book. There is an extended discussion and some tools available in the NAV Blog community.

There are a number of methods by which data can be pushed or pulled from an NAV database for processing and presentation outside NAV. Several are discussed briefly in Chapter 9, *Extend, Integrate, and Design – into the Future*. These allow use of more sophisticated graphical displays, or the use of other specialized data analysis tools such as Microsoft Excel or various **Business Intelligence (BI)** tools.

Ongoing maintenance

As with any database-oriented application software, ongoing maintenance of Master data, reference data, and setup and control data is required, as appropriate. Of course at this point, the cycle of processing returns to the first step of the data flow sequence, Data Preparation.

Role Center pages

One of the key additions to Dynamics NAV in the NAV 2009 version is the concept of a Role Tailored user experience centered around user function defined Role Centers. The intent of the Role Tailored approach is to provide a single point of access into the system for each user. That point of access (the Role Center) focuses on the tasks that define users' jobs throughout the day, while providing easy access to other permitted functions only a click or two away.

Included in the NAV system as distributed by Microsoft are 21 different Role Center pages, identified for user roles such as Bookkeeper, Sales Manager, Shop Supervisor, Purchasing Agent, and so on. One of the critical tasks of implementing a new system will be to analyze the work flow and responsibilities of the system's intended users and configure Role Centers to fit the users.

In many cases, the supplied Role Centers can be used out of the box or with minimal configuration. On occasion, it will be necessary for you to create new Role Centers. Even then, most of the time, you will be able to begin your job with a copy of an existing Role Center Page, which you will then modify as required. In any of these cases, it is important to understand the structure of the Role Center Page and how it is built.

Role Center structure

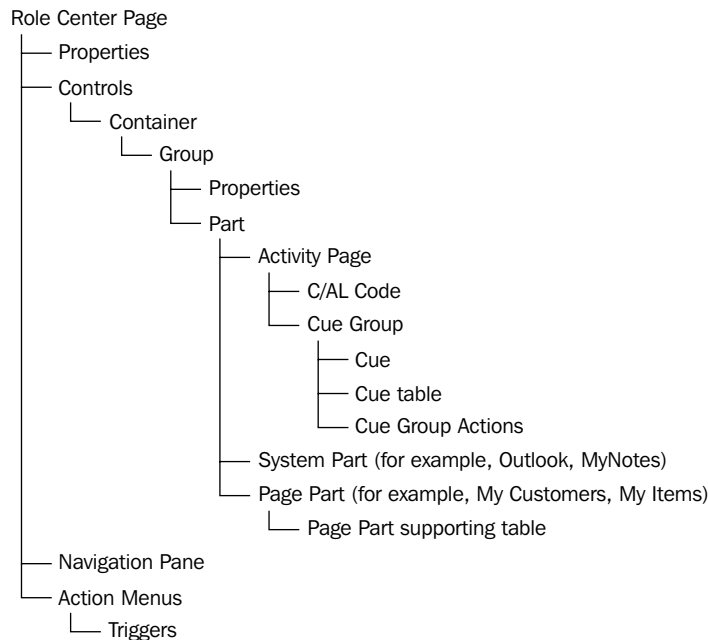
The following screenshot shows the Small Business Owner RC (aka President – Small Business) Role Center Page (Page 9020) with labels identifying the various pieces we deal with as developers.

The screenshot displays the Microsoft Dynamics NAV Role Center for a user named 'Small Business Owner'. The interface is divided into several sections, each with a label pointing to it:

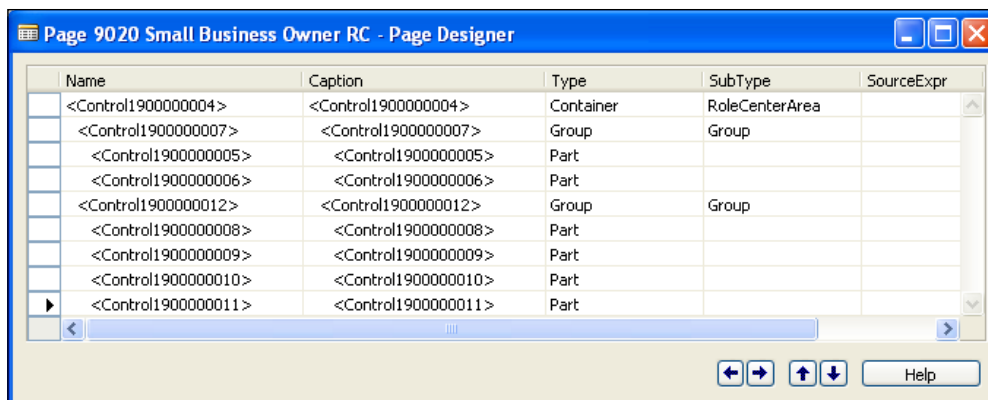
- Navigation Pane:** Located on the left, it contains a tree view of the role center's structure, including 'Sales Orders', 'Sales Quotes', 'Sales Blanket Orders', 'Sales Invoices', 'Sales Return Orders', 'Sales Credit Memos', 'Items', 'Customers', 'Item Journals', 'Sales Journals', and 'Cash Receipt Journals'.
- Cues in Groups:** This label points to the 'For Release' section, which displays a group of cues (Sales Quotes, Sales Orders) with counts (0, 20) and a 'New Sales Quote' button.
- Role Activities:** This label points to the 'Sales Orders Released Not Shipped' section, which displays a group of cues (Ready to Ship, Partially Shipped, Delayed) with counts (6, 0, 15) and a 'Navigate' button.
- List Parts:** This label points to the 'My Customers' table, which lists customer information (Customer No., Phone No., Name) and includes a search bar.
- System Parts:** This label points to the 'My Items' table, which lists item information (Item No., Description, Unit Price) and includes a search bar.
- User's Outlook:** This label points to the 'Microsoft Outlook' section, which displays a list of tasks (Mail, Calendar, Tasks) and a 'New Sales Return Order' button.
- Cue Group Actions:** This label points to the 'New Sales Credit Memo' button, which is part of the 'Returns' section.

The bottom of the screen shows the user's name 'CRONUS International Ltd.', the date 'Thursday, January 27, 2011', and the time '10:10'.

A general representation of the structure of a Role Center Page is shown in the following outline:

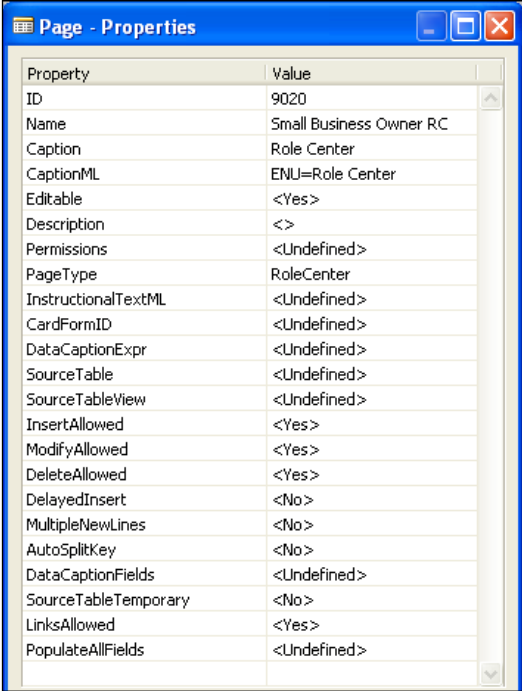


In order to understand the construction of a Role Center Page, we will dissect Page 9020, Small Business Owner RC. Our goal will be to understand what the component parts are and how they fit together. We want to be prepared to either modify an existing Role Center or create a new one by through either "clone and modify" or building from scratch. Our first step is to find Page 9020 in the Page section of the Object Designer and design the page to see the following screenshot:



This picture of a page should look pretty familiar, as it is similar in overall structure to the pages we've designed previously. There is a Container control of **SubType** `RoleCenterArea`. This is required for a Role Center page. There are two Group Controls which represent the two columns (left and right) of the Role Center page display. Each group contains several parts, each of which shows up individually in the Role Center display.

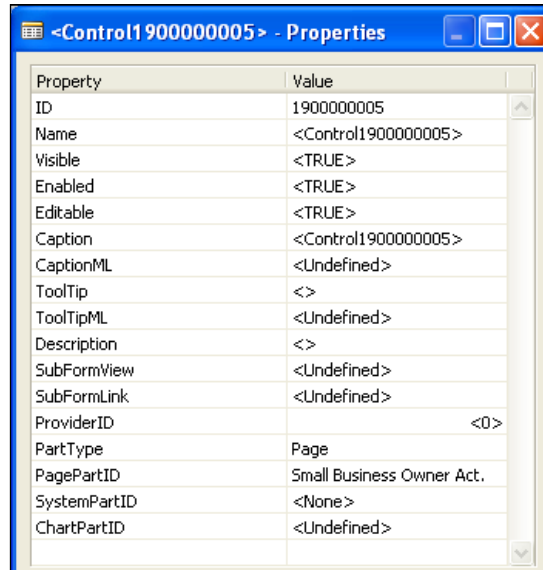
Let's take a look at the properties for the Role Center page. As we did previously, we access the page Properties by highlighting the first blank line on the Page Designer form (that is the line below all of the defined controls), then click on the Properties icon, or right-click and choose the **Properties** option, or, click on **View | Properties** or press *Shift + F4*. The Properties for the Role Center page, in this case the Small Business Owner Role Center, will be displayed as shown in the following screenshot. The only things different at this level from a Card page is that the **PageType** is `RoleCenter`, and there is no **Source Table**.



Property	Value
ID	9020
Name	Small Business Owner RC
Caption	Role Center
CaptionML	ENU=Role Center
Editable	<Yes>
Description	<>
Permissions	<Undefined>
PageType	RoleCenter
InstructionalTextML	<Undefined>
CardFormID	<Undefined>
DataCaptionExpr	<Undefined>
SourceTable	<Undefined>
SourceTableView	<Undefined>
InsertAllowed	<Yes>
ModifyAllowed	<Yes>
DeleteAllowed	<Yes>
DelayedInsert	<No>
MultipleNewLines	<No>
AutoSplitKey	<No>
DataCaptionFields	<Undefined>
SourceTableTemporary	<No>
LinksAllowed	<Yes>
PopulateAllFields	<Undefined>

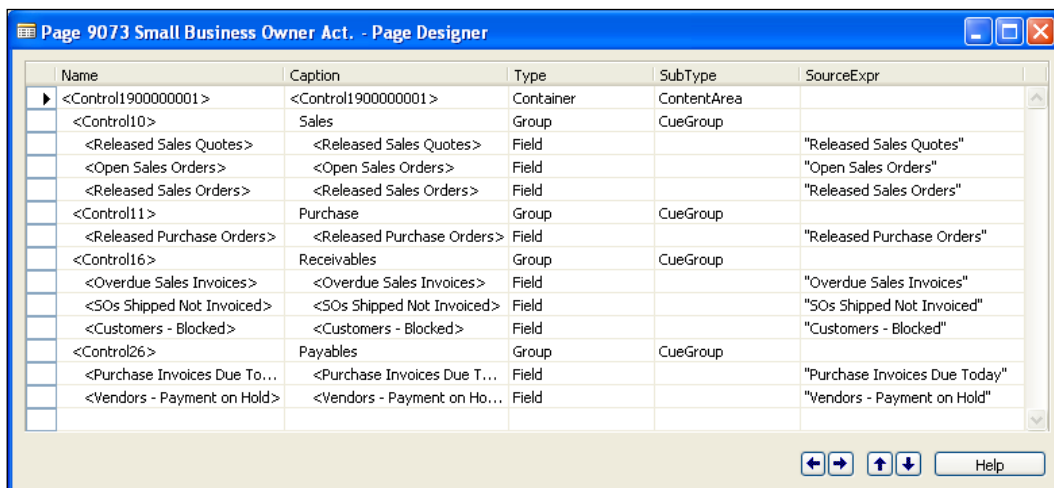
Role Center activities page

As the Group Control has no underlying code or settings to consider, the first Control to examine is the first Part Control. Examining the Properties for that Control, we see the information shown in the following screenshot. The **PagePartId** refers to the Page object Small Business Owner Act.



Cue Groups and Cues

If we shift our focus to the referenced page: Page 9073 – Small Business Owner Act, and design the page, we see the layout shown in the following screenshot. Comparing the controls we see in that layout to those of the Role Center screenshot, we can clearly see this Page Part is the source of the material in the **Role Activities** section of the Role Center Page. There are four **CueGroup** Controls – **Sales**, **Purchase**, **Receivables** and **Payables**. Within each CueGroup, there are the **Field Controls** for the individual **Cues**.

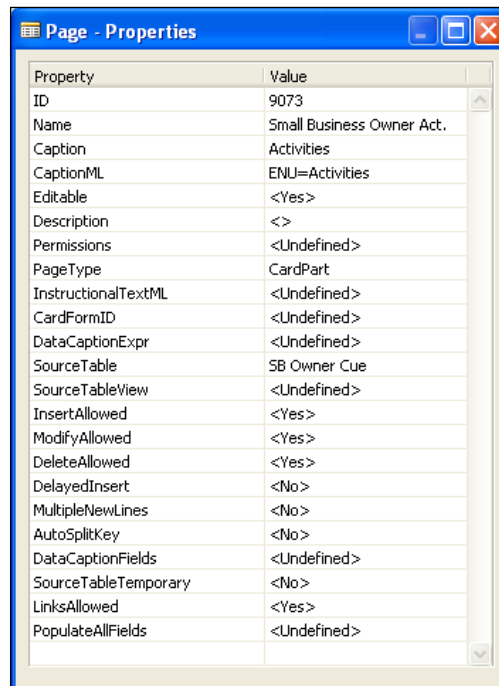


An individual Cue is displayed as an iconic shortcut to a filtered list. The size of the stack of papers in the Cue icon represents the number of records in that list. The actual number of entries is also displayed as part of the icon (see the **Released Purchase Orders** example in following screenshot). The purpose of a Cue is to provide a focus on and easy access to a specific user task. The set of Cues is intended to represent the full set of primary activities for a user, their Role.



Cue source table

When we take a look at the Properties of the Small Business Owner Act. Page (see the following screenshot), we see this is a **PageType** of CardPart tied to **SourceTable** SB Owner Cue:



Following the designated path, we **Design** the referenced table, **SB Owner Cue**. What we see there (see the following screenshot) is a simply structured table, with

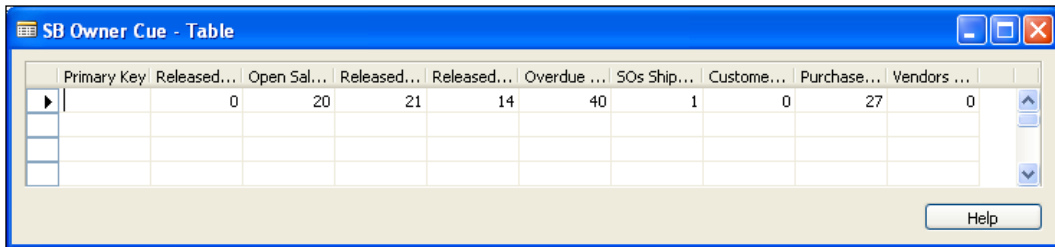
an integer field for each of the Cues that were displayed in the Role Center we are analyzing. There is also a key field and two fields identified as Date Filters:

E.	Field No.	Field Name	Data Type	Length	Description
<input checked="" type="checkbox"/>	1	Primary Key	Code	10	
<input checked="" type="checkbox"/>	2	Released Sales Quotes	Integer		
<input checked="" type="checkbox"/>	3	Open Sales Orders	Integer		
<input checked="" type="checkbox"/>	4	Released Sales Orders	Integer		
<input checked="" type="checkbox"/>	5	Released Purchase Orders	Integer		
<input checked="" type="checkbox"/>	6	Overdue Sales Invoices	Integer		
<input checked="" type="checkbox"/>	7	SOs Shipped Not Invoiced	Integer		
<input checked="" type="checkbox"/>	8	Customers - Blocked	Integer		
<input checked="" type="checkbox"/>	9	Purchase Invoices Due Today	Integer		
<input checked="" type="checkbox"/>	10	Vendors - Payment on Hold	Integer		
<input checked="" type="checkbox"/>	20	Due Date Filter	Date		
<input checked="" type="checkbox"/>	21	Overdue Date Filter	Date		

When we display the properties of one of these integer fields, the field named **Released Sales Quotes**, we find it is a `FlowField` providing a Count of the Sales Quotes with a Status equal to Released. In fact, if we inspect each of the other integer fields in this table, we will find a similar `FlowField` setup. Each is defined to fit the specific Cue to which it's tied. It is obvious, thinking about what the Cues show and how `FlowFields` work, that this is a fairly simple, direct method of providing the information necessary to support Cue displays.

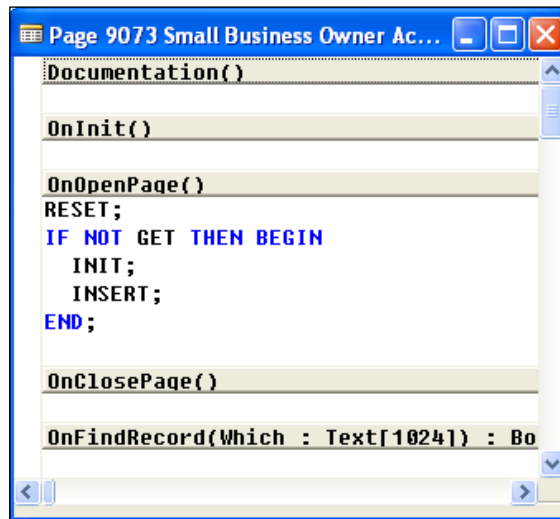
Property	Value
Field No.	2
Name	Released Sales Quotes
Caption	Released Sales Quotes
CaptionML	ENU=Released Sales Quotes
Description	<>
Data Type	Integer
Enabled	<Yes>
InitValue	<Undefined>
FieldClass	FlowField
CalcFormula	Count("Sales Header" WHERE (Document Type=CONST(Quote),Status=FILTER(Released)))
BlankNumbers	<DontBlank>
BlankZero	<No>
SignDisplacement	<0>
AutoFormatType	<0>
AutoFormatExpr	<>
CaptionClass	<>
Editable	<Yes>
MinValue	<>
MaxValue	<>
NotBlank	<No>
ValuesAllowed	<>
TableRelation	<Undefined>
ValidateTableRelation	<Yes>
ExtendedDatatype	<None>

As a confirmation of our analysis, we can **Run** the SB Owner Cue table to view the data there. There is one record with the Integer FlowFields we saw in Design mode. The Counts that are displayed match those that are shown in the Role Center display. As this table is designed to contain only a single record, the key field is blank. And, in addition to the FlowFields, there are also a couple of FlowFilter fields used in the definition of the date based FlowFields.



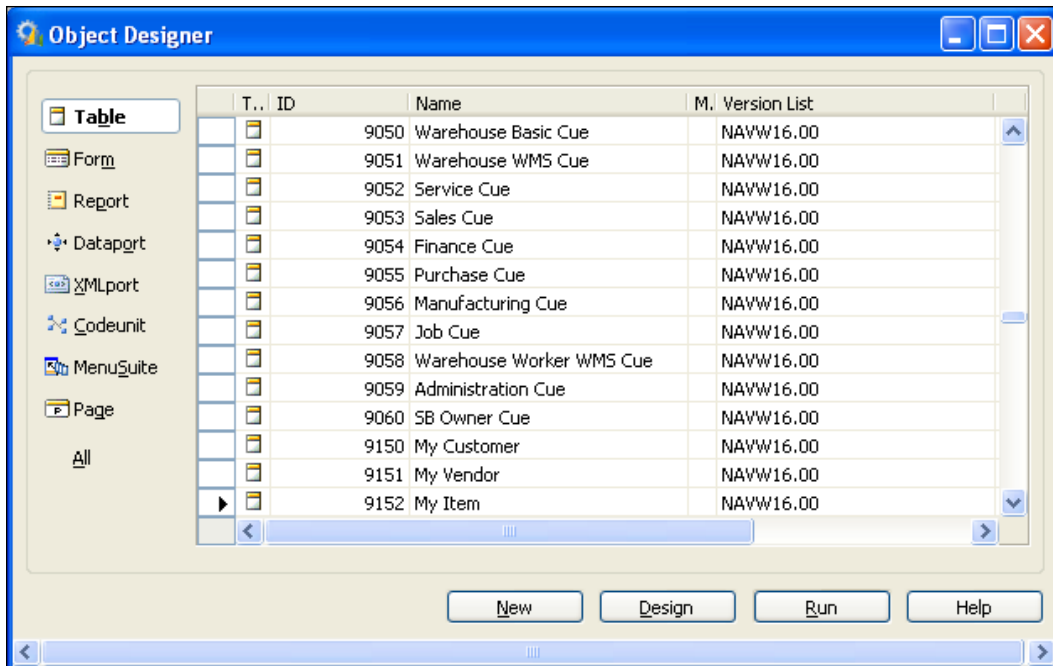
Primary Key	Released...	Open Sal...	Released...	Released...	Overdue ...	SOs Ship...	Custome...	Purchase...	Vendors ...
	0	20	21	14	40	1	0	27	0

There is very little C/AL code underlying all of these Role Center components. One exception is a small amount of code—the first time the Role Center is displayed, the supporting data record will be initialized. That simple code is shown in the following screenshot. There is also occasional filter setting code in some page parts, most of it associated with the WORKDATE or the USERID.



```
Documentation()
OnInit()
OnOpenPage()
RESET;
IF NOT GET THEN BEGIN
    INIT;
    INSERT;
END;
OnClosePage()
OnFindRecord(Which : Text[1024]) : Bo
```

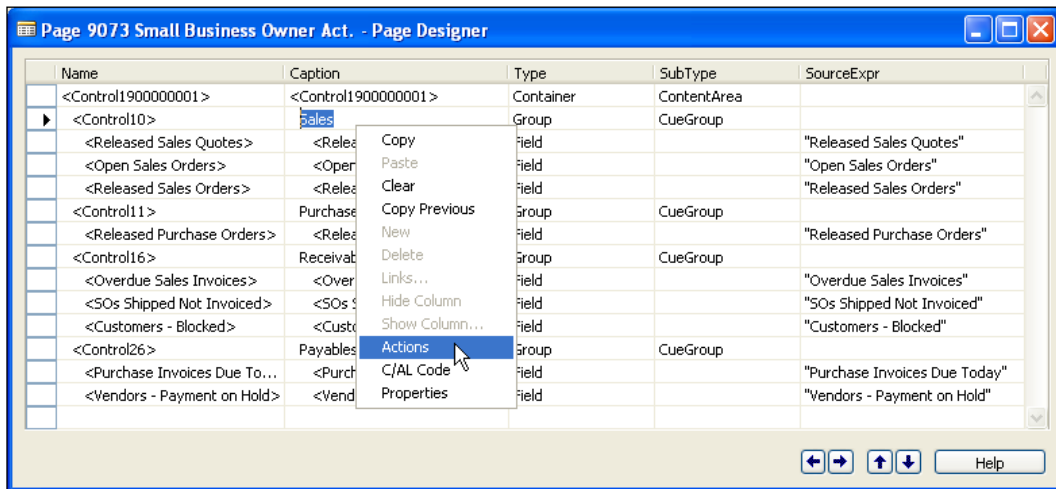
At this point, we step back to look at the bigger picture of Role Center support objects. The following screenshot shows the list of tables that serve the same purpose as Table 9060 - SB Owner Cue. Each of these Cue tables contains a series of FlowFields that support the Cues in the associated Role Center. As there are 21 Role Centers defined in the standard product distribution and fewer than 21 Cue tables, obviously, some of the Role Centers rely on the same Cue tables (in other words, some Cue sets appear on more than one Role Center page).



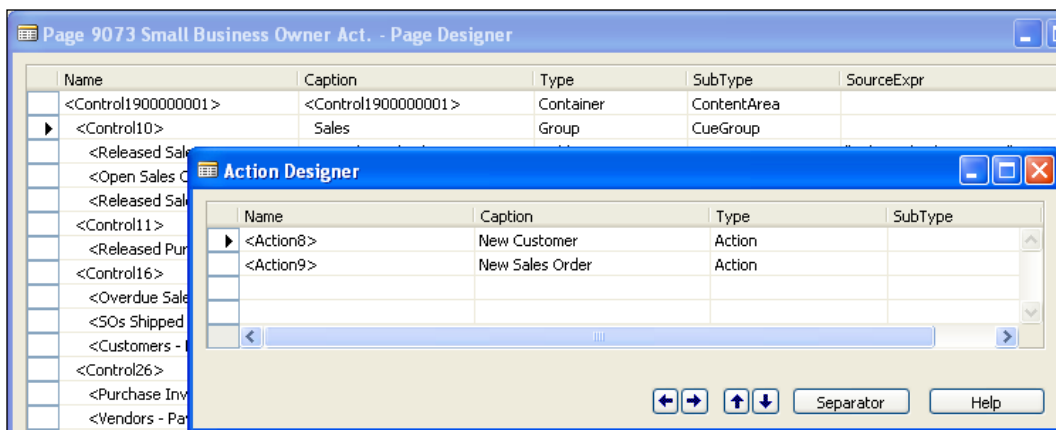
Cue Group Actions

Another set of Role Center page components that we need to analyze are the **Cue Group Actions**. While the Cues are the primary tasks that are presented to the user, the Cue Group Actions are a related secondary set of tasks. Cue Group Actions are defined in the Role Center in essentially the same way as Actions are defined in other page types.

As the name implies, Cue Group Actions are associated with a Control with the **SubType** CueGroup. If you right click on the CueGroup Control, one of the options available is **Actions** (as shown in the following screenshot):



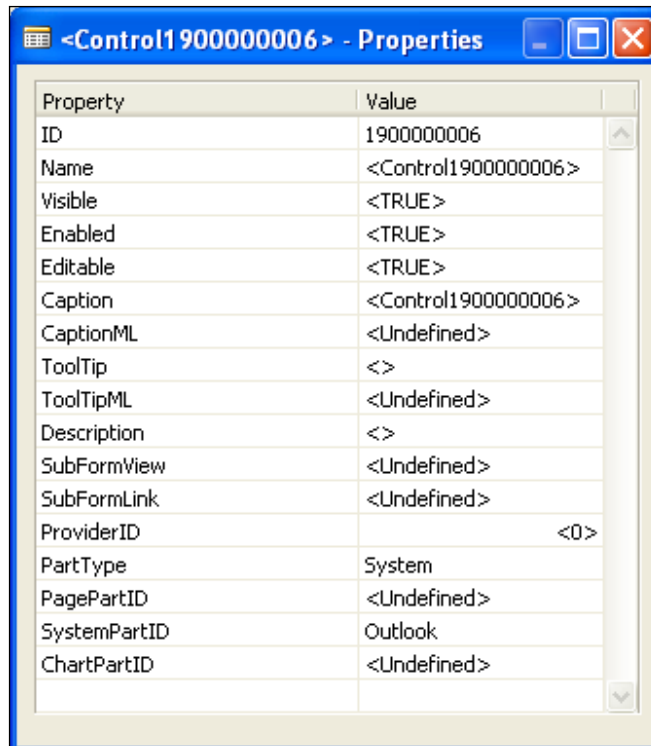
When you choose **Actions**, the **Action Designer** form is displayed. In this case, it shows the two CueGroup actions that are defined and which display for the Sales CueGroup in the Role Center page that we are analyzing. The Action Designer form is where we define the actions that we want available for a particular CueGroup in the Role Center.



System Part

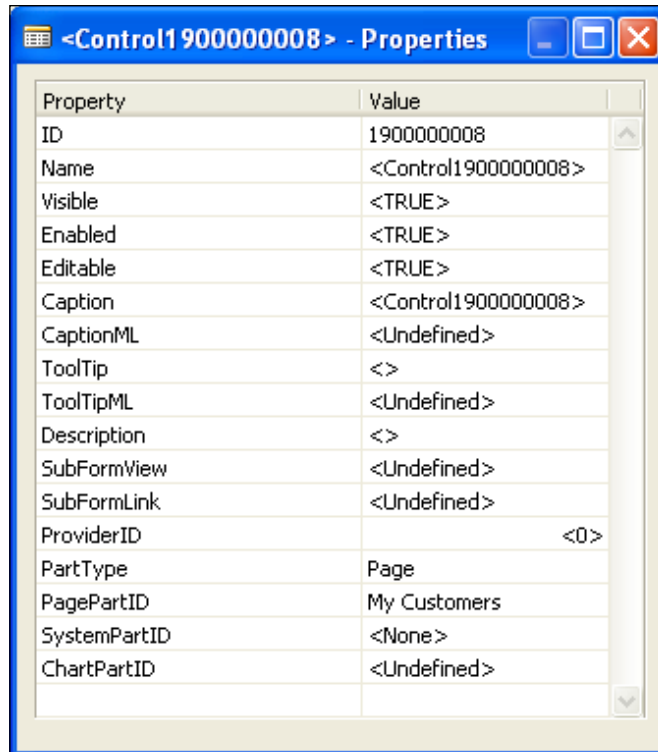
Now that we have thoroughly covered the components of the Role Activities portion of the Role Center page, let's take a look at the other components.

Returning to Page 9020 in the **Page Designer**, we examine the **Properties** of the next Part Control. Looking at this control's properties, we can see that this has a **PartType** of *System* and a **SystemPartID** of *Outlook*. That makes it easy to see that this Page Part is the one that incorporates a formatted display of the users' Outlook data into the Role Center.

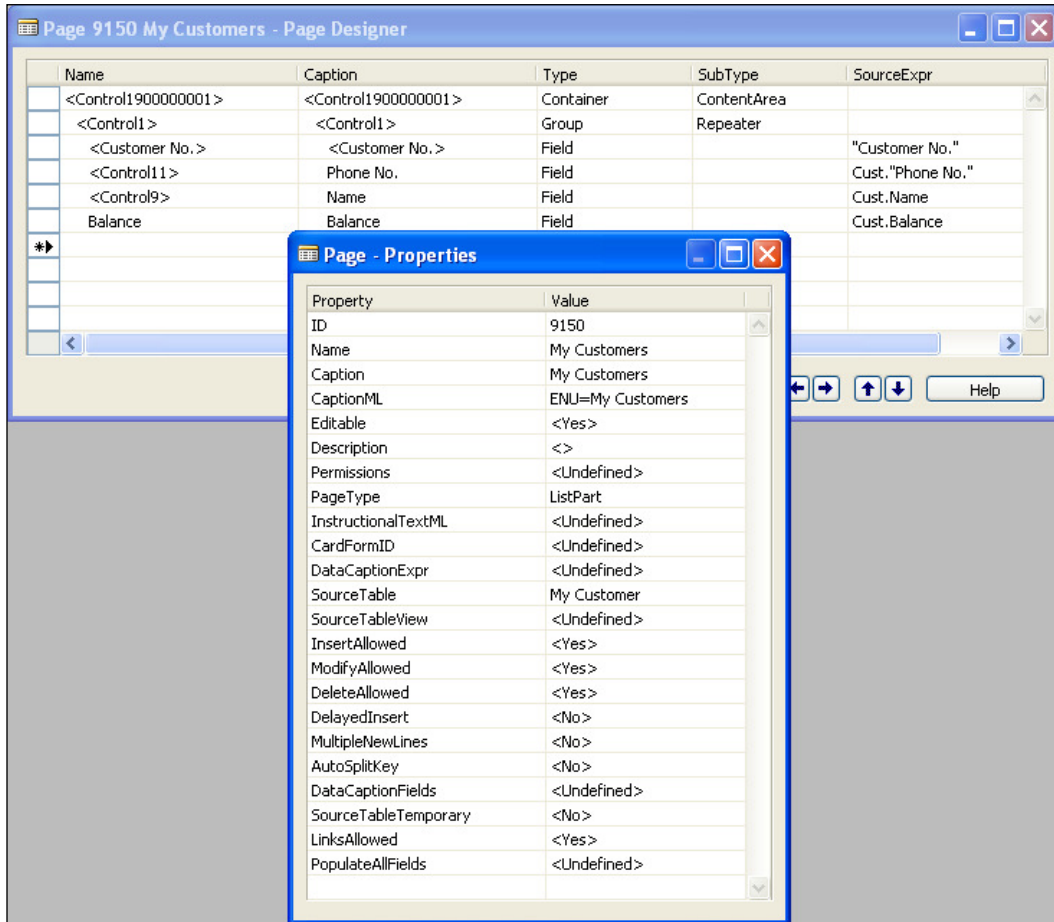


Page Part

Look at the Properties of the first Control in the second group, as shown in the following screenshot. In this case, the **PartType** is **Page** and the **PagePartID** is **MyCustomers**, which is Page 9150. This is obviously tied to the My Customers part of the Role Center page.

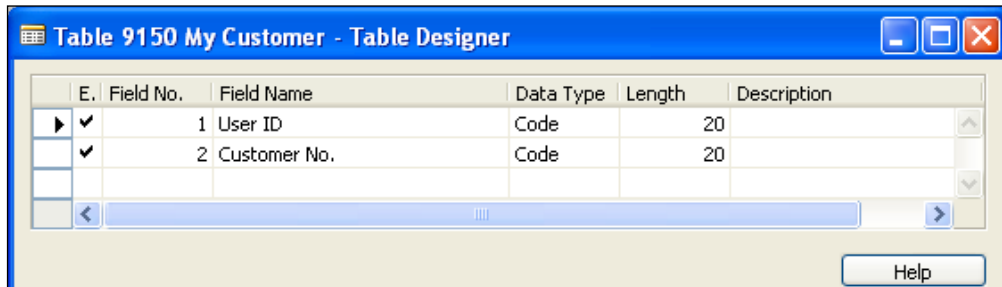


Looking at Page 9150 in the Page Designer, and specifically at the Page Properties of that page, we see what is shown in the following screenshot. The page has a **PageType** of **ListPart** and a **SourceTable** of **My Customer**.

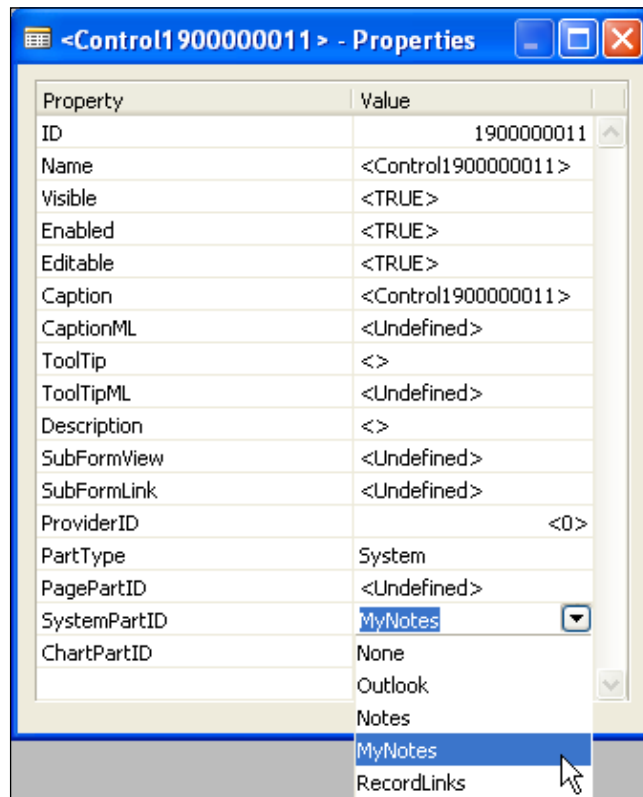


The My Customer source table is about as simple as a table can be (see the following Table Designer screenshot). It contains only the ID of the User who "owns" the customer along with the Customer No. As the key to this table is the combination of the two fields, any customer can be associated with any number of users, in other words several users could be watching over a particularly important customer.

There is a small amount of C/AL code in Page 9150 to support the filtering and reading of the data. In the OnOpenPage trigger, there is a SETRANGE function call to filter the customer records to be displayed. There is also a small amount of code to load data.



Inspecting the next two controls in Page 9020 (the Role Center page), we see two more very similar constructs – one for **My Vendors** and one for **My Items**. Continuing on to the last Control in Page 9020, we see another **PartType** of **System**, in this case one for the **SystemPartID** of **MyNotes**.

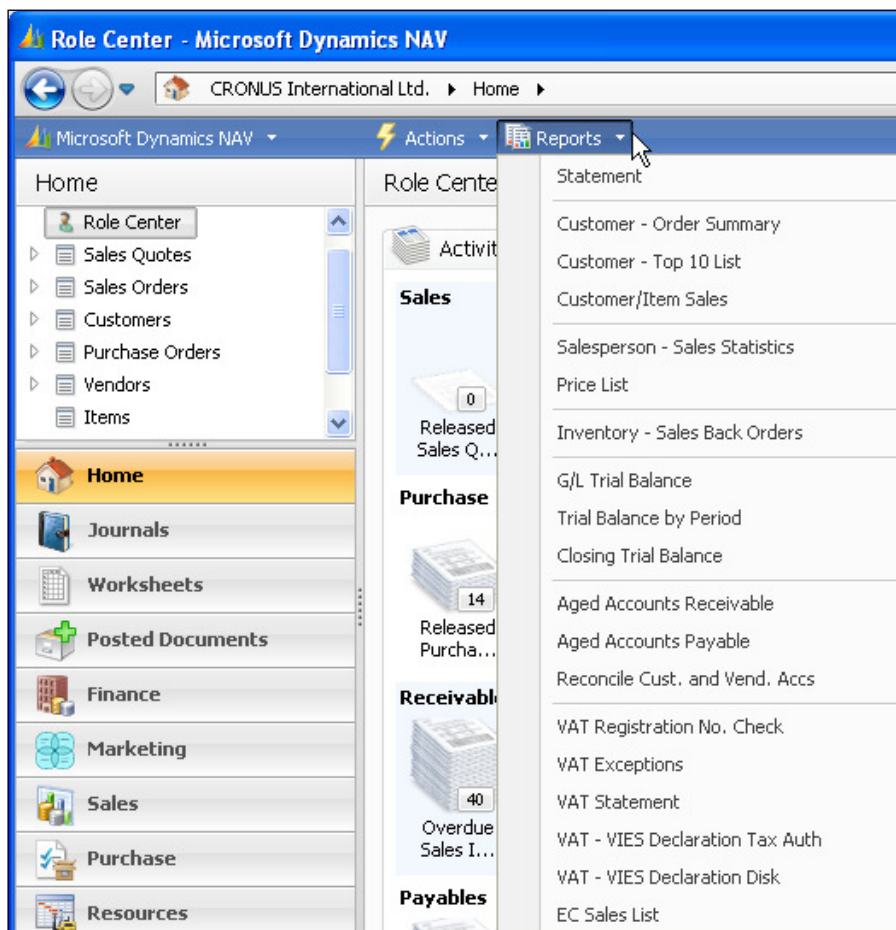


Navigation Pane and Action Menus

The last major component of the Role Center Page is the **Navigation Pane** plus the **Action Menus**. While on-screen there are four identifiable sections of the Role Center Page that provide lists of actions, that is menus, all the four are defined in the same area of the **Page Designer**—the **Action Designer**.

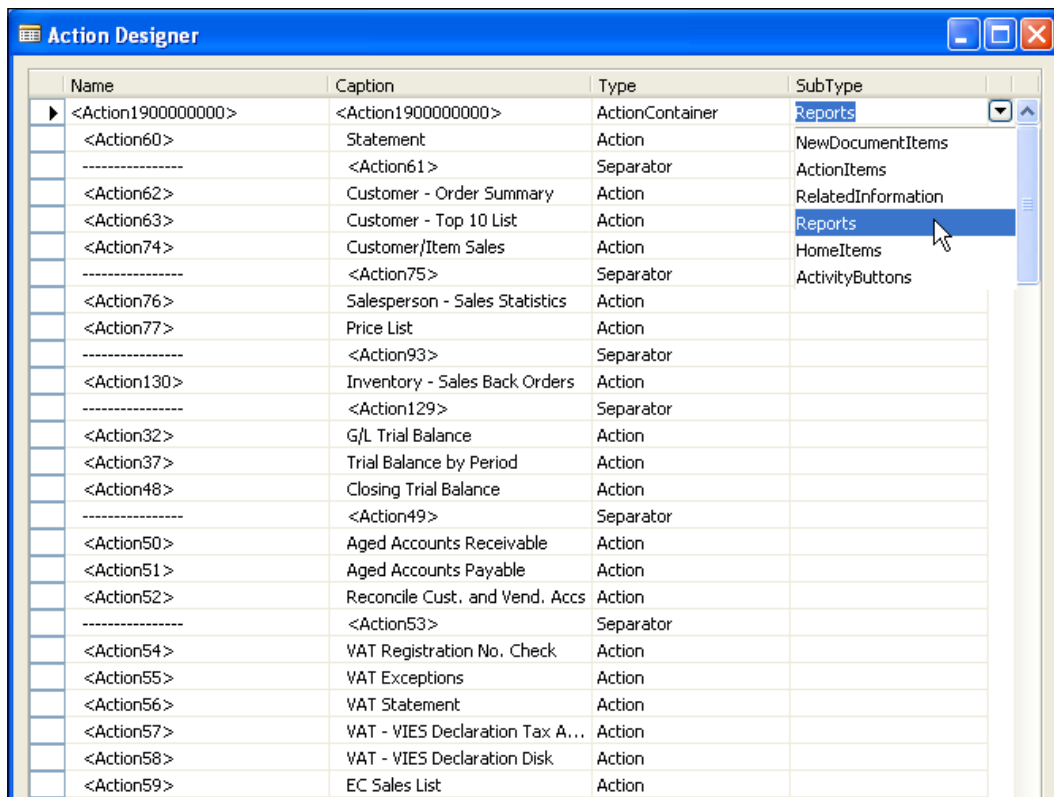
We will examine the four sections for Actions from right to left, top down, across the Role Center Page. The first two, **Reports** and **Actions**, are in the Command bar area.

The following two screenshots are the visible **Reports** action menu and the underlying controls for that menu, displayed in the **Action Designer**.



It is easy to compare the menu items in the page screenshot to the Captions in the Action Designer screenshot and see that these are different looks at the same material (outside look versus inside look). The Action Designer screenshot also shows the options that are available in the Action Designer lines. In this example, we're looking at instances of Reports, ActionItems, HomeItems, and ActivityButtons.

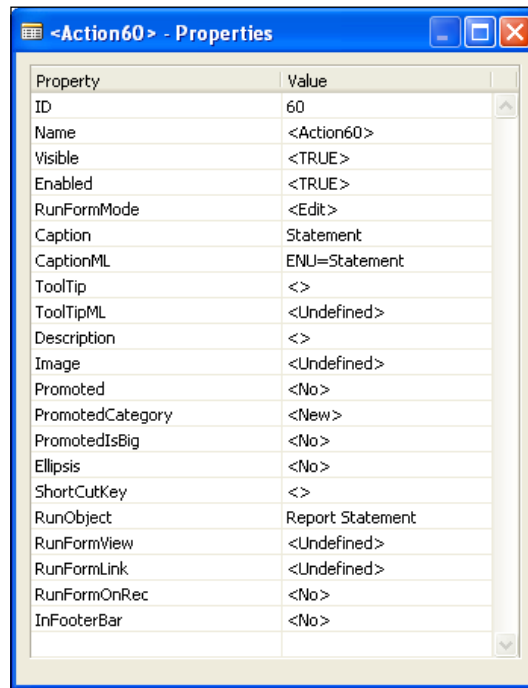
The **Action Designer** is accessed from the **Page Designer** form by focusing on the first blank line below the controls, then clicking on **View**, and selecting **Actions**.



The screenshot shows the 'Action Designer' window with a table containing the following data:

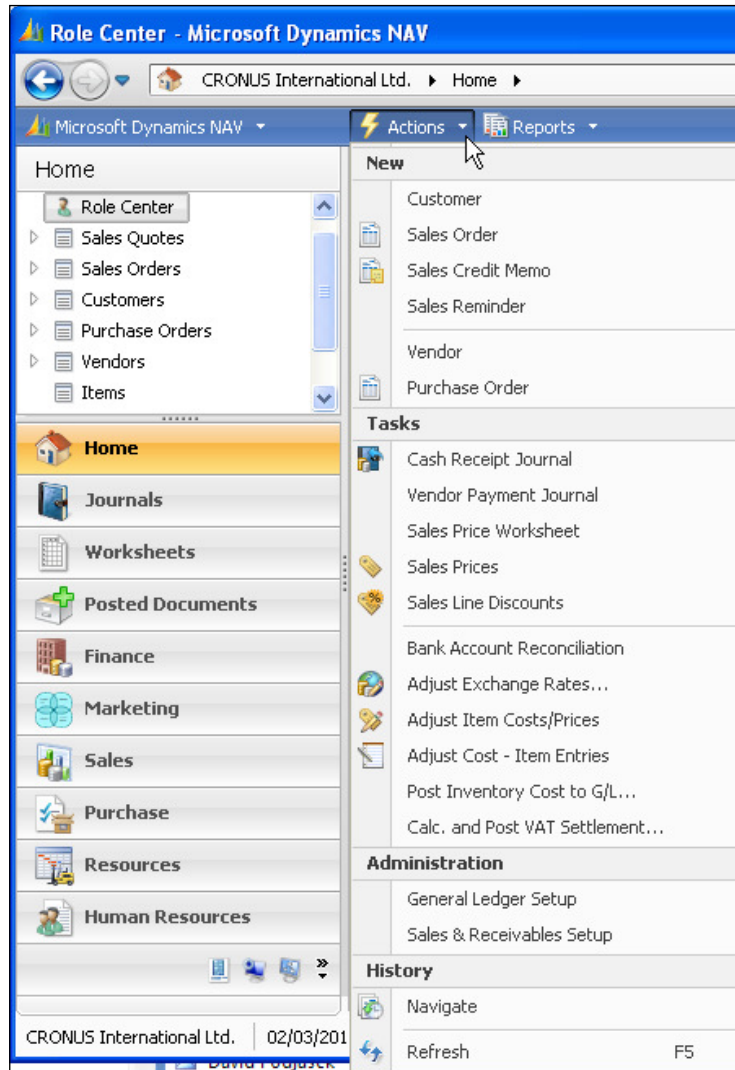
Name	Caption	Type	SubType
<Action1900000000>	<Action1900000000>	ActionContainer	Reports
<Action60>	Statement	Action	NewDocumentItems
<Action61>	<Action61>	Separator	ActionItems
<Action62>	Customer - Order Summary	Action	RelatedInformation
<Action63>	Customer - Top 10 List	Action	Reports
<Action74>	Customer/Item Sales	Action	HomeItems
<Action75>	<Action75>	Separator	ActivityButtons
<Action76>	Salesperson - Sales Statistics	Action	
<Action77>	Price List	Action	
<Action93>	<Action93>	Separator	
<Action130>	Inventory - Sales Back Orders	Action	
<Action129>	<Action129>	Separator	
<Action32>	G/L Trial Balance	Action	
<Action37>	Trial Balance by Period	Action	
<Action48>	Closing Trial Balance	Action	
<Action49>	<Action49>	Separator	
<Action50>	Aged Accounts Receivable	Action	
<Action51>	Aged Accounts Payable	Action	
<Action52>	Reconcile Cust. and Vend. Accs	Action	
<Action53>	<Action53>	Separator	
<Action54>	VAT Registration No. Check	Action	
<Action55>	VAT Exceptions	Action	
<Action56>	VAT Statement	Action	
<Action57>	VAT - VIES Declaration Tax A...	Action	
<Action58>	VAT - VIES Declaration Disk	Action	
<Action59>	EC Sales List	Action	

The specific action for each line is defined in the properties for that line (see the following screenshot for the properties of the first Action line in the Reports action menu):

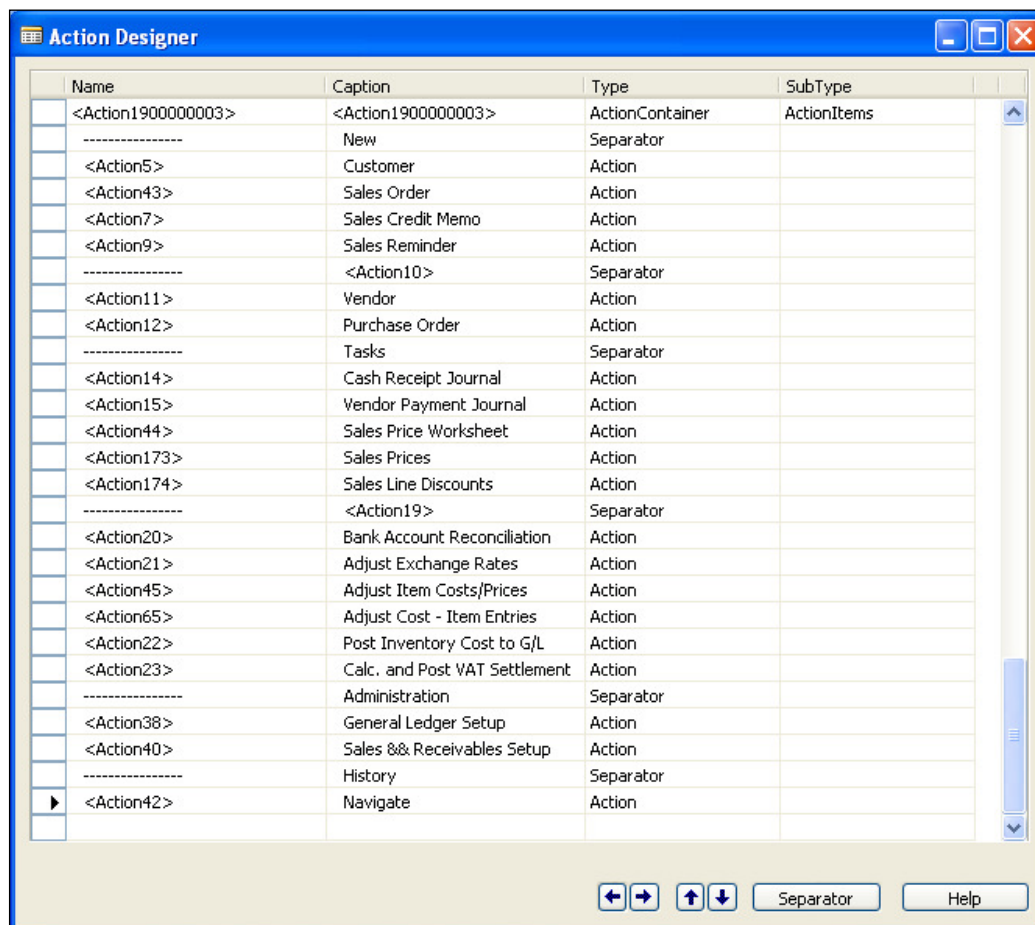


There is also a trigger for each action line, but this is misleading, because no code is allowed in action triggers in Role Center pages.

The following screenshot is of the external view of the active **Role Center Actions** menu:



The following screenshot shows the **Action Designer** contents for the same Role Center Actions menu. It is easy to compare the menu item descriptions in the page screenshot of the Actions menu to the Captions you see in the **Action Designer**.



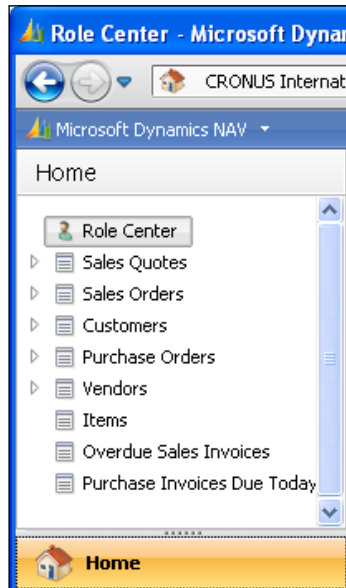
The screenshot shows the 'Action Designer' window with a table containing the following data:

Name	Caption	Type	SubType
<Action1900000003>	<Action1900000003>	ActionContainer	ActionItems
-----	New	Separator	
<Action5>	Customer	Action	
<Action43>	Sales Order	Action	
<Action7>	Sales Credit Memo	Action	
<Action9>	Sales Reminder	Action	
-----	<Action10>	Separator	
<Action11>	Vendor	Action	
<Action12>	Purchase Order	Action	
-----	Tasks	Separator	
<Action14>	Cash Receipt Journal	Action	
<Action15>	Vendor Payment Journal	Action	
<Action44>	Sales Price Worksheet	Action	
<Action173>	Sales Prices	Action	
<Action174>	Sales Line Discounts	Action	
-----	<Action19>	Separator	
<Action20>	Bank Account Reconciliation	Action	
<Action21>	Adjust Exchange Rates	Action	
<Action45>	Adjust Item Costs/Prices	Action	
<Action65>	Adjust Cost - Item Entries	Action	
<Action22>	Post Inventory Cost to G/L	Action	
<Action23>	Calc. and Post VAT Settlement	Action	
-----	Administration	Separator	
<Action38>	General Ledger Setup	Action	
<Action40>	Sales && Receivables Setup	Action	
-----	History	Separator	
▶ <Action42>	Navigate	Action	

At the bottom of the window, there are navigation buttons (left, right, up, down), a 'Separator' button, and a 'Help' button.

Turning our attention to the Navigation Pane, it has two main areas. The first area, at the top, is the Home button where the HomeItems are displayed. The comparative screenshots for the Home area action items follow.

The next screenshot is the user view:



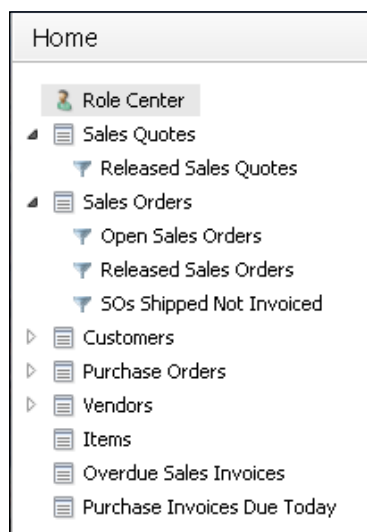
Following is the developer view:

A screenshot of the 'Action Designer' window, which displays a table of actions. The table has four columns: Name, Caption, Type, and SubType. The first row is an 'ActionContainer' named '<Action1900000001>' with the subtitle 'HomeItems'. The subsequent rows are 'Action' items with various captions like 'Sales Quotes', 'Sales Orders', 'Customers', 'Balance', 'Purchase Orders', 'Vendors', 'Balance', and 'Items'.

Name	Caption	Type	SubType
<Action1900000001>	<Action1900000001>	ActionContainer	HomeItems
<Action107>	Sales Quotes	Action	
<Action115>	Sales Orders	Action	
<Action112>	Customers	Action	
<Action113>	Balance	Action	
<Action105>	Purchase Orders	Action	
<Action102>	Vendors	Action	
<Action103>	Balance	Action	
<Action110>	Items	Action	

If you are doing development work on a Role Center, you can run the Role Center as a page from the C/SIDE Object Designer in the same way as other pages. However, the Role Center page will launch as a task page on top of whatever Role Center is configured for the active user. The Navigation Pane of the Role Center being modified will not be active and can't be tested with this approach. In order to test all of the aspects of the Role Center page, you must launch it as the assigned Role Center for the active user.

When you click on the small triangle to the left of the **Home** actions, the sub-home action items are displayed (see the following screenshot). These may come from items that were defined in the Action Designer form, or from an automatic merge of the items in the Cue Groups of the associated Role Center Page. In this screenshot, the child action items displayed are all automatically merged Cues, thus providing access to the associated lists either from the Cues or from the Home section of the Role Center Navigation Pane.



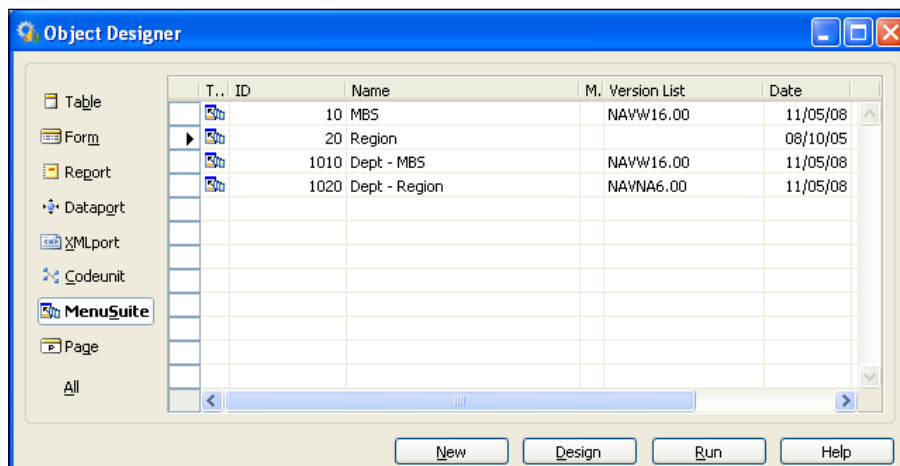
Departments

In the Navigation Pane, there are always at least two Activity buttons—Home and Departments. The links and pages available via the Departments Activity button are dependent on the MenuSuite. MenuSuites automatically adapt to show only the functions currently enabled by the active license and the user's permissions. Let's take a look at how the MenuSuite is constructed and how MenuSuite maintenance is accomplished.

MenuSuite levels


There are 15 levels of menu objects. They go from level 1 to 15, 1 being a "lower" level than level 2, and so on. The set of menus that is displayed is built up by first using the lowest level, then amending that by applying the next higher level, and so forth until all of the defined levels have been applied. Wherever a higher level redefines a lower level, the higher level definition is king.

The available menu levels are **MBS, Region, Country, Add-on 1** through **Add-on 10, Partner, and Company**. The lowest level that can be modified in Design mode without a special license is the Partner Level (you can open lower levels, but you cannot save changes). The lower levels are reserved to the NAV corporate developers and the ISVs who create add-ons. The following screenshot shows a menu suite with the original Microsoft master MenuSuite object (the MBS level), a regional localization object (Region), and then the entries created from those two entries for the Departments button by MenuSuite transformation:



MenuSuite structure

The menu that you see when you enter NAV is a roll-up of the contents of all of the menu objects, filtered based on your license and your assigned permissions. Each level has the ability to override the lower levels for any entry with the same **GUID** number.

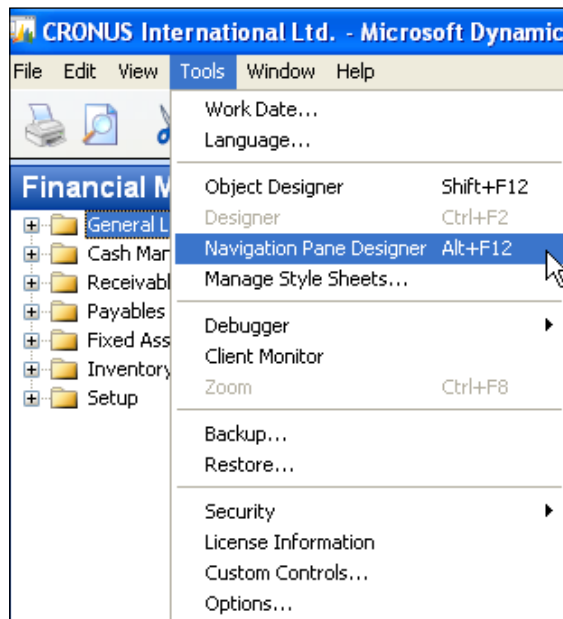
 **Globally Unique Identifier (GUID)** numbers are unique numbers which can be used for the identification of database objects, data records, and so on. The value of each GUID is generated by a Microsoft developed algorithm. The standard GUID representation is {12345678-1234-1234-1234-1234567890AB}.

A changed menu entry description is a second, higher level entry overriding the lower level entry. The lower level entry isn't really changed. A deleted lower level entry is not really deleted. Its display is blocked by the existence of a higher level entry indicating the effective deletion.

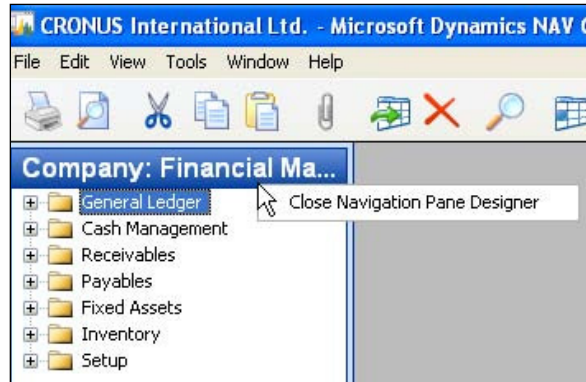
Changes that are made to one MenuSuite object level are stored as the differences between the effective result and the entries in the lower level objects. On the whole, this doesn't really make any difference in how you maintain entries, but if you export menu objects to text and study them, it may help explain some of what you see there and what happens when changes are made to MenuSuite entries.

MenuSuite development

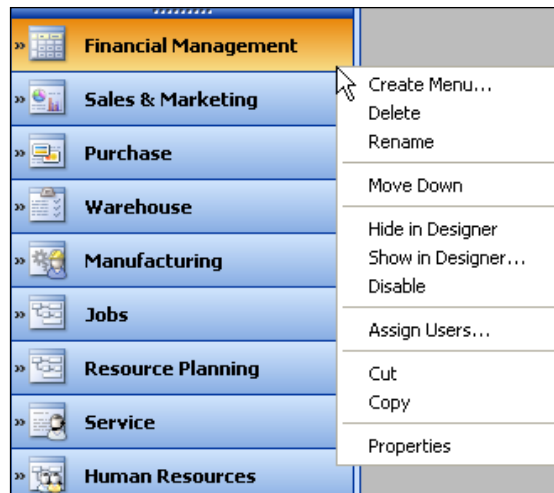
Developer access to the MenuSuite for modification is through **Tools | Object Designer | MenuSuite** in a fashion essentially similar to accessing other NAV objects for development purposes.



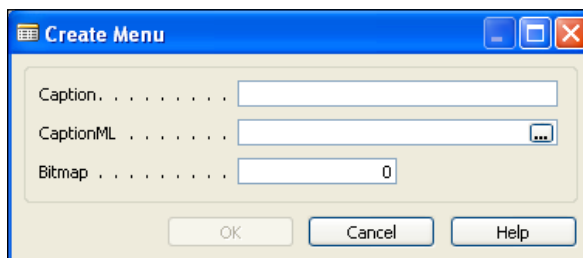
Exiting the MenuSuite Designer is done by executing a right-click on the open MenuSuite object heading. You will see the **Close Navigation Pane Designer** option as shown in the following screenshot (**Navigation Pane Designer** is an alternate name for the MenuSuite Designer). Click on that option to close the Designer. You will have the usual opportunity to respond to **Do you want to save the changes...**



Once you have opened the MenuSuite Designer at the desired level (typically Partner or Company), your next step is to create menu modifications. As mentioned earlier in this book, the development tools for MenuSuites are quite different from those of other NAV objects. In order to access the development options for a MenuSuite, highlight an item (for example, a menu or menu entry) while in the MenuSuite Designer, and right-click. If you highlight a menu, you will see the display shown in the following screenshot:

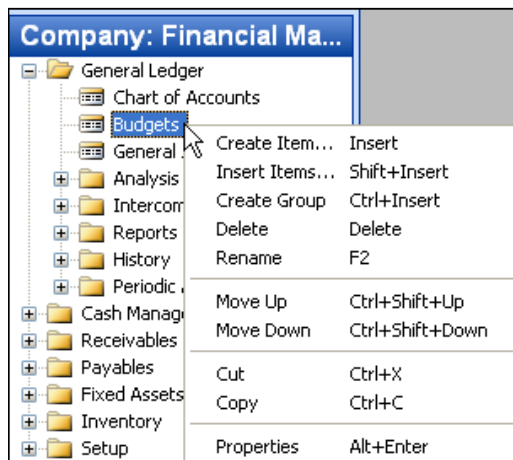


If you select the **Create Menu** option, you will see the form in the following screenshot. This allows you to create a new Menu (for example, a set of Actions). For example, you might want to create a new menu which allows access only to a limited set of inquiry pages:

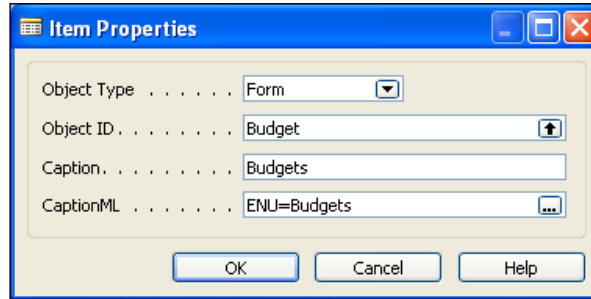


In this form, you can enter whatever you want the new menu's caption to be, as well as choose a bitmap to be displayed as the icon at the left of the caption string, when the MenuSuite is displayed in the Classic client.

If you highlight a menu entry and right-click, you will see the option list shown in the following screenshot:



The options shown on this menu are the total set of options available for a MenuSuite item. The first set of options allow you to create, delete, or rename items. The Move Up and Down are the principal positioning tools. If you click on Properties for a MenuSuite entry, you will see a display like that in the following screenshot:



The only object types allowed are Table, Form (which the RTC interprets as Page), Report, Dataport (which the RTC interprets as XMLport), and Codeunit. Once you have chosen an object type, you can specify the particular object of that type to be executed and what the captions are to be. And that's all that you can do in a MenuSuite as a developer.

If you are creating a new menu which will contain items that exist in other menus, you can populate it very quickly and easily. First, you create the new menu, and then just copy and paste desired menu items from other menus.

There is no allowance for any type of embedded C/AL code in a MenuSuite item to handle control, filtering, special validation, or other logic. You cannot invoke a specific function within an object. If you want to include any such customized capability, you must put that logic in an object such as a processing only report dedicated to handle the control task. Your MenuSuite item can then execute that control object which will apply your enhanced control logic, and will then invoke another object.

MenuSuite transformation

After you have made modifications to the MenuSuite, you must massage it through a process called **transformation** in order to convert the MenuSuite to Departments button Actions. The Transformation process is described in detail in the C/SIDE Reference Guide Help in "Transforming MenuSuite Objects" and "How to: Transform MenuSuite Objects". You may also want to refer to "How to: Create and Modify a MenuSuite Object".

It would be a good idea to experiment with adding new MenuSuite entries, changing or deleting existing entries, and reorganizing MenuSuite entries. View the results in the Classic Client, then transform the MenuSuite and view the results in the Role Tailored Client. Only through this process of hands-on experimentation will you gain a good understanding of how various changes will affect the Navigation Pane Departments Actions appearance.

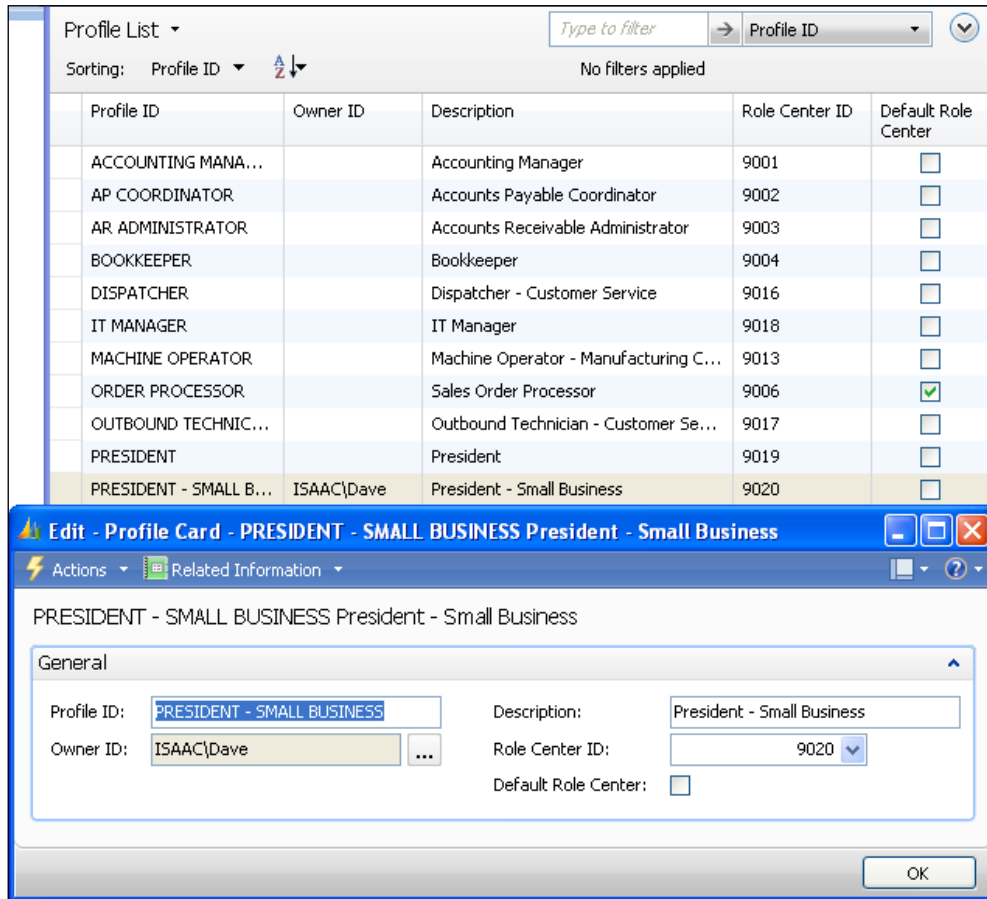
Configuration and personalization

At this point, we have covered most of the development based aspects of creating or maintaining a Role Center. In NAV 2009, once a Role Center has been defined, the implementer or a super user can configure the Role Center. Configuring the Role Center means defining which of the components that the Developer included will actually be presented and, in many cases, how they will be laid out on the screen. A Configured Role Center applies to all users assigned to that Role Center.

The configuration can only be changed by someone identified as the Owner of the Role Center and only when the Role Center is opened in Configuration mode. One Role Center is identified as the Default. The default Role Center will be used whenever a user logs into the system that has not been specifically assigned to another Role Center. The C/SIDE Help provides good information on configuring a Role Center. The Dynamics NAV Help accessible from the Role Tailored Client for Customizing Pages provides good information for a user to tailor their pages (terminology check: tailor=personalize=customize, the term may vary).

The Developer creates the overall boundaries of the Role Center toolset, but ultimately each user decides what parts of that toolset fits their needs best. Once the Developer has defined a full set of tools for a Role, the Implementer or System Administrator chooses a subset and arrangement for a site-specific Configuration. Then each user can further subset and arrange the Role Center page layout to suit their personal needs, re-choosing and rearranging at will.

The following screenshot of the Profile List (accessible via **Departments | Administration | Application Setup | Role Tailored Client | Lists**) shows a Role Center with the Owner identified and another Role Center set as the default Role Center:



Creating new C/AL routines

Now that you have a good overall picture of how you enable users to access the tools you create you are ready to start creating your own NAV C/AL routines. But first it is important that you learn your way around the NAV C/AL code in the standard product first. You may recall the advice in a previous chapter that the new code you create should be visually and logically compatible with what already exists. If you think of your new code as a guest being hosted by the original system, you will be doing what any thoughtful guest does – fitting smoothly into the host's environment.

An equally important aspect of becoming familiar with the existing code is to increase the likelihood that you can take advantage of the features and components of the standard product to address some of your application requirements. There are at least two groups of material that you can use.

One group is the Callable Functions that are used liberally throughout NAV. There is no documentation for most of those functions, so you must either learn about them here or by doing your homework (that is by studying NAV code). The second group includes many code snippets that you can copy when you face a problem similar to something the NAV developers have already addressed.

The code snippets differ from the callable functions in two ways. First, they are not structured as coherent and callable entities. Second, they are likely to only apply to your problem as a model, code that must be modified to fit the situation (for example changing variable names, adding or removing constraints, and so on).

In the following sections, we will look at some of those code structures. We will also discuss techniques for working with the code, for debugging, and as a developer, taking advantage of the strengths of the C/SIDE and NAV environment. Following this chapter, you should have enough tools in your NAV toolkit to begin working on basic development projects.

Callable functions

There are many callable functions in the standard NAV product. Most of these are designed to handle a very specific set of data or conditions and have no general-purpose use (for example the routines for updating Check Ledger entries during a posting process are likely to apply only to that specific function). If you are making modifications to a particular application area within NAV, you may find functions that you can utilize, either as they already exist or as models for new similar functions which you create.

There are functions within NAV that are relatively general purpose. They either act on data that is common in many different situations (for example dates) or they perform processing tasks that are common to many situations (for example provide access to an external file). We will review a number of such functions you may find usable at some point. If nothing else, they are useful as study guides for "here is how NAV does it". Example functions follow.

Codeunit 358 – Date Filter-Calc

This codeunit contains two functions you could use in your code to create filters based on the Accounting Period Calendar. The first is `CreateFiscalYearFilter` that has the following syntax:

```
CreateFiscalYearFilter.DateFilterCalc  
    (Filter, Name, BaseDate, NextStep)
```

The calling parameters are `Filter` (text, length 30), `Name` (text, length 30), `BaseDate` (date), and `NextStep` (integer).

The second such function is `CreateAccountingPeriodFilter` that has the following syntax:

```
CreateAccountingPeriodFilter.DateFilterCalc  
    (Filter, Name, BaseDate, NextStep)
```

The calling parameters are `Filter` (text, length 30), `Name` (text, length 30), `BaseDate` (date), and `NextStep` (integer).

In the following code screenshot from Page 151 – Customer Statistics, you can see how NAV calls these functions. Page 152 – Vendor Statistics, Page 223 – Resource Statistics, and a number of other Master table statistics forms also use this set of functions.

```
OnAfterGetRecord()  
  
IF CurrentDate <> WORKDATE THEN BEGIN  
    CurrentDate := WORKDATE;  
    DateFilterCalc.CreateAccountingPeriodFilter(CustDateFilter[1],CustDateName[1],CurrentDate,0);  
    DateFilterCalc.CreateFiscalYearFilter(CustDateFilter[2],CustDateName[2],CurrentDate,0);  
    DateFilterCalc.CreateFiscalYearFilter(CustDateFilter[3],CustDateName[3],CurrentDate,-1);  
END;
```

In the next code screenshot, NAV uses the filters stored in the `CustDateFilter` array to constrain the calculation of a series of `FlowFields` for the Customer Statistics page.

```
FOR i := 1 TO 4 DO BEGIN  
    SETFILTER("Date Filter",CustDateFilter[i]);  
    CALCULATE(  
        "Sales (LCY)","Profit (LCY)","Inv. Discounts (LCY)","Inv. Amounts (LCY)","Pmt. Discounts (LCY)",  
        "Pmt. Disc. Tolerance (LCY)","Pmt. Tolerance (LCY)",  
        "Fin. Charge Memo Amounts (LCY)","Cr. Memo Amounts (LCY)","Payments (LCY)",  
        "Reminder Amounts (LCY)","Refunds (LCY)","Other Amounts (LCY)");  
END;
```

When one of these functions is called, the `Filter` and `Name` fields are updated within the function, so you can use them effectively as return parameters, allowing the function to return a workable filter and a name for that filter. The filter is calculated from the `BaseDate` and `NextStep` you supply.

The returned filter is supplied back in the format of a range filter string, 'startdate..enddate' (for example 01/01/10..12/31/10). If you call `CreateFiscalYear`, the `Filter` will be for the range of a fiscal year, as defined by the system's Accounting Period table. If you call `CreateAccountingPeriodFilter`, the `Filter` will be for the range of a fiscal period, as defined by the same table.

The dates of the Period or Year filter returned are tied to the `BaseDate` parameter, which can be any legal date. The `NextStep` parameter says which period or year to use, depending on which function is called. A `NextStep = 0` says use the period or year containing the `BaseDate`, `NextStep = 1` says use the next period or year into the future, and `NextStep = -2` says use the period or year before last (that is, step back two periods or years).

The `Name` value returned is also derived from the Accounting Period table. If the call is to the `CreateAccountingPeriodFilter`, then `Name` will contain the appropriate Accounting Period Name. If the call is to the `CreateFiscalYearFilter`, then `Name` will contain 'Fiscal Year yyyy', where yyyy will be the four-digit numeric year.

Codeunit 359 – Period Form Management

This codeunit contains three functions that can be used for date handling. They are `FindDate`, `NextDate`, and `CreatePeriodFormat`:

- `FindDate` function
 - Calling Parameters (`SearchString` (text, length 3), `CalendarRecord` (Date table), `PeriodType` (Option, integer))
 - Returns `DateFound` Boolean
- ```
FindDate(SearchString, CalendarRec, PeriodType)
```

This function is often used in pages to assist with the date calculation. The purpose of this function is to find a date in the `CalendarRecord` table based on the parameters passed in. The search starts with an initial record in the `CalendarRecord` table. If you pass in a record that has already been initialized (that is you positioned the table to some date), then that will be the base date, otherwise the Work Date will be used.



The `PeriodType` is an Option field with the option value choices of 'day, week, month, quarter, year, accounting period'. For ease of coding, you could call the function with the integer equivalent (0, 1, 2, 3, 4, 5) or set up your own equivalent Option variable.

In general, it's a much better practice to set up an Option variable because the Option strings make the code self-documenting. Just using the Option integer values doesn't offer the next reader of your code to easily understand what's happening. In addition, if you only use the integer values and a change to the Option choices happens later, the integers might not represent the same Option choices (typically not a problem, but possible and not worth risking).

Finally, the `SearchString` allows you to pass in a logical control string containing `=`, `>`, `<`, `<=`, `>=`, and so on. `FindDate` will find the first date starting with the initialized `CalendarRecord` date that satisfies the `SearchString` logic instruction and fits the `PeriodType` defined. For example, if the `PeriodType` is day and the date 01/25/10 is used along with the `SearchString` of `>`, then the date 01/26/10 will be returned in the `CalendarRecord`.

- `NextDate` function
  - Calling Parameters (`NextStep` (integer), `CalendarRecord` (Date table), `PeriodType` (Option, integer))
  - Returns `NextStep` integer  
`NextDate (NextStep, CalendarRec, PeriodType)`

`NextDate` will find the next date record in the `CalendarRecord` table that satisfies the calling parameters. The `CalendarRecord` and `PeriodType` calling parameters for `FindDate` have the same definition as they do for the `FindDate` function. However, for this function to be really useful, the `CalendarRecord` must be initialized before calling `NextDate`—otherwise, the function will calculate the appropriate next date from day 0. The `NextStep` parameter allows you to define the number of periods of `PeriodType` to move, so as to obtain the desired next date. For example, if you start with a `CalendarRecord` table positioned on 01/25/10, a `PeriodType` of quarter (that is 3), and a `NextStep` of 2, the `NextDate` will move forward two quarters and return with `CalendarRecord` focused on Quarter, 7/1/10 to 9/30/10.

- `CreatePeriodFormat` function
  - Calling Parameters (`PeriodType` (Option, integer), `DateData` (date))
  - Returns `FormattedPeriod` (Text, length 10)  
`FormattedDate := CreatePeriodFormat (PeriodType, DateData)`

`CreatePeriodFormat` simply allows you to supply a date and specify which of its format options you want via the `PeriodType`. The function's return value is a ten-character formatted text value—for example `mm/dd/yy` or `ww/yyyy` or `mon/yyyy` or `qtr/yyyy` or `yyyy`.

## Codeunit 365 – Format Address

The functions in the `Format Address` codeunit, as the name suggests, serve the purpose of formatting addresses. The address data in any master record (Customer, Vendor, Sales Order Sell-to, Sales Order Ship-to, Employee, and so on.) may contain embedded blank lines, for example, the Address 2 line may be empty. When you print out the address information on a document or report, it will look better if there are no blank lines. These functions take care of that.

In addition, NAV provides setup options for multiple formats of City – Post Code – County – Country combinations. The `Format Address` functions also take care of formatting your addresses according to what was chosen in the setup or has been defined in the Countries/Regions form for different Postal areas.

There are over 50 data-specific functions in the `Format Address` codeunit. These data-specific functions allow you to pass a record parameter for the record containing the raw address data (such as a Customer record, a Vendor Record, a Sales Order, and so on.). These function calls also require a parameter of a one-dimensional Text array with 8 elements of length 90. Each function extracts the address data from its specific master record and stores it in the array. The function passes that data to a general-purpose function, which does the actual work of re-sequencing and compressing the data according to the various setup rules.

The following is an example of function call format for these functions for `Company` and the `Sales Ship-to` addresses. In each case `AddressArray` is Text, Length 90, and one-dimensional with 8 elements.

```
"Format Address".Company(AddressArray, CompanyRec) ;
"Format Address".SalesHeaderShipTo(AddressArray, SalesHeaderRec) ;
```

The result of the function's processing is returned in the `AddressArray` parameter.

In addition to the data-specific functions in the `Format Address` codeunit, you can also directly utilize the more general-purpose functions contained therein and called by the data-specific functions. If you added a new address structure as part of an enhancement, you probably want to create your own data-specific address formatting function in your own codeunit. But you might as well design your function to call the general purpose functions that already exist (and are debugged).

The primary general-purpose address formatting function (and the one you are most likely to call directly) is `FormatAddr`. This is the function that does most of the work in this codeunit. The syntax for the `FormatAddr` function is as follows:

```
FormatAddr (AddressArray, Name, Name2, ContactName, Address1, Address2,
 City, PostCode, County, CountyCode)
```

The calling parameters of `AddressArray`, `Name`, `Name2` and `ContactName` are all text, length 90. `Address1`, `Address2`, `City`, and `County` are all text, length 50. `PostCode` and `CountryCode` are code, length 20 and length 10, respectively.

Your data is passed into the function in the individual `Address` fields. The results are passed back in the `AddressArray` parameter for you to use.

There are two other functions in the `Format Address` codeunit that can be called directly. They are `FormatPostCodeCity` and `GeneratePostCodeCity`. The `FormatPostCodeCity` function serves the purpose of finding the applicable setup rule for `PostCode` + `City` + `County` + `Country` formatting. It then calls the `GeneratePostCodeCity` function, which does the actual formatting.

If you are going to use functions from Codeunit 365, take care to truly understand how they operate. In this case, as well as all others, study a function and test with it before assuming you understand how it works. There is no documentation for these functions, so their proper use is totally up to you.

## Codeunit 396 – NoSeriesManagement

Throughout NAV, master records (for example `Customer`, `Vendor`, `Item`, and so on.) and activity documents (`Sales Order`, `Purchase Order`, `Warehouse Transfer Orders`, and so on) are controlled by the unique identifying number assigned to each one. This unique identifying number is assigned through a call to a function within the `NoSeriesManagement` codeunit. That function is `InitSeries`. The calling format for `InitSeries` is as follows:

```
NoSeriesManagement.InitSeries (WhichNumberSeriesToUse,
 LastDataRecNumberSeriesCode, SeriesDateToApply, NumberToUse,
 NumberSeriesUsed)
```

The parameter `WhichNumberSeriesToUse` is generally defined on a **Numbers** Tab in the Setup record for the applicable application area. The `LastDataRecNumberSeriesCode` tells the function what `Number Series` was used for the previous record in this table. The `SeriesDateToApply` parameter allows the function to assign ID numbers in a date-dependent fashion. The `NumberToUse` and the `NumberSeriesUsed` are return parameters.

The following screenshots show examples for first, Table 18 - Customer and second Table 36 - Sales Header.

```
OnInsert()
IF "No." = '' THEN BEGIN
 SalesSetup.GET;
 SalesSetup.TESTFIELD("Customer Nos.");
 NoSeriesMgt.InitSeries(SalesSetup."Customer Nos.",xRec."No. Series",00,"No.", "No. Series");
END;
```

```
OnInsert()
SalesSetup.GET;

IF "No." = '' THEN BEGIN
 TestNoSeries;
 NoSeriesMgt.InitSeries(GetNoSeriesCode,xRec."No. Series","Posting Date","No.", "No. Series");
END;
```

With the exception of `GetNextNo` (used in assigning unique identifying numbers to each of a series of transactions), you are not likely to use other functions in the `NoSeriesManagement` codeunit. The other functions are principally used either by the `InitSeries` function or other NAV routines whose job it is to maintain Number Series control information and data.

## Codeunit 397 – Mail

This codeunit contains a series of functions for interfacing with Microsoft Outlook as an Automation Controller. As the complexity of both Automation Controllers and the APIs for various products, including Outlook, are beyond the scope of this book, we will not cover the functions of this codeunit in any detail. Suffice it to say that if you are going to create code that deals with Outlook in any way, you should start with this codeunit either for functions you can use directly or as a model for what you need to do.

If you need an SMTP interface for an Email interface, you should use codeunit 400 – SMTP Mail.

## Codeunit 408 – Dimension Management

The Dimension Management codeunit is of general interest because dimensions are so widely used (and useful) throughout NAV. Dimensions are a user-definable categorization of the data. There are two **Global Dimensions**, which are carried as values in the primary data records. Any dimensioned data can also have up to six additional categorizations (**Shortcut Dimensions**), which are stored in subordinate tables. Each dimension can have any number of possible values. More detailed information about Dimensions in NAV is available in the **Help**, and in the training documentation about the functional application. A good place to start is the Dimension Table **Help**.

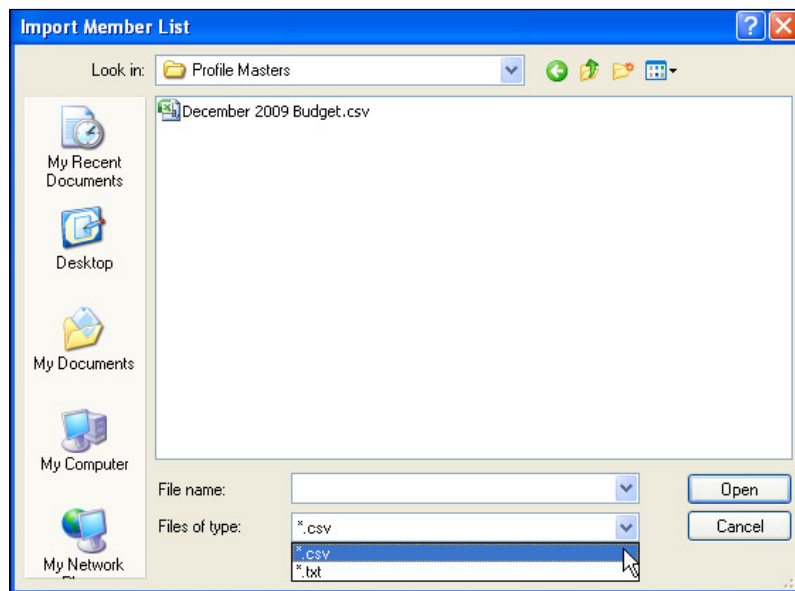
When you move, process, post, delete, or otherwise manipulate many different types of NAV data, you must also deal with the associated Dimensions data. The functions in Codeunit 408 support that activity. You would be wise not to be creative in your use of the Dimension Management functions, but simply find and study existing code for a situation similar to the one on which you are working.

If you are manipulating standard system records (records that are part of the system as delivered from Microsoft), then you will call functions in Codeunit 408 directly, perhaps even cloning existing code from another routine for your calling routines. If you are creating new tables that contain Dimensions, you may need to create your own Dimensions handling functions in your codeunit. In this case, you should model your new functions on Codeunit 408 code. Take note that, in most of NAV, posting of Dimensions initially uses a temporary table, in order to avoid degrading performance by locking critical **Dimensions** tables. You should follow this approach with your design.

## Codeunit 412 – Common Dialog Management

The Common Dialog Management codeunit contains just one function, the `OpenFile` function. This function provides the system's access to the OCX module for the Microsoft Common Dialog Control, which is a standard part of Windows. The Common Dialog Control provides a user interface function for accessing external files.

The following screenshot shows the type of screen that appears when the Common Dialog Control is executed:



The code that invoked this dialog is as follows:

```
CDM.OpenFile('Import Member List','',4,'*.csv|*.csv|*.txt|*.txt',0);
```

The syntax for the `OpenFile` function is as follows:

```
OpenFile (ScreenTitle,DefaultFileName,FileTypeOption,
FileTypeFilterString,Action)
```

The calling parameters are `ScreenTitle` (text, length 50), `DefaultFileName` (text, length 250), `FileTypeOption` (option choices of ' ', Excel, Word, Custom), `FileTypeFilterString` (text, length 250), and `Action` (option choices of Integer, Open, Save).

In this instance, the `ScreenTitle` is defined, the `DefaultFileName` is omitted, there is a `FileTypeOption` of Custom, which allows the `FileTypeFilterString` to be specified for `*.csv` and `*.txt` (see **Files of type** in preceding screenshot), and an `Action` of 0 (zero), which defines what action button (**Open** or **Save**) will be displayed (**Open** is chosen here).

The syntax rule for the `FilterTypeFilterString` is a string sequence consisting of the Filter Type description, followed by a pipe symbol, which in turn is followed by the Filter Mask. Each subsequent filter option description+mask sequence is separated from the preceding one by another pipe symbol.


The default filter options in **Codeunit 412** are defined as **Text** strings, as shown in the following screenshot:

| Name    | ConstValue                                                |
|---------|-----------------------------------------------------------|
| Text003 | Text Files (*.txt) *.txt All Files (*.*) *.*              |
| Text004 | Microsoft Excel Files (*.xlsx) *.xlsx All Files (*.*) *.* |
| Text005 | Word Documents (*.doc) *.doc All Files (*.*) *.*          |

## Sampling of function models to review

It is very helpful when you're creating new code to have a model that works, which you can study (or clone). This is especially true in NAV where there is little or no "how to" documentation available for many different functions. One of the more challenging aspects of learning to develop in the NAV environment is learning how to address the wide variety of common issues in the "NAV way". The "NAV way" is most beneficial, because there is no better place to learn the strengths and the subtle features of the product than to study the code written by the developers who are part of the inner circle.

A selection of objects containing functions you may find useful as models follows. You certainly will find these useful for study. Here is how "it's" done in NAV ("it" obviously varies depending on the function's purpose). When you build your function modeled on NAV functions, the new code should reside in a customer licensed codeunit.

 It is not good practice to add custom functions to the standard NAV Codeunits. Keeping customizations well segregated in clearly identified custom objects makes both maintenance and upgrades easier.

## Codeunit 228 – Test Report-Print

This codeunit contains a series of functions to invoke the printing of various Test Reports. These functions are called from various data entry forms, typically Journal forms. You can use these functions as models for any situation where you want to allow the user to print a test report from a form/page menu or command button.

Although all of the functions in this codeunit are used to print Test Reports, there isn't any reason you couldn't use the same logic structure for any type of report. The basic logic structure is to create an instance of the table of interest, apply any of the desired filters to that table instance, execute other appropriate setups for the particular report, and then call the report with a code line similar to the following:

```
REPORT.RUN (ReportID, TRUE, FALSE, DataRecName)
```

The first Boolean option will enable the Report Request form/page if `TRUE`. The second Boolean function chooses to print the report on the system printer (`TRUE`) or the printer defined through the Printer Selection table (`FALSE`).

Use of the `RUN` function will invoke a new instance of the report object. In cases where a new instance is not desired, `RUNMODAL` is used.

## Codeunit 229 – print documents

This codeunit is very similar to Codeunit 228 in its internal logic structure. It contains a series of functions for invoking the printing of document formatted reports. A document report in NAV is formatted such that a page is the basic unit (for example invoices, orders, statements, checks). For those documents printed from the Sales Header or Purchase Header, a basic "calculate discounts" function is called from Codeunit 229 for the header record prior to calling the report object which will print the chosen document.

In some cases, there are several reports (in this case, documents) all generated from the same table, for example the Sales Quote, Sales Order, Sales Invoice and Sales Credit Memo all of which are from Tables 36 – Sales Header and 37 – Sales Line. For such situations, the function has a common set of pre-processing logic followed by a CASE statement to choose the proper report object call. In the case where there is only one report-table combination (for example, Bank Checks), the function is simpler, but still basically, has the same structure (just without the CASE statement).

## Other objects to review

Other Codeunits, that you should review for an insight into how NAV manages certain activities and interfaces, are:

- Codeunit 1 – Application Management: A library of utility functions widely used in the system
- Codeunits 11, 12, 13, 21, 22, 23, 80, 81, 82, 90, 91, 92 - the Posting sequences for Sales, Purchases, General Ledger, Item Ledger; these control the posting of journal data into the various ledgers
- Codeunit 419 – 3-tier Management: Functions specific to three-tier operation
- Codeunits 800, 801, 802 – Online Map interfacing
- Codeunit 5054 – Word Management: Interfaces to Microsoft Word
- Codeunit 5063 – Archive Management: Storing copies of processed documents
- Codeunits 5300 thru 5314 – Outlook interfacing
- Codeunits 5813 thru 5819 – Undo functions
- Table 330 – Currency Exchange Rate: Contains some of the key currency conversion functions
- Table 370 – Excel Buffer: Excel interfacing
- Page 344 – Navigate: Home of the unique and powerful Navigate feature

## Management codeunits

There are approximately 100 codeunits with the word "Management" as part of their description name. Each of these codeunits contains functions in which the purpose is the management of some specific aspect of NAV data. Many are very specific to a narrow range of data. Some are more general, because they contain functions you can reuse in another application area (for example, Codeunit 396 – NoSeriesManagement).



When you are working on an enhancement in a particular functional area, it is extremely important to check the Management codeunits utilized in that area.

You may be able to use some functions directly. This will have the benefit of reducing the code you have to create and debug. Of course, when a new version is released, you will have to check to see if the functions on which you relied have changed in a way that affects your code.

If you can't use the existing material as is, you may find functions you can use as models for tasks in the area of your enhancement. And, even if that is not true, by researching and studying the existing code, you will learn more about how data is structured and processes flow in the standard system.

## Documenting modifications

We have discussed many of the good documentation practices that you should follow, when modifying an NAV system. We will briefly review those here.

Identify and document your modifications. Assign a unique project ID and use it for version tags and all internal documentation tags. Assign a specific number range for any new objects.

Wrap your customizations in code with tagged comments. Place an identifying "Modification starts here" comment before any modification and a "Modification ends here" comment at the end. Retain any replaced code inside comments. Depending on the amount of replaced or deleted code, it should be commented out with either slashes or braces (// or { }).



No matter how much or what type of standard NAV C/AL code is affected, the original code should remain intact as comments.

Always include explanatory documentation in the Documentation Trigger of modified objects. In the case of changes that can't be documented in-line such as changes to properties, the Documentation Trigger may be the only place you can easily create a documentation trail describing the changes.

If your modification requires a significant number of lines of new code or code that can be called from more than one place, you should strongly consider putting the body of the modification into a separate new codeunit (or codeunits) or at least in a new separate function (or functions). This approach allows you to minimize the footprint of your modifications within the standard code. That will make support, maintenance, and upgrading easier.

Where feasible to do so, create new versions of standard objects and modify those rather than modifying the original object. This works well for many reports and some pages, but often doesn't work well for tables, codeunits, and many pages.

Maintain an external document that describes the purpose of the modification and a list of what objects were affected. Ideally this documentation should begin with a Change Request or Modification Specification and then be expanded as the work proceeds. The description should be relatively detailed and written, so that knowledgeable users (as well as trained C/AL developers) can understand what has been done and why. When you have to work hard to understand the reason for a code structure in the standard product, it should make you appreciate why you should document well to make your work easier for others to understand.

## Multi-language system

The NAV system is designed as a multi-language system, meaning it can interface with users in more languages than just English. The base product is distributed with American English as the primary language, but each local version comes with one or more other languages ready for use. As the system can be set up to operate from a single database displaying user interfaces in several different languages, NAV is particularly suitable for firms operating from a central system serving users in multiple countries. NAV is used by businesses all over the world, operating in dozens of different languages. It is important to note that when the application language is changed, that has no affect on the data in the database and, in fact, does not multi-language enable the data.

The basic elements that support the multi-language feature include:

- Multi-Language Captioning properties (for example, **CaptionML**) supporting definition of alternative language captions for all fields, button labels, titles, and so on.
- Application Management codeunit logic that allows language choice on login.
- `fin.stx` files supplied by NAV, which are language specific and contain texts used by C/SIDE for various menus such as File, Edit, View, Tools, and so on. (`fin.stx` cannot be modified except by the Microsoft NAV Development Team).
- The Text Constants property **ConstantValueML** supporting definition of alternative language messages.

Before embarking on creating modifications that need to be multi-language enabled, be sure to review all the available documentation on the topic. It would also be wise to do some small scale testing to ensure you understand what is required, and that your approach will work (of course, this is always a good idea for any potentially significant compatibility issue).

## Multi-currency system

NAV was one of the first ERP systems to fully implement a multi-currency system. Transactions can start in one currency and finish in another. For example, you can create the order in US dollars and accept payment for the invoice in Euros. For this reason, where there are money values, they are generally stored in the local currency (for example **LCY**) as defined in setup. However, there is a set of currency conversion tools built into the applications and there are standard (by practice) code structures to support and utilize those tools. Two examples of code segments from the *Sales Line* table illustrating handling of money fields follow:

```
GetSalesHeader;
IF SalesHeader."Currency Code" <> '' THEN BEGIN
 Currency.TESTFIELD("Unit-Amount Rounding Precision");
 "Unit Cost" :=
 ROUND(
 CurrExchRate.ExchangeAmtLCYToFCY(
 GetDate,SalesHeader."Currency Code",
 "Unit Cost {LCY}",SalesHeader."Currency Factor",
 Currency."Unit-Amount Rounding Precision")
)
END ELSE
 "Unit Cost" := "Unit Cost {LCY}";
```

In both cases, there's a function call to **ROUND** and use of the currency specific **Currency. "Amount Rounding Precision"** control value.

```
"Line Discount Amount" :=
 ROUND(
 ROUND(Quantity * "Unit Price",Currency."Amount Rounding Precision") *
 "Line Discount %" / 100,Currency."Amount Rounding Precision");
```

As you can see, before creating any modification that has money fields, you must familiarize yourself with the NAV currency conversion feature and the code that supports it. A good place to start is the *C/AL* code within Table 37 - Sales Line, Table 39 - Purchase Line, and Table 330 - Currency Exchange Rate.

## Code analysis and debugging tools

The NAV tools and techniques that you use to determine what code to modify and to help you debug modifications are essentially the same. The goal in the first case is to focus on your modifications, so that you have the minimum effect on the standard code. This results in multiple benefits. Smaller pieces of well focused code are easier to debug, easier to document, easier to maintain, and easier to upgrade.

As of NAV's relatively tight structure and unique combination of features, it is not unusual to spend significantly more time in determining the right way to make a modification than it actually takes to code the modification. Obviously this depends on the type of modification being made. Unfortunately, the lack of documentation regarding the internals of NAV also contributes to an extended analysis time required to design modifications. The following sections review some of the tools and techniques you can use to analyze and test.

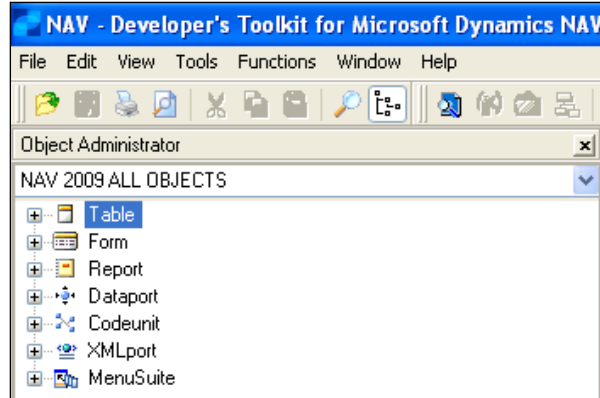
## Developer's Toolkit

To paraphrase the introduction in the NAV Developer's Toolkit documentation, the Toolkit is designed to help you analyze the source code. This makes it easier to design and develop application customizations and to perform updates. The Developer's Toolkit is not part of the standard product distribution, but is available to all Microsoft Partners for NAV for download from the Partner website. While it takes a few minutes to set up the Developer's Toolkit for the database on which you will be working, the investment is worthwhile. Follow the instructions in the Developer's Toolkit manual for creating and loading your Toolkit database. The Help files in the Developer's Toolkit are also useful. As of late 2009, the current NAV Developer's Toolkit is V3.00. V3.00 does not deal with the new features of NAV 2009 associated with the Role Tailored Client or three tier functionality.

The NAV Developer's Toolkit has two major categories of tools – the Compare and Merge Tools, and the Source Analyzer:

- The Compare and Merge Tools are useful anytime you want to compare a production database's objects to an unmodified set of objects to identify what has been changed. This might be in the process of upgrading the database to a new version or simply to better understand the contents of a database when you are about to embark on a new modification adventure.

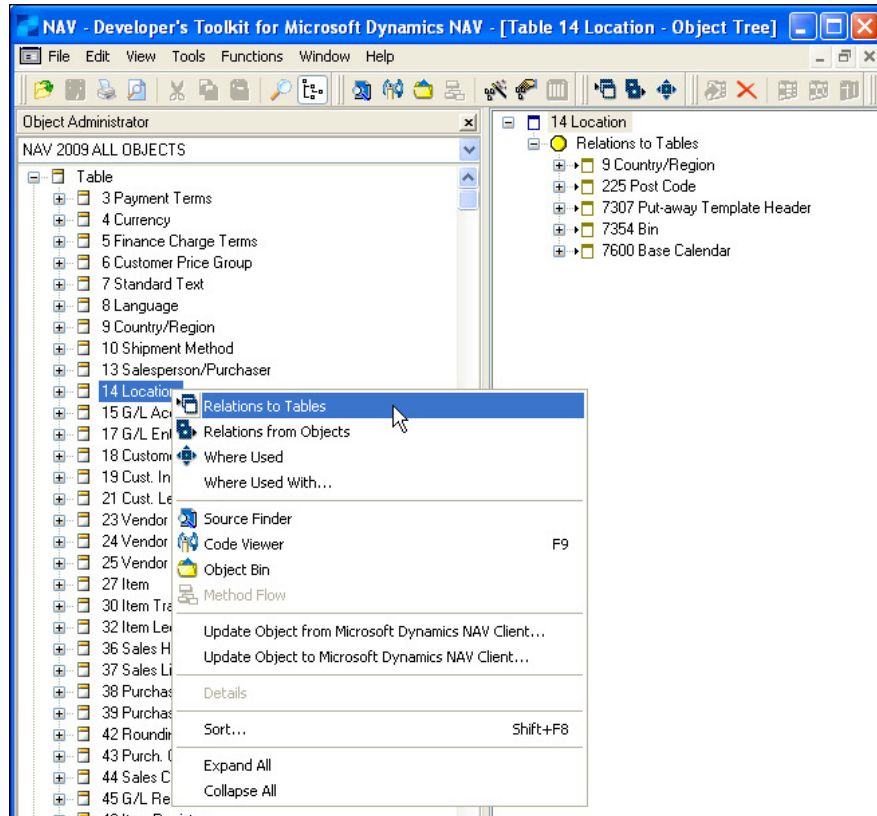
- The Source Analyzer tools are the more general purpose set of tools. Once you have loaded the source information for all your objects into the Developer's Tools database, you will be able to quickly generate a variety of useful code analyses. The starting point for your code analyses will be the **Object Administrator** view as shown in the following screenshot:



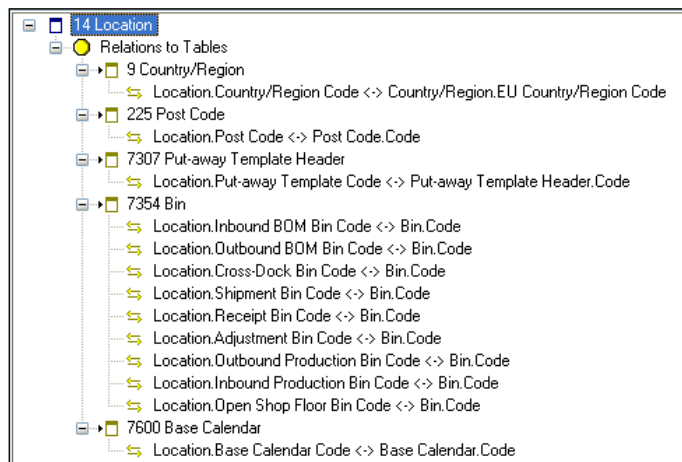
When you get to this point, it's worthwhile experimenting with various menu options for each of the object types to get comfortable with the environment and how the tools work. Not only are there several tool options, but multiple viewing options. Some will be more useful than others depending on the specifics of the modification task you are addressing as well as your working habits.

## Relations to Tables

With rare exceptions, table relations are defined between tables. The Toolkit allows you to select an object and request analysis of the defined relations between elements in that object and various tables. As a test of how the **Relations to Tables** analysis works, we will expand our **Table** entry in the **Object Administrator** to show all the tables. Then we will choose the **Location** table, right-click, and choose the option to view its **Relations to** other **Tables** with the result shown in the following screenshot:



If we want to see more detail, we can right-click on the **Location** table name in the right window, choose the **Expand All** option, and see the results as shown in the following screenshot:

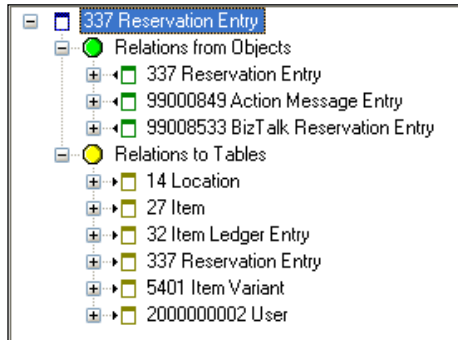


This shows us the **Relations to Tables**, with the relating (from) field and the related (to) field both showing in each line.

## Relations from Objects

If you are checking to see what objects have a relationship pointing back to a particular table (the inverse of what we just looked at), you can find that out in essentially the same fashion. Right-click on the table of interest and choose the **Relations from Objects** option. If you wanted to see both sets of relationships in the same display, you can right-click on the table name in the right window and choose the **Relation to Tables** option.

At that point your display will show both sets of relationships as shown in the following screenshot for the table Reservation Entry:



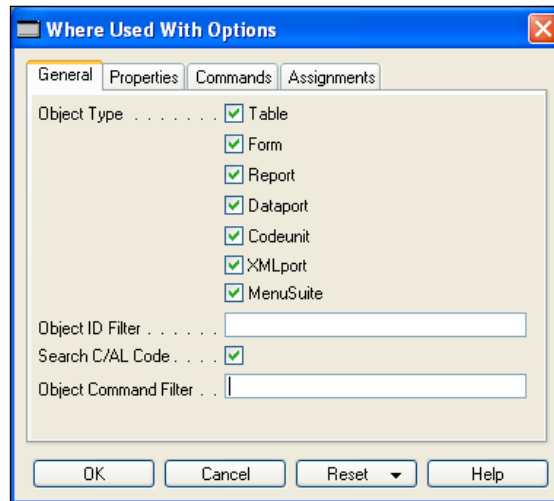
## Source Access

On any of these screens you could select one of the relationships and drill down further into the detail of the underlying C/AL code. There is a search tool, the **Source Finder**. When you highlight one of the identified relationships and access the **Code Viewer**, the Toolkit will show you the object code where the relationship is defined.

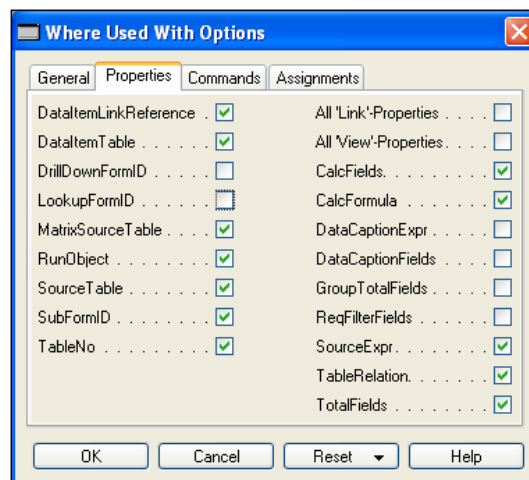
## Where Used

The Developer's Toolkit contains other tools that are also quite valuable to you as a developer. The idea of **Where Used** is fairly simple: list all the places where an element is used within the total library of source information. There are two different types of **Where Used**.

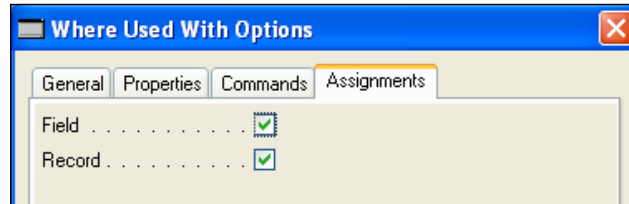
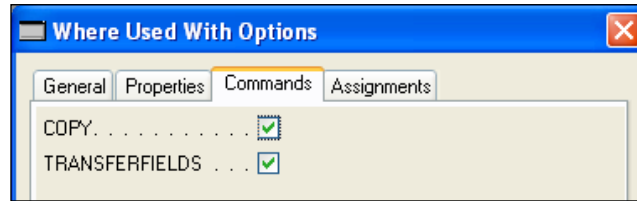
- The Toolkit's first type of **Where Used** is powerful because it can search for uses of whole tables or key sequences or individual fields. Many developers also use other tools (primarily developer's text editors) to accomplish some of this. However, the Developer's Toolkit is specifically designed for use with C/AL and C/SIDE.
- The second type of "Where Used" is **Where Used With**. This version of the Toolkit **Where Used** tool allows you to focus the search. Selecting the **Where Used With Options** brings up the screen in the following screenshot. As you can see, the degree of control you have over the search is extensive.



Screenshots of the other three tabs of the **Where Used With Options** form follow:



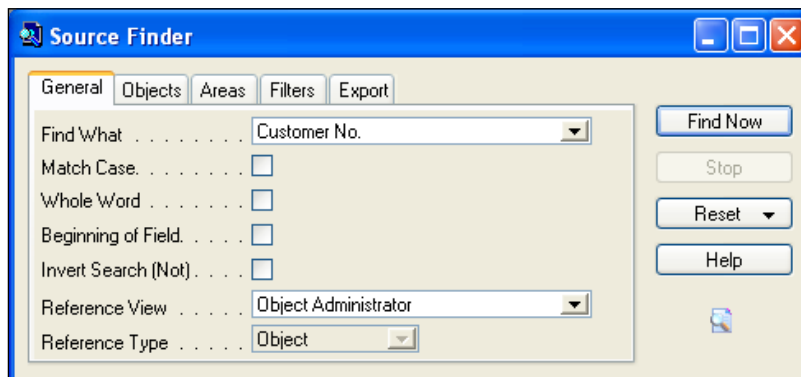




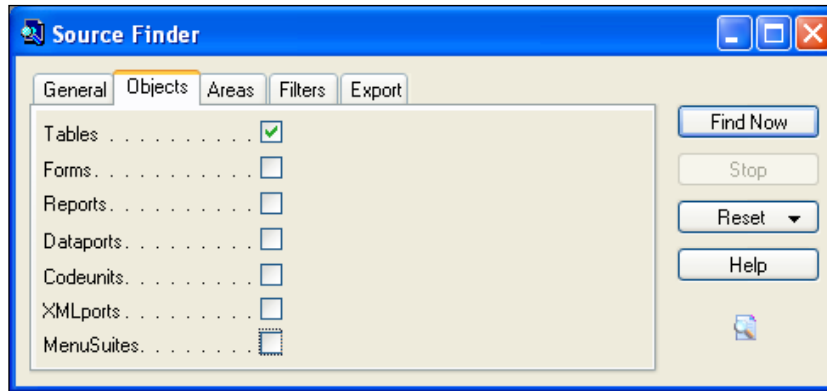
## Trying it out

To really appreciate the capabilities and flexibilities of the Developer's Toolkit, you must work with it to address a real-life task. For example, what if your firm was in a market where the merger of firms was a frequent occurrence? In order to manage this, the manager of accounting might decide that the system needs to be able to merge the data for two customers, including accounting and sales history under a single customer number. If you do that, you must first find all the instances of the **Customer No.** referenced in keys of other tables. The tool to do this in the Developer's Toolkit is the **Source Finder**.

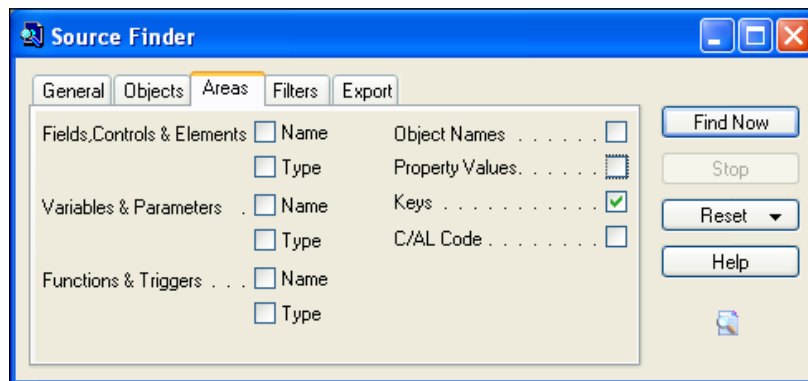
Calling up the **Source Finder**, first you **Reset** all fields by clearing them. Then enter the name of the field you are looking for, in this case that is **Customer No.**, as shown in the following screenshot:



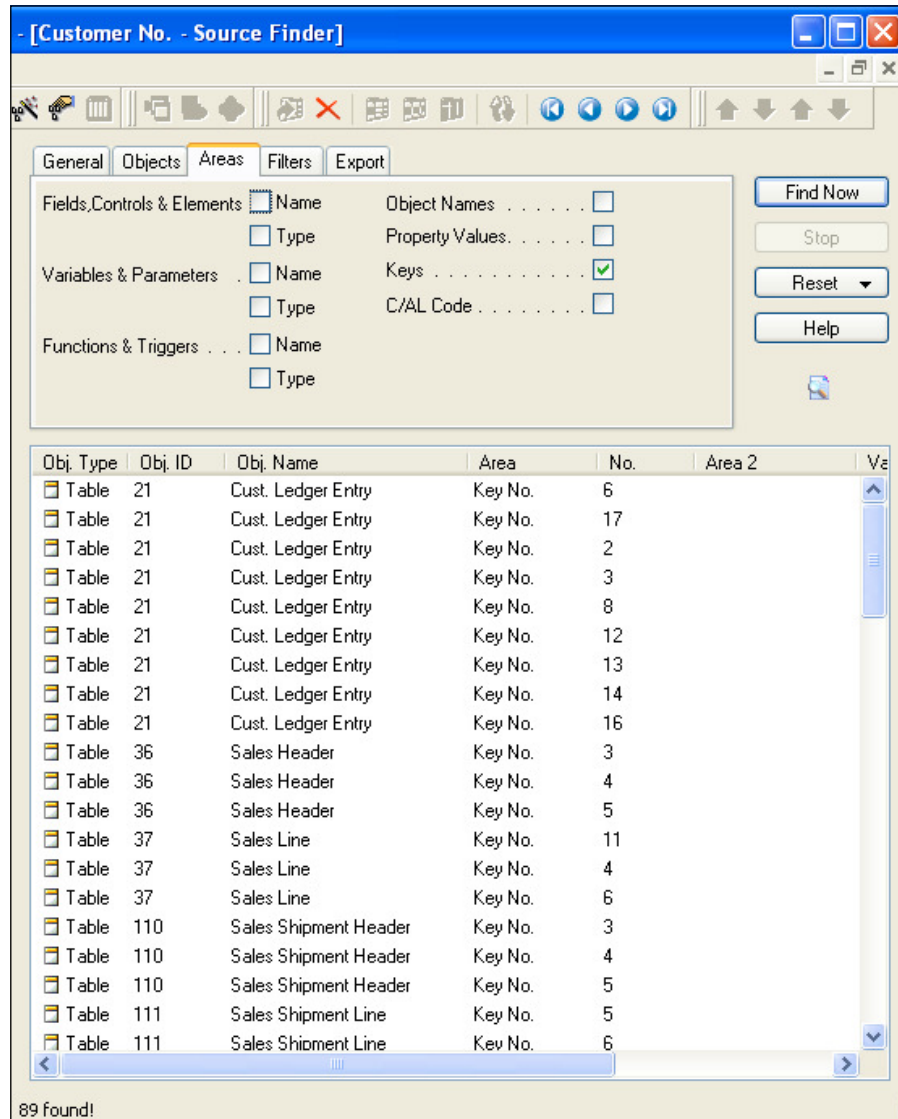
Now specify you are only looking for information contained in **Tables**, as shown in the following screenshot:



Next, specify that the search should only be in **Keys**, as shown in the following screenshot:



Your initial results will look like those in the following screenshot:



This data can be further constrained through the use of **Filters** (for example to find only Key 1 entries) and can be sorted by clicking on a column head.

Of course, as mentioned earlier, it will help you to experiment along the way. Don't make the mistake of thinking the Developer's Toolkit is the only tool you need to use. At the same time, don't make the mistake of ignoring this tool just because it won't do everything.

## Working in exported text code

As mentioned a little earlier, some developers export objects into text files, then use a text editor to manipulate them. Let us take a look at an object that has been exported into text and imported into a text editor.

We will use one of the tables that are part of our ICAN development, the Donor Type table, 50001 as shown in the following screenshot:

```

OBJECT Table 50001 Donor Type
{
 OBJECT-PROPERTIES
 {
 Date=03/21/09;
 Time=[9:24:25 PM];
 Modified=Yes;
 Version List=;
 }
 PROPERTIES
 {
 LookupFormID=Form50002;
 DrillDownFormID=Form50002;
 }
 FIELDS
 {
 { 10 ; ;Code ;Code10 }
 { 20 ; ;Description ;Text30 }
 }
 KEYS
 {
 { ;Code ;Clustered=Yes }
 }
 FIELDGROUPS
 {
 }
 CODE
 {
 BEGIN
 END.
 }
}

```

The general structure of all exported objects is similar, with differences that you would expect for the different objects. For example, Table objects have no Sections, but Report objects do. You can also see here that this particular table contains no C/AL-coded logic, as those statements would be quoted in the text listing.

You can see by looking at this table object text screenshot that you could easily search for instances of the string `Code` throughout the text export of the entire system, but it would be more difficult to look for references to the **Donor Type** form/page, `Form50002`. And, while you can find the instances of `Code` with your text editor, it would be quite difficult to differentiate those instances that relate to the **Donor Type** table from those in any other table. This includes those that have nothing to do with our ICAN system enhancement, as well as those simply defined in an object as Global Variables. However, the Developer's Toolkit can make that differentiation.

If you were determined to use a text editor to find all instances of `"Donor Type".Code`, you could do the following:

Rename the field in question to something unique. C/SIDE will rename all the references to this field. Then export all the sources to text followed by using your text editor (or even Microsoft Word) to find the unique name. You must either remember to return the field in the database to the original name or you must be working in a temporary "work copy" of the database, which you will shortly discard. Otherwise, you will have quite a mess.

One task that needs to be done occasionally is to renumber an object or to change a reference inside an object that refers to a no longer existing element. The C/SIDE editor may not let you do that easily, or in some cases, not at all. In such a case, the best answer is to export the object into text, make the change there and then import it back in as modified. Be careful though. When you import a text object, C/SIDE does not check to see if you are overwriting another instance of that object number. C/SIDE makes that check when you import a `fob` (that is a compiled object) and warns you. If you must do renumbering, you should check the NAV forums on the Internet for the renumbering tools that are available there.

Theoretically, you could write all of your C/AL code with a text editor and then import the result. Given the difficulty of such a task and the usefulness of the tools embedded in C/SIDE, such an approach would be foolish. However, there are occasions when it is very helpful to simply view an object "flattened out" in text format. In a report where you may have overlapping logic in multiple data items and in several control triggers as well, the only way to see all the logic at once is in text format.

You can use any text editor you like, Notepad or Word or one of the visual programming editors; the exported object is just text. You need to cope with the fact that when you export a large number of objects in one pass, they all end up in the same text file. That makes the exported file relatively difficult to use. The solution is to split that file into individual text files, named logically, one for each NAV object. There are several freeware tools to do just that, available from the NAV forums on the Internet.



Two excellent NAV forums are [www.mibuso.com](http://www.mibuso.com) and [www.dynamicsuser.net](http://www.dynamicsuser.net).

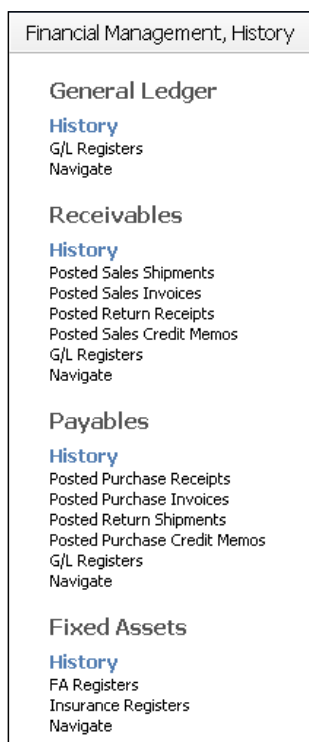
## Using Navigate

Navigate is an often under-appreciated tool both for the user and for the developer. Our focus is on its value to the developer. You might enhance your extension of the NAV system by expanding the coverage of the Navigate function.

## Testing with Navigate

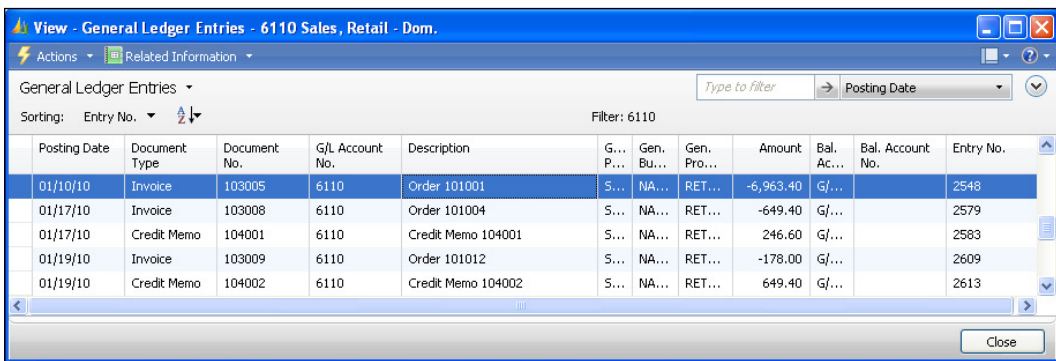
Navigate (Page 344) searches for and displays the number and types of all the associated entries for a particular posting transaction. The term "associated", in this case, is defined as those entries having the same **Document Number** and **Posting Date**.

Navigate can be called from the **Navigate** action, which appears on each screen that displays any of the entries that a Navigate might find and display. It can also be called directly from various Navigate entries in Action lists. These are generally located within History menu groups as shown in the following screenshot:



If you invoke the Navigate page using the menu action item, you must enter the **Posting Date** and **Document Number** for the entries you wish to find. Alternately, you can enter a **Business Contact Type** (Vendor or Customer), a **Business Contact No.** (Vendor No. or Customer No.), and optionally, an **External Document No.** There are occasions when this option is useful, but the **Posting Date + Document No.** option is much more frequently useful.

Instead of seeking out a Navigate page and entering the critical data fields, it is much easier to call Navigate from a **Navigate** action on a page showing data. In this case, you just highlight a record and click on **Navigate** to search for all the related entries. In the following example, the first **General Ledger Entry** displayed is highlighted.



View - General Ledger Entries - 6110 Sales, Retail - Dom.

Actions Related Information

General Ledger Entries

Filter: 6110

Sorting: Entry No.

| Posting Date | Document Type | Document No. | G/L Account No. | Description        | G... P... | Gen. BU... | Gen. Pro... | Amount    | Bal. Ac... | Bal. Account No. | Entry No. |
|--------------|---------------|--------------|-----------------|--------------------|-----------|------------|-------------|-----------|------------|------------------|-----------|
| 01/10/10     | Invoice       | 103005       | 6110            | Order 101001       | S...      | NA...      | RET...      | -6,963.40 | G/...      |                  | 2548      |
| 01/17/10     | Invoice       | 103008       | 6110            | Order 101004       | S...      | NA...      | RET...      | -649.40   | G/...      |                  | 2579      |
| 01/17/10     | Credit Memo   | 104001       | 6110            | Credit Memo 104001 | S...      | NA...      | RET...      | 246.60    | G/...      |                  | 2583      |
| 01/19/10     | Invoice       | 103009       | 6110            | Order 101012       | S...      | NA...      | RET...      | -178.00   | G/...      |                  | 2609      |
| 01/19/10     | Credit Memo   | 104002       | 6110            | Credit Memo 104002 | S...      | NA...      | RET...      | 649.40    | G/...      |                  | 2613      |

Close

After clicking on the **Navigate** action, the **Navigate** page will pop up, filled in, with the completed search, and will look similar to the following screenshot:

**Edit - Navigate**

Actions

Find Show Print

Process

**General**

Document No.: 103005

Posting Date: 01/10/10

**Source**

Document Type: Posted Sales Invoice

Source Type: Customer

Source No.: 10000

Source Name: The Cannon Group PLC

**Document Entry**

| Table Name                 | No. of Rec... |
|----------------------------|---------------|
| Posted Sales Invoice       | 1             |
| G/L Entry                  | 4             |
| VAT Entry                  | 1             |
| Cust. Ledger Entry         | 1             |
| Detailed Cust. Ledg. Entry | 1             |
| Value Entry                | 6             |

General External Item Tr... Close

Had we accessed the **Navigate** page through one of the menu entries, we would have filled in the **Document No.** and **Posting Date** fields and clicked on **Find**. As you can see here, the **Navigate** form shows a list of related, posted entries including the one we highlighted to invoke the Navigate function. If you click on one of the items in the **Table Name** list at the bottom of the page, you will see an appropriately formatted display of the chosen entries.



For the **G/L Entry** table in this form, you would see a result like the following screenshot. Note that all the **G/L Entry** are displayed for same **Posting Date** and **Document No.**, matching those specified at the top of the **Navigate** page.

**Edit - Navigate**

Actions: Find, Show, Print

Process

General

Document No.: 103005  
Posting Date: 01/10/10

Source

Document Type: Posted Sales Invoice  
Source Type: Customer  
Source No.: 10000  
Source Name: The Cannon Group PLC

Document Entry

| Table Name                 | No. of Rec... |
|----------------------------|---------------|
| Posted Sales Invoice       | 1             |
| G/L Entry                  | 4             |
| VAT Entry                  | 1             |
| Cust. Ledger Entry         | 1             |
| Detailed Cust. Ledg. Entry | 1             |
| Value Entry                | 6             |

General External Item Tr... Close

**General Ledger Entries - 6110 Sales, Retail - Dom.**

Related Information

Entries

Filter: 6110

| Document Type | Document No. | G/L Account No. | Description        | G... P... | Gen. Bu... | Gen. Pro... | Amount    |
|---------------|--------------|-----------------|--------------------|-----------|------------|-------------|-----------|
| Invoice       | 103005       | 6110            | Order 101001       | S...      | NA...      | RET...      | -6,963.40 |
| Invoice       | 103008       | 6110            | Order 101004       | S...      | NA...      | RET...      | -649.40   |
| Credit Memo   | 104001       | 6110            | Credit Memo 104001 | S...      | NA...      | RET...      | 246.60    |
| Invoice       | 103009       | 6110            | Order 101012       | S...      | NA...      | RET...      | -178.00   |

**View - General Ledger Entries - 6110 Sales, Retail - Dom.**

Actions: Related Information

General Ledger Entries

Sorting: Document No., Posting Date

Filter: 103005 • 01/10/10

| Gen. Postin... | Gen. Bus. ... | Gen. Prod. ... | Amount    | Bal. Accou... | Bal. Accou... | Entry No. |
|----------------|---------------|----------------|-----------|---------------|---------------|-----------|
| Sale           | NATIONAL      | RETAIL         | -6,963.40 | G/L Account   |               | 2548      |
|                |               |                | -1,653.81 | G/L Account   |               | 2549      |
|                |               |                | 348.17    | G/L Account   |               | 2550      |
|                |               |                | 8,269.04  | G/L Account   |               | 2551      |

Close

You may ask "Why is this application page being discussed in a section about C/AL debugging?" The answer is: "When you have to test, you need to check the results. When it is easier to do a thorough check of your test results, your testing will go faster and be of higher quality". Whenever you make a modification that will affect any data that could be displayed through the use of **Navigate**, it will quickly become one of your favorite testing tools.

## Modifying for Navigate

If your modification creates a new table that will contain posted data and the records contain both Document No. and Posting Date fields, you can include this new table in the **Navigate** function.

The C/AL Code for **Posting Date + Document No.** Navigate functionality is found in the FindRecords function trigger of Page 344 – Navigate. The following screenshot illustrates the segment of the Navigate CASE statement code for the Check Ledger Entry table:

```

IF CheckLdgEntry.READPERMISSION THEN BEGIN
 CheckLdgEntry.RESET;
 CheckLdgEntry.SETCURRENTKEY("Document No.", "Posting Date");
 CheckLdgEntry.SETFILTER("Document No.", DocNoFilter);
 CheckLdgEntry.SETFILTER("Posting Date", PostingDateFilter);
 InsertIntoDocEntry(
 DATABASE::"Check Ledger Entry", 0, CheckLdgEntry.TABLECAPTION, CheckLdgEntry.COUNT);
END;

```

The code checks `READPERMISSION`. If that is enabled for this table, then the appropriate filtering is applied. Next, there is a call to the `InsertIntoDocEntry` function, which fills the temporary table that is displayed in the **Navigate** page. If you wish to add a new table to the **Navigate** function, you must replicate this code for your new table. In addition, you must add the code that will call up the appropriate page to display the records that **Navigate** finds. This code should be inserted in the `ShowRecords` function trigger of the **Navigate** page, similar to the lines in the following screenshot:

```

DATABASE::"Check Ledger Entry":
 FORM.RUN(0, CheckLdgEntry);

```

Making a change like this, when appropriate, will not only provide a powerful tool for your users, but will also provide a powerful tool for you as a developer.

## The C/SIDE Debugger

C/SIDE has a built-in **Debugger** which runs in the Classic Client environment. It is very helpful in tracking down the location of bugs in logic and processes. There are two basic usages of the available debugger. The first is identification of the location of a run-time error. This is fairly simple process, accomplished by setting the debugger (from the **Tools** Menu) to **Active** with the **Break on Triggers** option turned off, as shown in the following screenshot. When the run-time error occurs, the debugger will be activated and displayed exactly where the error is occurring.

The second option is the support of traditional tracing of logic. Use of the Debugger for this purpose is hard work. It requires that you, the developer, have firmly in mind how the code logic is supposed to work, what the values of variables should be under all conditions, and the ability to discern when the code is not working as intended.

The Debugger allows you to see what code is being executed, either on a step-by-step basis (by pressing *F8*) or trigger by trigger (by pressing *F5*). You can set your own **Breakpoint** ((by pressing *F9*), points at which the Debugger will break the execution so you can examine the status of the system. The method by which execution is halted in the Debugger doesn't matter (so long as it is not by a run-time error); you have a myriad of tools with which to examine the status of the system at that point.

In spite of its limitations, the C/SIDE Debugger is quite a useful tool and it is the only debugger that works in C/SIDE. Learning to use the C/SIDE Debugger will pay off. The best way to learn it is through hands-on practice. The Debugger is documented reasonably well in the **C/SIDE Reference Guide** Help. Beyond studying that documentation, the best way to learn more about the C/SIDE debugger is to experiment with its use.

## The C/SIDE Code Coverage tool

Code Coverage is another "Classic Client only" debugging tool. Code Coverage tracks code as it is executed and logs it for your review and analysis. Code Coverage is accessed from the **Tools | Debugger** option. When you invoke Code Coverage, it simply opens the Code Coverage form. Start Code Coverage by clicking on Start, and stop it by returning to the form via the Windows menu option, as the Code Coverage form will remain open while it is running. The **C/SIDE Reference Guide** provides some information on how to interpret the information collected.

Just like the C/SIDE Debugger, there is no substitute for experimenting to learn more about using Code Coverage. Code Coverage is a tool for gathering a volume of data about the path taken by the code while performing a task or series of tasks. This is very useful for two different purposes. One is simply to determine what code is being executed. But this tracking is done in high speed with a log file, whereas if you do the same thing in the debugger, the process is excruciatingly slow and you have to log the data manually.

The second use is to identify the volume of use of routines. By knowing how often a routine is executed within a particular process, you can make an informed judgement about what routines are using up the most processing time. That, in turn, allows you to focus any speed-up efforts on the code that is used the most. This approach will help you to realize the most benefit for your code acceleration work.

## Client Monitor

Client Monitor is a performance analysis tool. It can be very effectively used in combination with Code Coverage to identify what is happening, in what sequence, how it is happening and how long it is taking. Before you start Client Monitor, there are several settings you can adjust to control its behavior, some specific to the SQL Server environment.

**Client Monitor** is accessed from **Tools | Client Monitor**. **Client Monitor** helps identify the code that is taking the major share of the processing time so that you can focus on code design optimization efforts. In the SQL Server environment, Client Monitor will help to see what calls are being made by SQL Server and clearly identify problems with improperly defined or chosen indexes and filters.

If you are familiar with T-SQL and SQL Server calls, the Client Monitor output will be even more meaningful to you. In fact, you may decide to combine these tools with the SQL Server Error Log for a more in-depth analysis of speed or locking problems. Look at the **Performance Troubleshooting Guide** from Microsoft for additional information and guidance. This manual also contains instructions on how to use Excel pivot tables for analyzing the output from Client Monitor.

## Debugging NAV in Visual Studio

Debugging code running in the Role Tailored Client requires using Visual Studio based debugging tools. The VS debugging function setup is described in relatively complete detail in the C/SIDE Reference Guide Help entitled "**Walkthrough: Debugging an Action on a Page**". It is also described in one of the several excellent NAV blogs on the Net. As of the time of this writing, there are some confusing sections in this help, so we'll briefly review the basic steps.

The first step in setting up VS debugging for NAV is to enable debugging via a setting in the Customsettings.config file that controls various actions of the NAV Server. This file's location and the setting to change are described in the Help. After you change this setting, you need to Stop and Start the NAV Server Service before the change will take effect.

After the VS debugger has been enabled, each time you start up the RTC, all the C# code for all of the NAV objects will be generated and stored. The storage location for the C# code will vary depending on your system setup (OS version and perhaps other factors) – the original Help overlooks this variability. Look for a directory containing the string `...\60\Server\MicrosoftDynamicsNav\Server\source\...` The **source** directory will contain subdirectories of Codeunit, Page, Record, Report, XMLport. Each directory contains a full set of appropriate files with a `.cs` extension containing the generated C# code. If you are familiar with C#, you may find it useful to study this code.

Visual Studio must then be attached to NAV Server Service before a breakpoint can be set. The referenced Walkthrough uses an example for breakpoint setting of creating a Codeunit with an easily identifiable piece of code. The point is to be able to identify a good spot in the C# code to set your breakpoint so that you can productively use the VS Debugger.

## Dialog function debugging techniques

In previous chapters, we have discussed some other simple debugging techniques that you can productively use when developing in C/AL and C/SIDE. Sometimes these simpler methods are more productive than the more sophisticated tools, because you can set up and test quickly, resolve the issue (or answer a question), and move on. All the simpler methods involve using one of the C/AL `DIALOG` functions such as `MESSAGE`, `CONFIRM`, `DIALOG`, or `ERROR`. All of these have the advantage of working well in the RTC environment. If you need detailed RTC performance information, the Code Coverage and Client Monitor tools do not work with the RTC. In that case, you should use one of the following tools/techniques to provide performance data (until a better method is provided by Microsoft).

### Debugging with MESSAGE

The simplest debug method is to insert `MESSAGE` statements at key points in your logic. It is very simple and, if structured properly, provides you a simple "trace" of the code logic path. You can number your messages to differentiate them and display any data (in small amounts) as part of a message.

The disadvantage is that `MESSAGE` statements do not display until processing either terminates or is interrupted for user interaction. If you force a user interaction at some point, then your accumulated messages will appear prior to the interaction. The simplest way to force user interaction is a `CONFIRM` message in the format as follows:

```
IF CONFIRM ('Test 1',TRUE) THEN;
```

### Debugging with CONFIRM

If you want to do a simple trace but want every message to be displayed as it is generated (that is have the tracking process move at a very measured pace), you could use `CONFIRM` statements for all the messages. You will then have to respond to each one before your program will move on, but sometimes that is what you want.

### Debugging with DIALOG

Another tool that is useful for certain kinds of progress tracking is the `DIALOG` function. This function is usually set up to display a window containing a small number of variable values. As processing progresses, the values are displayed in real time. This can be useful in several ways. A couple of examples follow:

- Simply tracking progress of processing through a volume of data. This is the same reason you would provide a `DIALOG` display for the benefit of the user. The act of displaying does slow down processing somewhat. During debugging that may or may not matter. In production it is often a concern, so you may want to update the `DIALOG` display occasionally, not on every record.
- Displaying indicators when processing reaches certain stages. This can be used as a very basic trace with the indicators showing the path taken so you may gauge the relative speed of progress through several steps. For example, you might have a six-step process to analyze. You could define six tracking variables and display all of them in the `DIALOG`.

In this case, each tracking variable is initialized with values that are dependent on what you are tracking, but generally each would be different (for example, A1, B2000, C300000, and so on.). At each step of your process, you would update the contents and display the current state of one or all of the variables. This can be a very visual and intuitive guide for how your process is operating.

## Debugging with text output

You can build a very handy debugging tool by outputting the values of critical variables or other informative indicators of progress either to an external text file or to a table created for this purpose. This approach allows you to run a considerable volume of test data through the system, tracking some important elements while collecting data on the variable values, progress through various sections of code, and so on. You can even timestamp your output records so that you can use this method to look for processing speed problems (a very minimalist code coverage replacement).

Following the test run, you can analyze the results of your test more quickly than if you were using displayed information. You can focus on just the items that appear most informative and ignore the rest. This type of debugging is fairly easy to set up and to refine, as you identify the variables or code segments of most interest. This approach can be combined with the following approach using the `ERROR` statement if you output to an external text file, then close it before invoking the `ERROR` statement, so that its contents are retained following the termination of the test run.

## Debugging with `ERROR`

One of the challenges of testing is maintaining repeatability. Quite often you need to test several times using the same data, but the test changes the data. If you have a small database, you can always back up the database and start with a fresh copy each time. But that can be inefficient and, if the database is large, impractical. One alternative is to conclude your test with an `ERROR` function. This allows you to test and retest with exactly the same data.

The `ERROR` function forces a run-time error status, which means the database is not updated (it is rolled back to the status at the beginning of the process). This works well when your debugging information is provided by using the Debugger or by use of any of the `DIALOG` functions just mentioned prior to the execution of the `ERROR` function. If you are using `MESSAGE` to generate debugging information, you could execute a `CONFIRM` immediately prior to the `ERROR` statement and be assured that all of the messages are displayed. Obviously this method won't work well when your testing validation is dependent on checking results using Navigate or your test is a multi-step process such as order entry, review, and posting. But in applicable situations, it is a very handy technique for repeating a test with minimal effort.

When testing the posting of an item, it can be useful to place the test-concluding `ERROR` function just before the point in the applicable Posting codeunit where the process would otherwise be completed successfully.

## C/SIDE test driven development

New in NAV 2009 SP1 is a C/AL Testability feature set designed to support C/AL development to be test driven. **Test-driven development** is an approach where the application tests are defined prior to the development of the application code. In an ideal situation, the code supporting application tests is written prior to, or at least at the same time as, the code implementing the target application function written.

The C/AL Testability feature provides two more types of Codeunits – Test Codeunits and Test Running Codeunits. Test Codeunits contain Test methods, C/AL code to support Test methods. Test Runner Codeunits are used to invoke Test Codeunits, thus providing test execution management, automation and integration. Test Runner Codeunits have two special triggers, each of which run in separate transactions, so the test execution state and results can be tracked.

- `OnBeforeTestRun` is called before each test. It allows defining, via a Boolean, whether or not the test should be executed.
- `OnAfterTestRun` is called when each test completes and the test results are available. This allows the test results to be logged, or otherwise processed via C/AL code.

Among the ultimate goals of the C/AL Testability feature are:

- The ability to run suites of application tests automated and regressively:
  - **Automated** means that a defined series of tests could be run and results recorded, all without user intervention.
  - **Regressively** means that the test can be run repeatedly as part of a new testing pass to make sure that features previously tested are still in working order.

- The ability to design tests in an "atomic" way, matching the granularity of the application code. In this way, the test functions can be focused and simplified. This in turn allows for relatively easy construction of a suite of tests and, in some cases, reuse of test codeunits (or at least reuse of the structure of previously created Test Codeunits).
- The ability to develop and run the Test and Test Runner Codeunits within the familiar C/SIDE environment. The code for developing these testing codeunits is C/AL.
- Once the testing codeunits have been developed, the actual testing process should be simple and fast in order to run and evaluate results.

Both positive and negative testing are supported. **Positive testing** is that where you look for a specific result, a correct answer. **Negative testing** is where you check that errors are presented when expected, especially when data or parameters are out of range. The testing structure is designed to support the logging of test results, both failures and success, to tables for review, reporting and analysis.

A function property defines functions within Test Codeunits to be either Test or TestHandler or Normal. Another function property, TestMethodType, allows the definition of a variety of Test Function types to be defined. TestMethodType property options include the following which are designed to handle User Interface events without the need for a person to intervene:

- CatchMessage—catches and handles the MESSAGE statement
- ConfirmDialogHandler—catches and handles CONFIRM dialogs
- StrMenuHandler—catches and handles STRMENU menu dialogs
- FormRunHandler—catches and handles Forms/Pages that are executed via RUN
- FormRunModalHandler—catches and handles Forms/Pages that are executed via RUNMODAL
- ReportHandler—handles reports

Use of the C/SIDE Test Driven Development approach will work along the following lines:

- Define an application function specification
- Define the application technical specification
- Define the testing technical specification including both Positive and Negative tests



- Develop Test and Test Running codeunits (frequently only one or a few Test Running codeunits will be required)
- Develop Application objects
- As soon as feasible, begin running Application object tests by means of the Test Running codeunits, and logging test results for historical and analytical purposes
- Continue the development – testing cycle, updating the tests and the application as appropriate throughout the process
- At the end of successful completion of development and testing, retain all the Test and TestRunning codeunits for use in regression testing the next time the application must be modified or upgraded

## Summary

In this chapter, we reviewed a number of tools and techniques aimed at making your life as a developer in the NAV environment easier and more efficient. Many of these topics require more study and some hands-on practice by you. Among the topics we covered were functions that you can use "as is" and functions that you can use as models for your own code. We reviewed some guidelines for documenting your modifications and briefly discussed dealing with Multi-Language and Multi-Currency compatibility. Finally, we reviewed a host of handy code analysis and debugging techniques including use of the Developer Toolkit, working in objects exported as text, using Navigate, using the Debuggers and associated tools, and some handy tips on other creative techniques.

By this point in the book, we have covered many of the core elements of NAV development. You should be just about ready to begin your own development project.

In the next chapter, we are going to consider a number of other important NAV design and development concepts, as well as features and tools for integrating and extending NAV.

---

## Review questions

1. In NAV, data can be entered into Journals, edited and then Posted to the Ledgers or, if the user is very careful, can be entered directly into the Ledger. True or False?
2. Role Center pages can contain actions in the Cue Groups, in the Navigation Pane, and in the Action Menus. True or False?
3. The Action items for the Departments button come from what source? Choose one:
  - a. Action Menu entries
  - b. Cue Group definitions
  - c. MenuSuite objects
  - d. Navigation Pane objects
4. Custom C/AL code is not allowed to call functions that exist in the base Microsoft created NAV objects. True or False?
5. Which one of the following is a library of functions for various purposes widely used throughout the NAV system? Choose 1.
  - a. Codeunit 412 – Common Dialog Management
  - b. Codeunit 408 – Dimension Management
  - c. Codeunit 396 – NoSeriesManagement
  - d. Codeunit 1 – Application Management
6. NAV's multi-language capability allows for an installation to have multiple languages active at one time. True or False?
7. Each of the following fully support NAV 2009 RTC development and debugging. Individually, identify True or False:
  - a. The NAV Developer's Toolkit – True or False?
  - b. The C/SIDE Debugger – True or False?
  - c. The C/SIDE Code Coverage Tool – True or False?
  - d. The Client Monitor – True or False?

8. The Navigate feature can be used for which of the following? Choose three:
  - a. Auditing by a professional accountant
  - b. User analysis of data processing
  - c. Reversing posting errors
  - d. Debugging
9. You can enhance the Navigate function to include new tables that have been added to the system as part of an enhancement. True or False?
10. The primary underlying code for NAV 2009 is C#. True or False?
11. Once a Role Center layout has been defined by the Developer, it cannot be changed by the users. True or False?
12. The external Microsoft application that is always integrated into a Role Center is which of the following? Choose one:
  - a. Excel
  - b. Outlook
  - c. Word
  - d. Exchange

# 9

## Extend, Integrate, and Design—into the Future

*Prediction is very difficult, especially if it's about the future – Niels Bohr*

*The best way to predict the future is to create it – Peter Drucker*

One of the challenges with any significant, useful tool is to learn how to use it in many situations, how to take advantage of its strengths to solve new problems, and how to smoothly integrate it into the environment in which it must operate. Fortunately, the designers of Dynamics NAV have, from the inception, understood that our system does not operate in a vacuum, but must interact and integrate with other systems.

As new versions of NAV are released, new capabilities for extending and integrating NAV are added to allow us to build new functionality that is a combination of the strengths of NAV and the capabilities of other tools and systems. In this chapter, we will look at how your NAV processes can interface with outside data sources and how the outside world (systems and users) can interface with and utilize NAV data and objects. We will also look at the NAV tools and capabilities that help us to integrate NAV with a variety of non-NAV systems.

### Interfaces

Some NAV systems must communicate with other software or even with hardware. Sometimes that communication is *Inside-Out* (that is, initiated by NAV), sometimes it is *Outside-In* (that is, initiated by the outside connection). It's not unusual for system-to-system communications to be a two-way street, a meeting of peers. To supply, receive, or exchange information with other systems (hardware or software), we need at least a basic understanding of the interface tools that are part of NAV.



It is critical to understand that, because of NAV's unique data structures (particularly FlowFields), it is very risky for an external system to access NAV data directly without using C/AL based routines as an intermediary.

## XMLports

**XML** is **eXtensible Markup Language**, a structured text format developed specifically to describe data to be shared by dissimilar systems. XML has become the default standard for communications between systems. To make handling XML-formatted data simpler and more error resistant, NAV has XMLports, a data import/export function that processes XML-formatted data. Use of XMLports allows the setup of XML-based communication with another system.

In addition to XML-formatted data, XMLports can also handle a wide variety of other text file formats. In Classic Client implementations of NAV (all versions), there is a separate Dataport object for handling non-XML text data and XMLports are limited to XML data. In the RTC environment, XMLports are the general purpose text data import / export tool.



Debugging of XMLports in the RTC environment requires use of the Visual Studio debugger.

XML data is text based, with each piece of information structured in one of two basic formats:

1. `<StartTag>data value</EndTag>` (an **Element** format)
2. `<Data Item Name = "data value">` (an **Attribute** format)

An Attribute must always be enclosed in quotation marks. Single or double quotes can be used for an Attribute. Elements are considered more general purpose than attributes, probably because they are easier to parse and generate simpler data structures when there are multiple entries created for one level. In general, it is good practice to use Elements for the core information being communicated and use Attributes for peripheral or descriptive information. Some references suggest that Elements should be used for data, Attributes for Metadata. Complex data structures are built up of combinations of these two formats.

For example:

```
<Table='Sample XML format'>
 <Record>
 <Data Item 1>12345</Data Item 1>
 <Data Item 2>23456</Data Item 2>
 </Record>
 <Record>
 <Data Item 1>987</Data Item 1>
 </Record>
 <Record>
 <Data Item 1>22233</Data Item 1>
 <Data Item 2>7766</Data Item 2>
 </Record>
</Table>
```

In this case, we have a set of data identified as a Table named 'Sample XML format', containing three Records, each Record containing data in one or two fields named Data Item 1 and Data Item 2. The data is in a clearly structured text format so it can be read and processed by any system that is prepared to read this particular XML format. If the field tags are well designed, the data is easily interpretable by humans as well. The key to successful exchange of data using XML is simply the sharing and common interpretation of the format between the transmitter and recipient of the information.

XML is a standard format in the sense that the data structure options are clearly defined. But it is very flexible in the sense that the identifying tag names in <> brackets and the related data structures that can be defined and handled are totally open ended. The specific structure and the labels are whatever the communicating parties decide they should be. The "rules" of XML only determine how the basic syntax shall operate.

XML data structures can be as simple as a flat file consisting of a set of identically formatted records or as complex as an order structure with headers containing a variety of data items, combined with associated detail lines containing their own variety of data items. If necessary for the application being addressed, an XML data structure can be even much more complicated.

## XMLport components

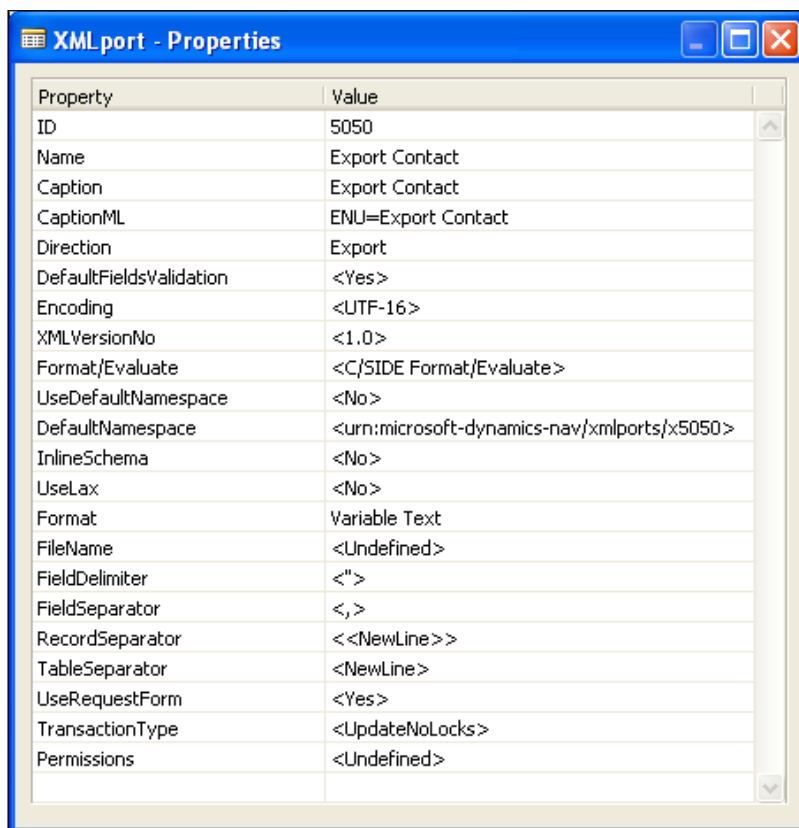
Although in theory XMLports can operate in both an import and an export mode, in practice, individual XMLport objects tend to be dedicated to either import or export. In general, this allows the internal logic to be simpler. XMLports consist of fewer components than Reports do, even though the general process of looping through and processing data is conceptually similar.

The components of XMLports are as follows:

- XMLport properties
- XMLport triggers
- XMLport lines (elements, attributes) aka nodes
  - Node properties (aka field properties)
  - Node triggers
- Request Page
  - Properties
  - Triggers
  - Controls
    - Properties
    - Triggers

## XMLport properties

XMLport properties are shown in the following screenshot of the **Properties** of the XMLport object 5050:



Descriptions of the individual properties follow:

- **ID:** The unique XMLport object number.
- **Name:** The name by which this XMLport is referred to within C/AL code.
- **Caption:** The name that is displayed for this XMLport; it defaults to the contents of the **Name** property.
- **CaptionML:** The **Caption** translation for a defined alternative language.
- **Direction:** This defines whether this XMLport can only **Import**, **Export**, or **<Both>**; the default is **<Both>**.
- **DefaultFieldsValidation:** This defines the default value (**Yes** or **No**) for the **FieldValidate** property for individual XMLport data fields. The default for this field is **Yes**, which would in turn set the default for individual field **FieldValidate** properties to **Yes**.



- **Encoding:** This defines the character encoding option to be used, **UTF-8** or **UTF-16**. **UTF-16** is the default. This is inserted into the heading of the XML document.
- **XMLVersionNo:** This defines to which version of XML the document conforms, **Version 1.0** or **1.1**. The default is **Version 1.0**. This is inserted into the heading of the XML document.
- **Format/Evaluate:** This can be **C/SIDE Format/Evaluate** or **XML Format Evaluate**. This property defines whether the external text data is (for imports) or will be (for exports) XML data types or C/SIDE data types. Default processing for all fields in each case will be appropriate to the defined data type. If the external data does not fit in either of these categories, then the XML data fields must be processed through a temporary table.

The temporary table processing approach reads the external data into text data fields with data conversion logic done in C/AL into data types that can then be stored in the NAV database. Additional information on this is available in the online C/SIDE Reference Guide (that is, the **Help** files).

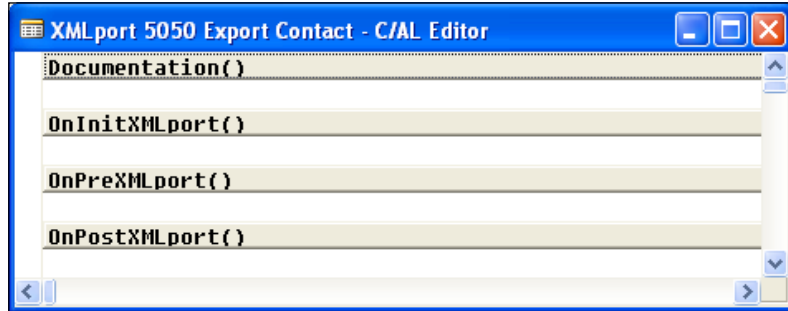
The default value for this property is **C/SIDE Format/ Evaluate**.

- **UseDefaultNamespace** and **DefaultNamespace:** These properties are provided to support compatibility with other systems which require the XML document to be in a specific namespace. **UseDefaultNamespace** defaults to **No**. A default namespace in the form of **URN (Uniform Resource Name** or, in this case, a **Namespace Identifier)** concluding with the object number of the XMLport is supplied for the **DefaultNamespace** property.
- **InlineSchema:** This property defaults to **No**. An inline schema allows the XML schema document (an XSD) to be embedded within the XML document. This can be used by setting the property to **Yes** when exporting an XML document, which will add the schema to that exported document.
- **UseLax:** This property defaults to **No**. Some systems may add information to the XML document, which is not defined in the XSD schema used by the XMLport. When this property is set to **Yes**, that extraneous material will be ignored, rather than resulting in an error.
- **Format:** This property has the options of XML, Variable Text or Fixed Text. It defaults to XML. This property controls the import/export data format that the XMLport will process. Choosing **XML** means that the processing will only deal with a properly formatted XML file. Choosing **Variable Text** means that the processing will only deal with a file formatted with delimiters set up as defined in the **FieldDelimiter**, **FieldSeparator**, **RecordSeparator**, and **TableSeparator** properties (such as CSV files). Choosing **Fixed Text** means that the each individual element and attribute must have its Width property set to a value greater than 0 (zero) and the data to be processed must be formatted accordingly.

- **FileName:** This can be filled with the predefined path and name of a specific external text data file to be either the source (for **Import**) or target (for **Export**) for the run of the XMLport or this property can be set dynamically. Only one file at a time can be opened, but the file in use can be changed during the execution of the XMLport (not often done).
- **FieldDelimiter:** This applies to **Variable Text** format external files only. It defaults to "<" – double quote, the standard for so-called "comma-delimited" text files. This property supplies the string that will be used as the starting delimiter for each data field in the text file. If this is an **Import**, then the XMLport will look for this string, then use the string following as data until the next **FieldDelimiter** string is found, terminating the data string. If this is an **Export**, the XMLport will insert this string at the beginning and end of each data field contents string.
- **FieldSeparator:** This applies to **VariableText** format external files only. Defaults to "<,>" – a comma, the standard for so-called "comma delimited" text files. This property supplies the string that will be used as the delimiter between each data field in the text file (looked for on **Imports** or inserted on **Exports**).
- **RecordSeparator:** This defines the string that will be used as the delimiter at the end of each data record in the text file. If this is an **Import**, the XMLport will look for this string to mark the end of each data record. If this is an **Export**, the XMLport will append this string at the end of each data record output. The default is <<NewLine>>, which represents any combination of CR (carriage return – ASCII value 13) and LF (line feed – ASCII value 10) characters.
- **DataItemSeparator:** This defines the string that will be used as the delimiter at the end of each Data Item (for example, each text file). The default is <<NewLine><NewLine>>.
- **UseRequestForm:** This determines whether a Request Page should be displayed to allow the user choice of Sort Sequence, entry of filters, and other requested control information. The options are **Yes** and **No**. The default is <Yes>. An XMLport Request Page only has the **OK** and **Cancel** options.
- **TransactionType:** This property identifies the XMLport processing Server Transaction Type as **Browse**, **Snapshot**, **UpdateNoLocks**, or **Update**. This is an advanced and seldom-used property. For more information, you can refer to the **C/SIDE Reference Guide** Help files and SQL Server documentation.
- **Permissions:** This property provides report-specific setting of permissions, which are rights to access data, subdivided into **Read**, **Insert**, **Modify**, and **Delete**. This allows the developer to define permissions that override the user-by-user permissions security setup.

## XMLport triggers

The XMLport has a limited set of triggers as shown in the following screenshot:



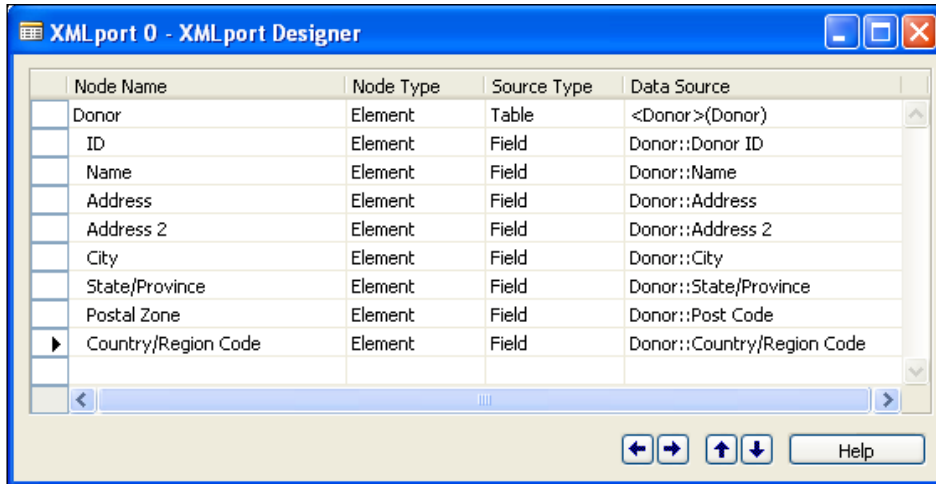
The XMLport trigger descriptions are as follows:

- **Documentation()** is for your documentation comments.
- **OnInitXMLport()** is executed once when the XMLport is loaded.
- **OnPreXMLport()** is executed once after the table views and filters have been set. Those can be reset here.
- **OnPostXMLport()** is executed once after all the data is processed, if the XMLport completes normally.

## XMLport data lines

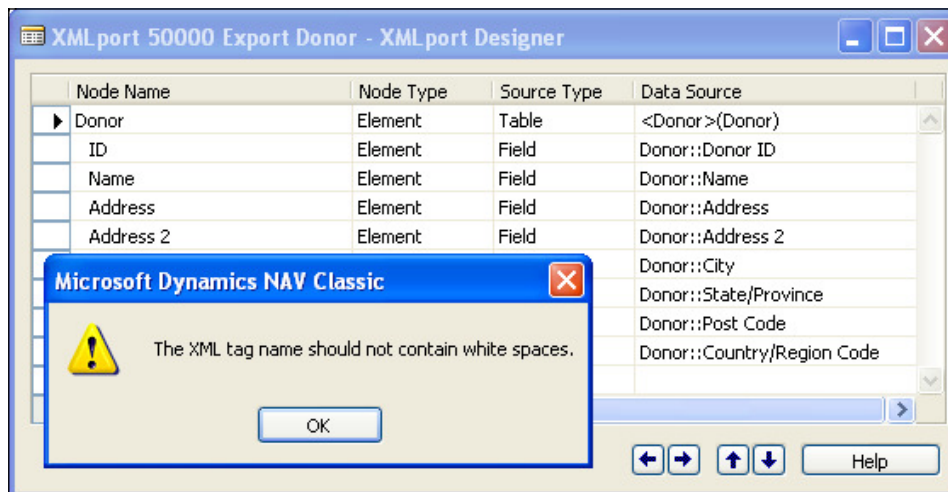
An XMLport can contain any number of data lines. The data lines are laid out in a strict hierarchical structure, with the elements and attributes mapping exactly, one for one, in the order of the data fields in the external text file, the XML document.

Let's create an XMLport to export Donor data from our ICAN application for use in another system. The first step is to define an XMLport line for the Donor table, then lines for each of the data fields we want to include. The initial design will look like the following screenshot:



This doesn't include everything we want to include, but let's save what we've created so far.

When we try to save our XMLport, we get an error message.

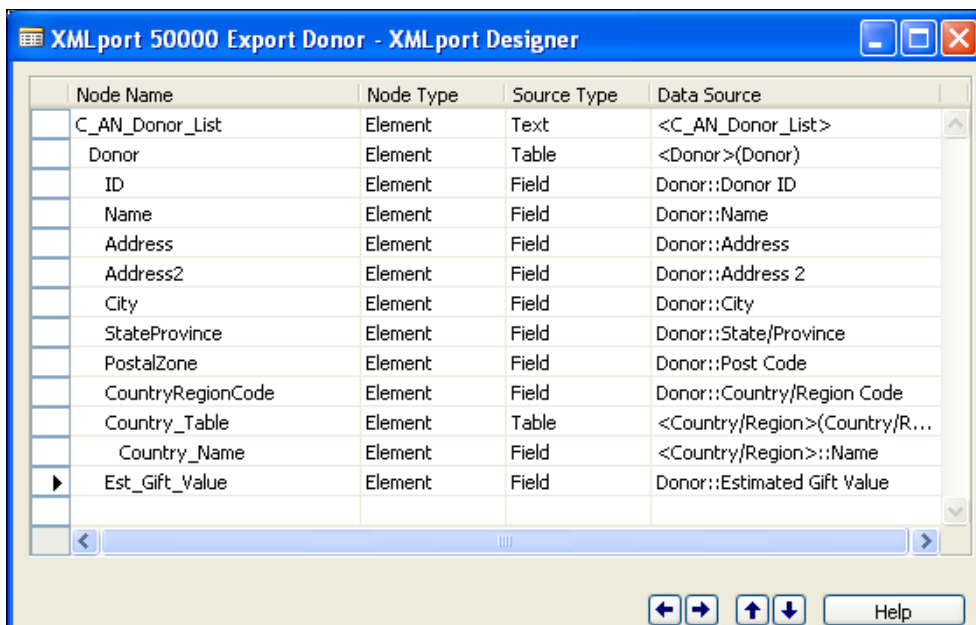


Based on that error message, we conclude that we need to remove the embedded blanks from the Node Names, so that **Address 2** becomes **Address2**. We do that and save again. This time we get a different error message.



This time it's a little harder to figure out the problem. Based on experience with other situations in NAV and elsewhere, we guess that the compiler is objecting to names that have special characters. So let's change **Country/RegionCode** to **CountryRegionCode**. This time it works. We assign the XMLport to 50000 named Donor Export.

Now let's proceed to add the other fields that we want. That includes the Country Name, which we'll get from the Country table and the Estimated Gift Value from the Donor table. This time, we'll make our names a little more readable by replacing the spaces with underscores. We will also add a header element that will help to identify our XML list, which requires indenting the rest of the elements on one level to be nested within the header element.



In order for the Country Name field to be properly selected, we need to set the **LinkTable** and **LinkFields** properties for the Country table in order to link its **Code** field to the **Country/Region Code** in the Donor table.

Property	Value
Indentation	2
NodeName	Country_Table
NodeType	Element
SourceType	Table
SourceTable	Country/Region
VariableName	<Country/Region>
SourceTableView	<Undefined>
ReqFilterHeading	<>
ReqFilterHeadingML	<>
CalcFields	<Undefined>
ReqFilterFields	<Undefined>
LinkTable	Donor
LinkTableForceInsert	<Yes>
LinkFields	Code=FIELD(Country/Region Code)
Temporary	<No>
Width	<0>
MinOccurs	<Once>
MaxOccurs	<Unbounded>

The actual Country Name is then referenced from the Country/Region table. Its indentation level is one more than the parent element so that it is nested within the parent (see the following screenshot).

Property	Value
Indentation	3
NodeName	Country_Name
NodeType	Element
SourceType	Field
SourceField	<Country/Region>::Name
FieldValidate	<Undefined>
AutoCalcField	<Yes>
Width	<0>
MinOccurs	<Once>
MaxOccurs	<Once>

The last data line in the sample (**Sales**, referencing a FlowField) has the **AutoCalcField** property set to **Yes** (the default) so that the FlowField will be calculated before it is output (see the following screenshot):



The XML document output from this sample XMLport is shown in the following screenshot. Note that the XML document is headed with a line showing the XML version and encoding type. That is followed by the heading element, and then the selected data.

```
<?xml version="1.0" encoding="UTF-16" standalone="no" ?>
- <C_AN_Donor_List>
- <Donor>
 <ID>1001</ID>
 <Name>Juan O'Hara</Name>
 <Address>1123 Riveria Way</Address>
 <Address2>Suite B-2</Address2>
 <City>Duluth</City>
 <StateProvince>MN</StateProvince>
 <PostalZone>55701</PostalZone>
 <CountryRegionCode>US</CountryRegionCode>
- <Country_Table>
 <Country_Name>USA</Country_Name>
</Country_Table>
 <Est_Gift_Value>298</Est_Gift_Value>
</Donor>
- <Donor>
 <ID>1002</ID>
 <Name>Karen Johnson Fashions, LTD</Name>
 <Address>19W243 Rue De La Paz</Address>
 <Address2 />
 <City>Nice</City>
 <StateProvince />
 <PostalZone>06000</PostalZone>
 <CountryRegionCode>FR</CountryRegionCode>
- <Country_Table>
 <Country_Name>France</Country_Name>
</Country_Table>
 <Est_Gift_Value>290</Est_Gift_Value>
</Donor>
</C_AN_Donor_List>
```

XMLports cannot run directly from a Navigation Pane action command, but can be run from actions on a page. You can choose to either run the XMLport by means of a properly constructed codeunit or directly from the page action. You need to keep in mind that RTC processes run on the NAV Service Tier. Any local storage device references will refer to the disks on the NAV Server.

When run from another object, XMLports are run from C/AL code that calls the XMLport and streams data either to or from an XML document file. This code is typically written in a Codeunit but can be placed in any object that can contain C/AL code.

The example process following is the minimum amount of code required to execute an exporting XMLport. The code is as follows:

```
XMLfile.CREATE('C:\DonorExport.xml');
XMLfile.CREATEOUTSTREAM(OutStreamObj);
XMLPORT.EXPORT(50000,OutStreamObj);
XMLfile.CLOSE;
ToFile := 'TargetFile.xml';
IF ISSERVICETIER THEN
 DOWNLOAD('C:\DonorExport.xml','Downloading file...','C:', '', ToFile);
```

Three variables are required to support the preceding code. In this example, the variables are defined as Globals, as shown in the following screenshot:

Name	DataType	Subtype
XMLfile	File	
OutStreamObj	OutStream	
ToFile	Variant	

The code purpose is explained, line by line, as follows:

- Line 1: Creates the data file to contain the XML document. The variable **OutStreamObj** is defined as **OutStream** data type.
- Line 2: Creates an **OutputStream**.
- Line 3: Executes the specific **XMLport**.
- Line 4: Closes the text data file.
- Line 5: Initializes the target file name on the client workstation.
- Line 6: Checks to see if this logic is running on a NAV Service Tier; if so, then uses the **DOWNLOAD** function to move the exported XML data to the client computer system.



An equivalent Codeunit designed to execute an importing XMLport would look very similar with the following differences:

- The **InStreamObj** variable would be defined as the **InStream** data type
- The **ISSERVICETIER** function would be at the start and would **UPLOAD** the file to be imported

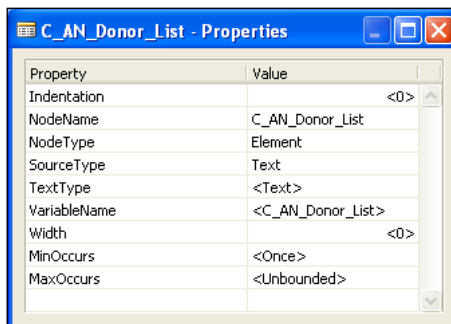
## XMLport line properties

The XMLport line properties which are active on a line depend on the contents of **SourceType** property. The first four properties listed are common to all three **SourceType** values (**Text**, **Table**, or **Field**) and the other properties specific to each are listed below the screenshots showing all the properties for each **SourceType**.

- **Indentation**: This indicates at what subordinate level in the hierarchy of the XMLport this entry exists. Indentation **0** is the primary level, parent to all higher numbered levels. Indentation **1** is a child of indentation **0**, indentation **2** is a child of **1**, and so forth. Only one Indentation **0** is allowed in an XMLport (that is, only one primary table).
- **NodeName**: This defines the Node Name that will be used in the XML document to open and close the data associated with this level. If the **NodeName** is **Customer**, then the starting and ending node names will be `<Customer>` and `</Customer>`. No spaces are allowed in a **NodeName**; you can use underscores, dashes, and periods but not other special characters.
- **NodeType**: This defines if this data item is an **Element** or an **Attribute**.
- **SourceType**: This defines the type of data this field corresponds to in the NAV database. The choices are **Text**, **Table**, and **Field**. Text means that the value in the **SourceField** property will act as a Global variable and, typically must be dealt with by embedded C/AL code. Table means that the value in the **SourceField** property will refer to an NAV table. Field means that the value in the **SourceField** property will refer to an NAV field within a parent table previously defined as an element.

## SourceType as Text

The following screenshot shows the properties for **SourceType** as **Text**:



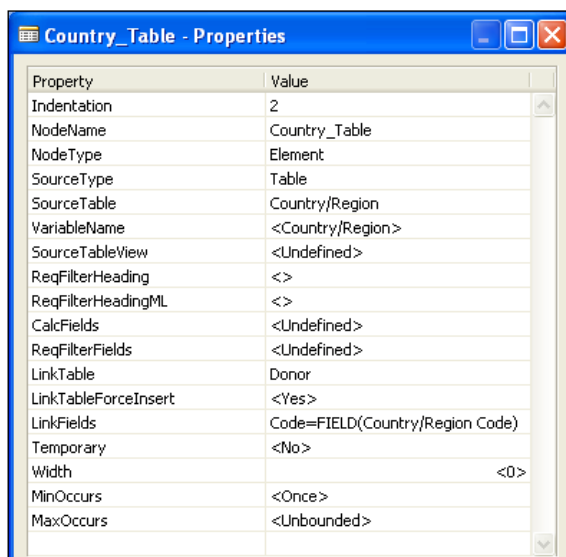
The description of the Text-specific properties is as follows:

- **TextType**: This defines the NAV Data Type as **Text** or **BigText**. Text is the default
- **VariableName**: This contains the name of the Global variable, which can be referenced by C/AL code

The **Width**, **MinOccurs**, and **MaxOccurs** properties are discussed later in this chapter.

## SourceType as Table

The following screenshot shows the properties for **SourceType** as **Table**:



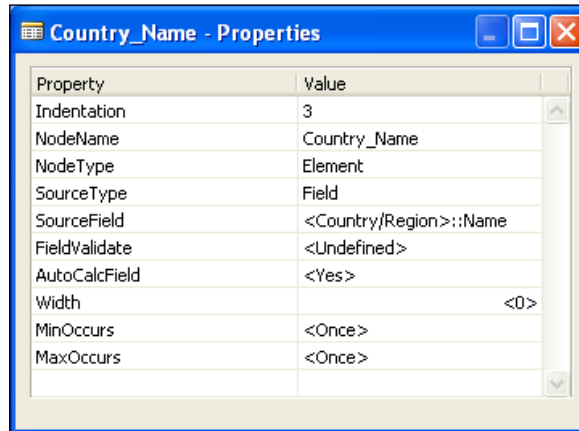
The description of the table-specific properties is as follows:

- **SourceTable:** This defines the NAV table being referenced.
- **VariableName:** This defines the name to be used in C/AL code for the NAV table. Essentially, this is a definition of a Global variable.
- **SourceTableView:** This enables the developer to define a view by choosing a key and sort order or by applying filters on the table.
- **ReqFilterHeading** and **ReqFilterHeadingML:** These fields allow the definition of the name of the Request Page filter definition tab that applies to this table.
- **CalcFields:** This lists the FlowFields in the table that are to be automatically calculated.
- **ReqFilterFields:** This lists the fields that will initially display on the Request page filter definition tab.
- **LinkTable:** This allows the linking of a field in a higher-level item to a key field in a lower-level item. If, for example, you were exporting all of the Purchase Orders for a Vendor, you might Link the **Buy-From Vendor No.** in a Purchase Header to the **No.** in a Vendor record. The **LinkTable** in this case would be Vendor and **LinkField** would be No.; therefore **LinkTable** and **LinkFields** work together. Use of the **LinkTable** and **LinkFields** operates the same as applying a filter on the higher-level table data so that only records relating to the defined lower-level table and field are processed. See the online **C/SIDE Reference Guide** Help for more detail.
- **LinkTableForceInsert:** This can be set to force insertion of the linked table data and execution of the related `OnAfterInitRecord()` trigger. This property is tied to the **LinkTable** and **LinkFields** properties. It also applies to **Import**.
- **LinkFields:** This defines the fields involved in a table + field linkage.
- **Temporary:** This defaults to **No.** If this property is set to **Yes**, it allows the creation of a Temporary table in working storage. Data imported into this table can then be evaluated, edited, and manipulated before being written out to the database. This Temporary table has the same capabilities and limitations as a Temporary table defined as a Global variable.

The **Width**, **MinOccurs**, and **MaxOccurs** properties are discussed later in this chapter.

## SourceType as Field

The following screenshot shows the properties for **SourceType** as **Field**:



The description of the Field-specific properties is as follows:

- **SourceField:** This defines the data field being referenced. It may be a field in any defined table.
- **FieldValidate:** This applies to **Import** only. If this property is **Yes**, then whenever the field is imported into the database, the `OnValidate()` trigger of the field will be executed.
- **AutoCalcField:** This applies to **Export** and FlowField Data fields only. If this property is set to **Yes**, the field will be calculated before it is retrieved from the database. Otherwise, a FlowField would export as an empty field.

The details of the **Width**, **MinOccurs**, and **MaxOccurs** properties follow in the next section.

## Element or attribute

An **Element** data item may appear many times but an **Attribute** data item may only appear (at most) once; the occurrence control properties differ based on the **TagType**.

## NodeType as an Element

The Element-specific properties are as follows:

- **Width:** When the XMLport **Format** property is **Fixed Text**, then this field is used to define the fixed width of this element's field.
- **MinOccurs:** This defines the minimum number of times this data item can occur in the XML document. This property can be **Zero** or **Once** (the default).
- **MaxOccurs:** This defines the maximum number of times this data item can occur in the XML document. This property can be **Once** or **Unbounded**. Unbounded (the default) means any number of times.

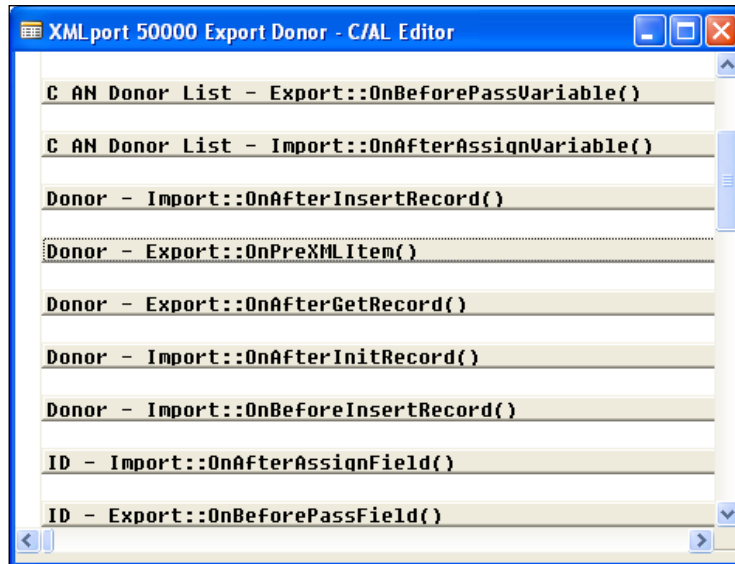
## TagType as an Attribute

The Attribute-specific property is as follows:

- **Occurrence:** This is either **Required** (the default) or **Optional**, depending on the text file being imported

## XMLport line triggers

The XMLport line triggers are shown in the following screenshot:



The triggers appearing for the XMLport data line depend on the values of the **DataType** field. As you can see in the preceding screenshot, there are different triggers depending whether **DataType** is **Text**, **Table** or **Field**.

---

## DataType as Text

The triggers for DataType as Text are:

- `Export::onBeforePassVariable()`, for **Export** only. This trigger is typically used for manipulation of the text variable.
- `Import::OnAfterAssignVariable()`, for **Import** only. This trigger gives you access to the imported value in text format.

## DataType as Table

The triggers for DataType as Table are as follows:

- `Import::OnAfterInsertRecord()`, for **Import** only: This trigger is typically used when the data is being imported into Temporary tables. This is where you would put the C/AL code to build and insert records for the permanent database table(s).
- `Export::OnPreXMLItem()`, for **Export** only: This trigger is typically used for setting filters and initializing before finding and processing the first database record.
- `Export::OnAfterGetRecord()`, for **Export** only: This trigger allows access to the data after the record is retrieved from the NAV database. This trigger is typically used to allow manipulation of table fields being exported and calculated depending on record contents.
- `Import::OnAfterInitRecord()`, for **Import** only: This trigger is typically used to check whether or not a record should be processed further or to manipulate the data.
- `Import::OnBeforeInsertRecord()`, for **Import** only: This is another place where you can manipulate data before it is inserted into the target table. This trigger is executed after the `OnAfterInitRecord()` trigger.

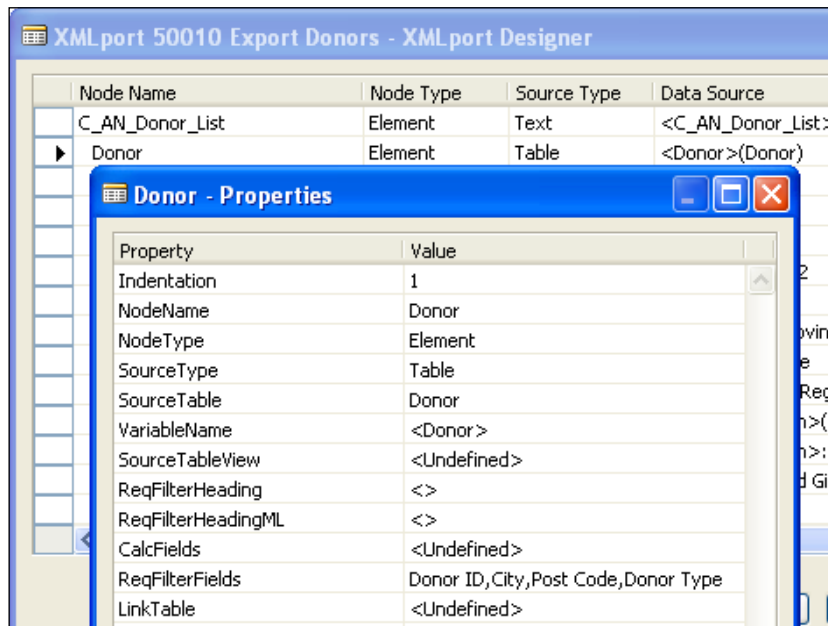
## DataType as Field

The triggers for DataType as Field are as follows:

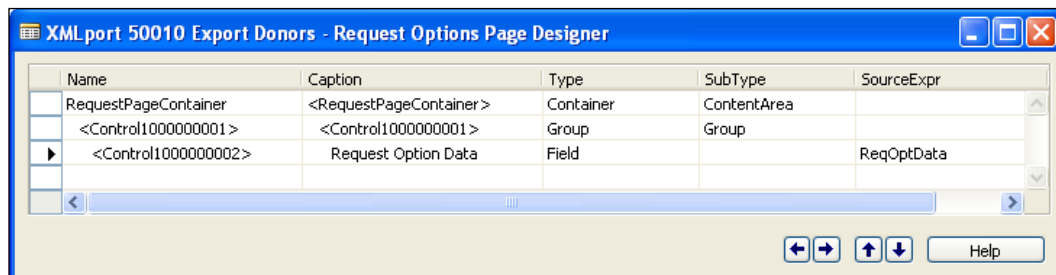
- `Import::OnAfterAssignField()`, for **Import** only. This trigger provides access to the imported data value for evaluation or manipulation before outputting to the database.
- `Export::OnBeforePassField()`, for **Export** only. This trigger provides access to the data field value just before the data is exported.

## XMLport Request Page

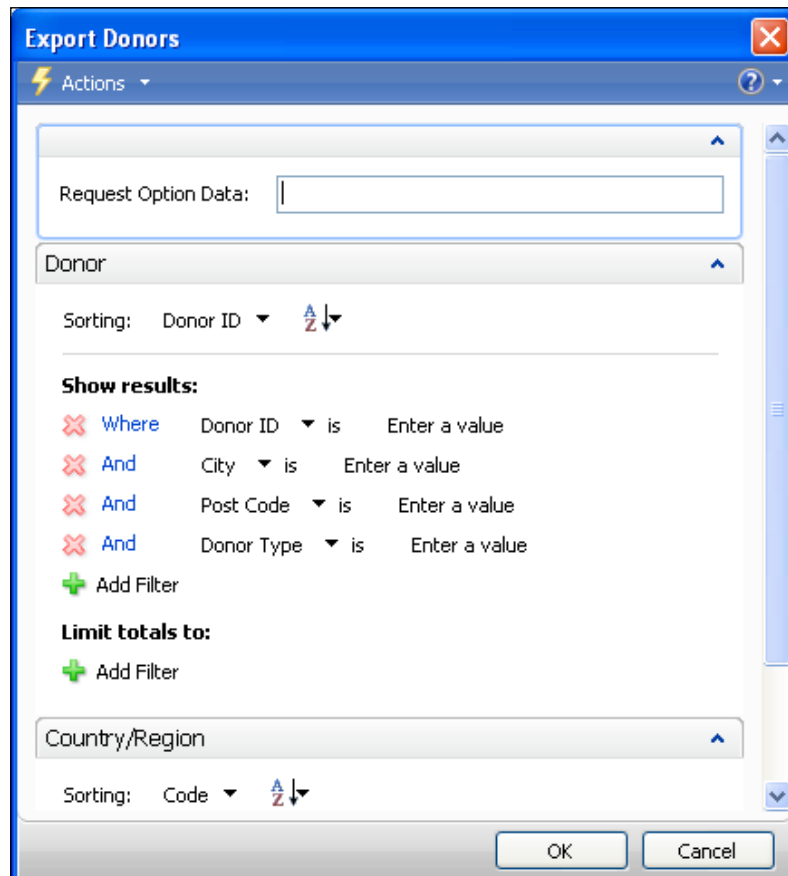
XMLports can also have a Request Page to allow the user to enter Option control information and filter the data being processed. Default filter fields that will appear on the Request Page are defined in the Properties form for the table XMLport Line (see the following screenshot for the Donor table line properties).



Any desired options that are to be available to the user as part of the Request Page must be defined in the **Request Options Page Designer**. This Designer is accessed from the XMLport Designer through **View | Request Page**. The definition of the contents and layout of the Request Options Page is done essentially the same way as other pages are done. To support the control definition in the following screenshot, a Global Variable was defined with a **DataType** of Text, named ReqOptData (in this case, a generic example field).



From the XMLport Request Page definitions shown in the two preceding screenshots, the actual Request Page is displayed as shown in the following screenshot. As with any other filter setup screen, the user has complete control of what fields are used for filtering and what filters are applied.



## Advanced interface tools

NAV has a number of other methods of interfacing with the world outside its database. We will review those very briefly here. To learn more about these, you should begin by reviewing the applicable material in the online **C/SIDE Reference Guide** Help material plus any documentation available with the software distribution. You should also study sample code, especially that in the standard system as represented by the Cronus Demonstration Database. And, of course, you should take advantage of any other resources available including the NAV-oriented Internet forums.



## Automation Controller

One option for NAV interfacing is by connection to COM Automation servers. A key group of Automation servers are the Microsoft Office products. Automation components can be instantiated, accessed, and manipulated from within NAV objects using C/AL code. Data can be transferred back and forth between the NAV database and COM Automation components.

Limitations include the fact that only non-visual controls are supported via this interface. You cannot use an Automation Controller defined COM component as a control on an NAV Page object. In NAV 2009, the Client Add-in feature, discussed later in this chapter, provides this capability through another integration interface (a significant enhancement). An Automation Controller defined COM component can have its own window providing an interactive graphical user interface.

Some common uses of Automation Controller interfaces are to:

- Populate Word template documents to create more attractive communications with customers, vendors, and prospects (for example, past due notices, purchase orders, promotional letters)
- Move data to Excel spreadsheets for manipulation (for example, last year's sales data to create this year's projections)
- Move data to and from Excel spreadsheets for manipulation (for example, last year's financial results out and next year's budgets back in)
- Use Excel's graphing capabilities to enhance management reports
- Access to and use of ActiveX Data Objects (ADO) Library objects to support access to and from external databases and their associated systems

## NAV Communication Component

The NAV Communication Component is an automation server that provides a consistent **Application Programming Interface (API)** communications bus adapter. Adapters for use with Microsoft Message Queue, Socket, and Named Pipe transport mechanisms are supplied with NAV. Adapters for other transport mechanisms can be added.

The NAV Communication Component enables input/output data-streaming communication with external services. Function calls can be done on either a synchronous or an asynchronous basis. The details on NAV Communication Component are available in the **Help** file, **devguide.chm**.

## Linked Server Data Sources

The two table properties, **LinkedObject** and **LinkedInTransaction**, are available for the SQL Server NAV database. Use of these properties in the prescribed fashion allows data access, including views, in linked server data sources such as Excel, Access, another instance of SQL Server, and even an Oracle database. For additional information, see the online **C/SIDE Reference Guide** Help section on Linked Objects.

## C/OCX

NAV interfaces with properly installed and registered Custom Controls (that is, .ocx routines) through the C/OCX interface granule. The interface and limitations are similar to those available for Automation Server Controls. Using C/OCX is one way, generally a relatively economical way, to interface between NAV and various other software products or hardware products. An excellent example would be to use Microsoft Common Dialog Control to invoke the standard File Open/Save dialog for user convenience.

## C/FRONT

C/FRONT is a programming interface that supports connection between C language-compatible software and the NAV database. C/FRONT provides a library of functions callable from C programs. These functions provide access to all aspects of the NAV database with the goal of providing a tool to integrate external applications with NAV. There is a separate manual called *C/FRONT Reference Guide*.

C/FRONT has continuously been enhanced to provide its API access to languages other than C and C++. Earlier, C/FRONT was made usable by additional languages such as Visual Basic and C#. Now, it can be used by any .NET language.

The addition of the Web Services interface capability will take over many of the uses of C/FRONT for integration.

## NAV Application Server (NAS)

The NAV Application Server is a middle-tier server that runs as a service. It is essentially an automated user client. Because NAS was created primarily using components from the standard NAV C/SIDE client module, NAS can access all of NAV's business rules.



Whenever feasible, in NAV 2009, the Web Services functionality should be used rather than NAS.



NAS is a very powerful tool, but it can run only NAV Report and Codeunit objects and only those that do not invoke a graphical user interface of any type. Any error messages that are generated by an NAS process are logged in the Event Viewer or written to a file defined when NAS is started from a command line.

NAS operates essentially the same as any other NAV C/SIDE client (except for being automated). It processes all requests in its queue one at a time, in the same manner as the GUI client. Therefore, as a developer, you need to limit the number of concurrent calls to an NAS instance as the queue should remain short to allow timely communications between interfaces. If necessary, additional NAV Application Servers can be added to the system configuration (with appropriate license purchases).

## Client Add-ins

One of the major new capabilities released in NAV 2009 Service Pack 1 is the Client Add-in feature. In brief, Client Add-in allows the developer to extend the Role Tailored Client through the integration of foreign (that is, non-NAV) controls. Previously, any extension to NAV that could be done was more of a "paste-on" (for example, COM and OCX), rather than fully integrated as are Client Add-ins.

## Client Add-in definition

A Client Add-in is a Microsoft .NET assembly (a collection of functionality built and deployed as a unit) that allows us to add custom functionality to the Role Tailored Client. The Client Add-in (aka Client Extensibility) feature is an API that supports the integration of add-ins which we can construct. Contrary to the limitations on other integration options, Client Add-ins can be graphical and appear on the RTC display as part of or mingled with the native NAV controls.

Some simple examples of how Client Add-ins might be used to extend RTC UI behavior:

- A standard appearing NAV text control that offers a special behavior when you double click it.
  - Increase the font size on a display
  - Change the active sort key on a lookup list display between Code and Description
  - Simply expand to show extended text
- A dashboard made up of several dials showing the percentage of some resources relative to target limits or goals. These dials could support click and drill into the underlying NAV detail data.

- An integrated sales call mapping function which integrates the process of choosing customers in an area on which to call, displays those customer locations on a map and creates a sequenced call list with pertinent sales data from the NAV database.
- Interactive visualization of a workflow or flow of goods in a process, perhaps showing the number of entries at each state in the process, then allowing filtering to entries related only to the current User.

All these are hypothetical examples and may be tasks better accomplished in some other way. The point here is that when it comes to using the brand new and very flexible NAV feature, let your imagination go (with appropriate budgetary limits, of course).

## Client Add-in construction

In many instances, Client Add-ins will be created and packaged by Partners specializing in their creation. But, on occasion, you may want to create a simple, special purpose add-in for a particular customer enhancement.

As you would expect when working with an API, there is a certain prescribed structure that you must use to create a Client Add-in interfacing with the RTC. However, so long as the code within the add-in is a well-behaved .NET code, there is a great deal of flexibility in the structure and purpose of the code and functionality within the add-in. The control can be one you create, a standard WinForms control or one that you've acquired from a third party.

Once you know what the control is that you're going to use, you need to build the add-in structure which envelopes the control. The most logical toolsets for building add-ins are any version of Visual Studio 2005 or 2008, or one of the free downloadable tools such as Visual Studio Express for C#.

Let's look at an overview of the construction process for an add-in.

1. Define a class library containing all the classes which must be referenced by the control.
2. Declare a new class in the assembly namespace of your choice for the add-in control. This may be derived from a base class like `StringControlAddInBase`. Alternatively a combination of interfaces can be used (for example, `IStringControlAddInDefinition` and `IEventControlAddInDefinition`).

3. Implement the `CreateControl` method and define (or include) the functionality. Using this method you can subscribe to events of the control and handle them. Based on the business scenarios that you implement, you can raise the `ControlAddIn` trigger in C/AL code by calling the `RaiseControlAddInEvent` method in the `StringControlAddInBase` or by executing a corresponding event in `IEventControlAddInDefinition`. You would typically pass a message ID of your choice along with the data you want to send with the event (of type `string` for the `StringControlAddInBase`).
4. Sign the assembly using the strong name key (third-party controls may have to undergo a separate signing step).
5. Build the solution and copy the general add-in assembly to each client workstation to the proper location.
6. Register the add-in within NAV in the Client Add-in table (Table 2000000069).
7. Add the add-in control to a field control on a page using the **`ControlAddIn`** property. Bind the control through the `SourceExpr` property as you are used to doing with other fields on pages.
8. If the Add-in provides information through eventing (that is, communicating an event), implement the data handling code to the **`OnControlAddIn`** trigger.

Now let's walk through each of the above steps in abbreviated form. We're not going to actually build a Client Add-in as good examples of that are available in the online **C/SIDE Reference Guide** Help and in one or more of the NAV team blogs on the Internet. Note that there may be variations on each of the following illustrations – these are generic, not specific code to be copied as is.

1. Define a class library:

```
using System;
using System.ComponentModel;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.Dynamics.Framework.UI.Extensibility;
using Microsoft.Dynamics.Framework.UI.Extensibility.WinForms;
```

2. Declare a new class for the add-in:

```
Namespace MyAddinsAssembly
[ControlAddInExport ("MyCompany.MyProduct.NewAddin")]
[Description("This is my new Client Add-in")]
public class MyNewAddin : StringControlAddInBase
```

The values `MyCompany.MyProduct.NewAddin`, the quoted description and `MyNewAddin` are developer defined names. .NET naming conventions should be followed as good practice. The `MyCompany.MyProduct.NewAddin` name will be used to Register the control in NAV. Instead of the convenience base class `StringControlAddInBase`, you can use the base class `WinFormsControlAddInBase` together with a combination of interfaces for data binding and eventing, based on your requirements. Supported right now are the `IStringControlAddInDefinition`, `IObjectControlAddInDefinition`, and `IEventControlAddInDefinition`.

3. Define the code to implement the instantiation of the new control.
4. Sign the add-in assembly by using the development environment tools for that purpose. This also needs to be done for third party controls which have not previously been signed as this process assigns a unique public key by which NAV can identify and access the add-in within its resident assembly.
5. Build the solution and copy the assembly to the proper client workstation location. The default directory is `C:\Program Files\Microsoft Dynamics NAV\60\RoleTailored Client\Add-ins`. The add-in assembly will consist of one or more Microsoft .NET Framework-based .dll files (more than one control add-in can be contained in a single .dll file). The assembly file(s) can be put in the add-in directory or a sub-directory (for more structured organization).
6. Obtain the public key, open NAV, and enter the registration data into the Client Add-in table (Table 2000000069) using the name that was assigned as part of the class definition. There is a Registration Tool available in a NAV blog to facilitate the key identification and registration process at <http://blogs.msdn.com/cabeln> (this same blog has much additional high quality information about the Client Add-in feature). Or you can run Table 2000000069 and manually enter at least the two required fields, the Name of the add-in and the assigned Public Key Token. The other two fields in the Client Add-in table, Version and Description, are optional.
7. Set up the appropriate field(s) in a NAV page to properly access the capabilities of your new add-in when the page is invoked during application processing.
8. Implement necessary data-handling code to the **OnControlAddIn** trigger. Typically there will be some C/AL code required for data communication between the NAV control and the add-in.
9. Test the newly integrated Client Add-in.

## **Client Add-in comments**

Obviously, in order to take advantage of the Client Add-in capabilities, you will need to develop some minimal Visual Studio development skills and probably some C# programming skills.

Care will have to be taken when designing add-ins that their User Interface style complements, rather than clashing with, the UI standards of the NAV RTC.

Client Add-ins are a major extension of the flexibility and capability of the Dynamics NAV system. This feature will allow ISVs to create and sell libraries of new controls and new control based micro-applications. It will allow vertically focused Partners to create versions of NAV that are much more tailored to their specific industries. This feature will allow the integration of third party products, software, and hardware, at an entirely new level.

In short, the Client Add-in feature is terrific. Learn it, use it, and both you and your users will benefit.

## **Web services**

Web services are an industry standard API defined by the W3C (World Wide Web Consortium), the group who develop and promote Web protocols and guidelines. Web services allow different software applications to interoperate using standard interface specifications along with standard communications services and protocols. In plain English, when NAV publishes some web services, those functions can be accessed and utilized by properly programmed software residing anywhere on the Web. The software does not need to be directly compatible with C/SIDE or even .NET, it just needs to obey web services conventions and have security access to the NAV Web Services.

There are many different types of web services and many different applications. Some of the primary benefits of using web services are:

- It's a widely accepted and widely used standard
- It's flexible, for example, web services communicates data via XML
- It's as reliable and simple
- The tools used on each end of the communications just need to talk to one another, they don't need to be members of the same "family"
- It allows access over a network, including the Internet

Some specific benefits of NAV Web Services are:

- Very simple to publish (that is, to expose a web service to a consuming program outside of NAV)
- Provides easy access to NAV data while respecting and enforcing NAV rules and logic
- Provides access to NAV business logic
- Uses code and design that already exists
- Uses Windows Authentication and respects NAV data constraints
- Supports SSL

Examples of the types of situations where NAV Web Services might be used include:

- Any of the wide variety of things that field sales people might want to do
  - Check customer order status
  - Enter orders
  - Check inventory availability
  - Check sales commissions
- Any of the wide variety of things that field service people might want to do
  - Enter service activity report
  - Enter parts replaced
  - Enter job time information
  - Trigger service call billing
  - Request next call assignment
- Time and expense entry
- Physical inventory of goods or assets from mobile devices
- Any remote access, limited functionality application

There are several factors that should be considered in judging the appropriateness of an application being considered for web services integration. Some of these are:

- The degree to which the functionality of the standard RTC interface is needed. Web services application should not try to replicate what would normally be done with a full client, but should be used for limited, focused functionality.



- The amount of data to be exchanged. Generally, web services are used remotely. Even if it is used locally, there are additional levels of security handshaking and inter-system communications required. Web services should be used for low data volume applications.
- How public is the user set. For security reasons, the user set for direct connection to your NAV system should generally be limited to known users, not the general public. Web services should not be used to provide Internet exposure to your NAV system, but rather for **intranet** access.

As web services are intended for use for low-intensity users, there are separate license options available with lower costs per user than the full client. This can make the cost of providing web services-based access to customers quite reasonable, especially if by doing so, you increase the level of customer service and (of course) profits.

## Exposing a web service

Two types of NAV objects can be published as Web services, Pages, and Codeunits. The essential purposes are:

- Pages provide access to the associated primary table. Use Card Pages for table access unless you have a specific reason for using another page type.
- Codeunits provide access to the functions contained within each Codeunit.

[  Terminology alert: **Application function = object method = NAV function.** ]

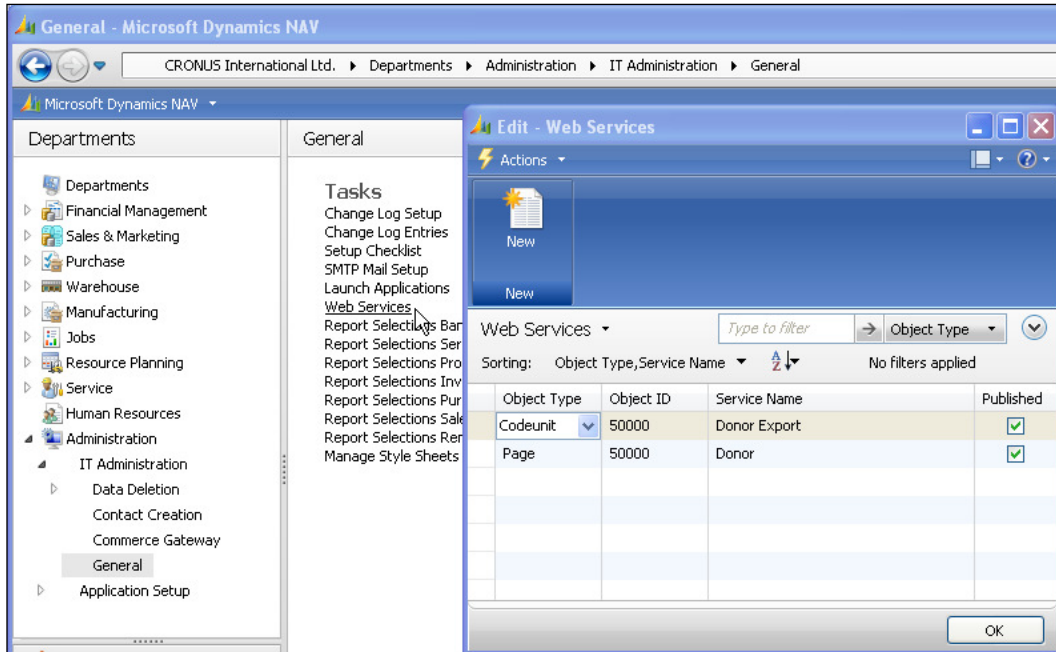
When a page has no special constraints, either via properties or permissions, there will normally be eight methods available. They are:

1. **Create**: Create a single record (similar to a NAV INSERT).
2. **CreateMultiple**: Create a set of records (passed argument must be an array).
3. **Read**: Read a single record (similar to a NAV GET).
4. **ReadMultiple**: Read a filtered set of records, paged. Page size is a parameter.
5. **Update**: Update a single record (similar to a NAV MODIFY).
6. **UpdateMultiple**: Update a set of records (passed argument must be an array).
7. **Delete**: Delete a single record.
8. **IsUpdated**: Check if the record has been updated since it was read.

Publishing a web service is one of the easiest things you will ever do in NAV. There are two options, one within the Classic Client, the other from within the Role Tailored Client. From within the Classic Client, you access the web services form on the Administration Menu of the Navigation Pane through **Navigation Pane | Administration | IT Administration | General Setup | Web Services**. The web services form will display as shown in the following screenshot.

[illegible]

From within the RTC, the process is essentially the same. The point of access is the Departments menu through **Navigation Pane | Departments | Administration | IT Administration | General | Web Services**. The web services page displays as shown in the following screenshot. The contents and usage of each of the data fields are the same as those we just reviewed for the Classic Client.



## Determining what was published

Once an object has been published, you may want to see exactly what is available as a web service. As these are web services, intended to be accessed from the Web, that's exactly where we'll go to see what other software will see.

In the address bar of our browser, we will enter the following (all as one string):

```
http://<Server>:<WebServicePort>/<ServerInstance>/WS/
<CompanyName>/services
```

The various elements enclosed in < > in this address will be replaced with the values, which can be found in the CustomSettings.config file. In a default installation, this file is located in the directory C:\Program Files\Microsoft Dynamics NAV\60\Service

Example URL addresses are:

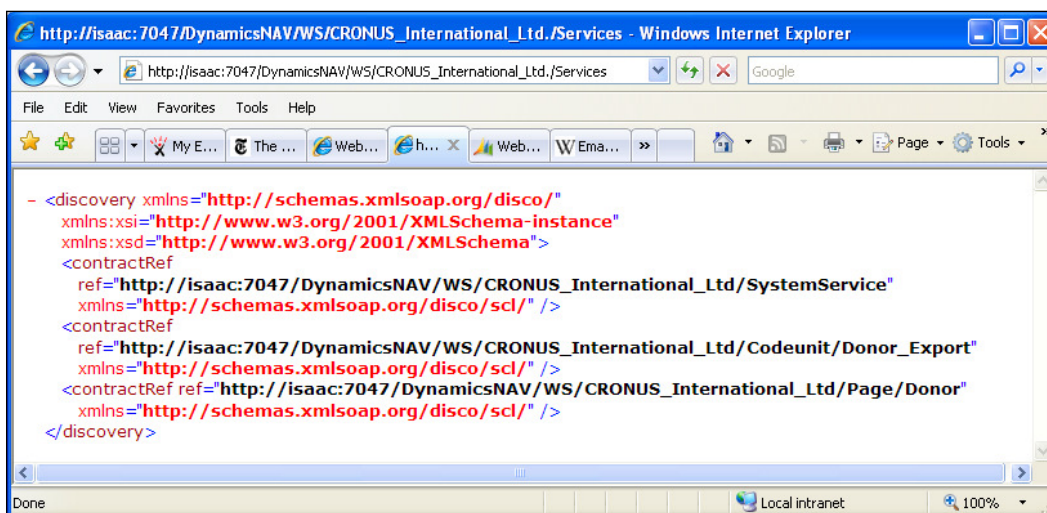
`http://localhost:7047/DynamicsNAV/WS/Services`

`http://Isaac:7047/DynamicsNAV/WS/CRONUS_International_Ltd/Services`



The company name is optional and case-sensitive. The terminating period in the company name is also optional.

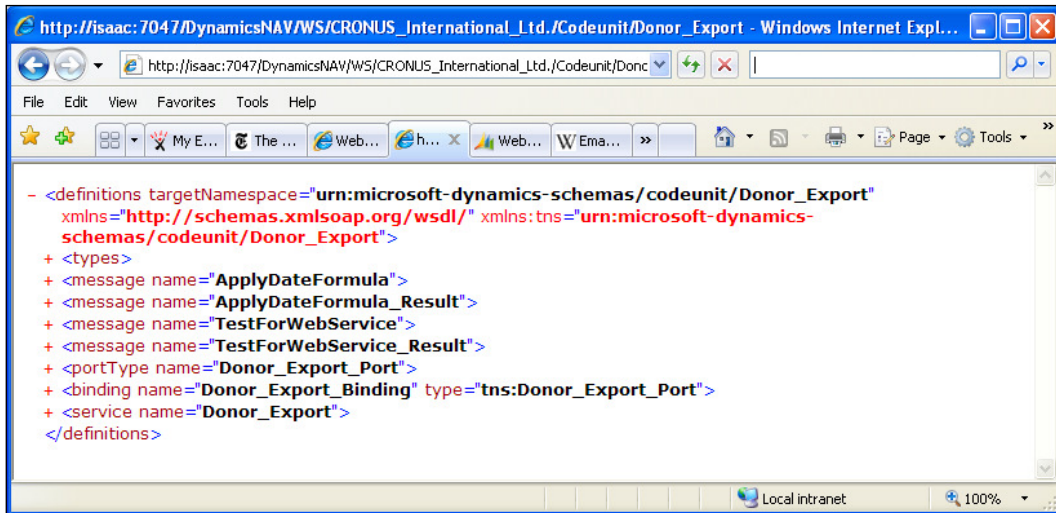
When the correct address string is entered, your browser will display a screen similar to the following image. This image is in an XML format of a data structure called **WSDL, Web Services Description Language**:



In this case, we can see that we have two Services available: Codeunit/Donor\_Export and Page/Donor.

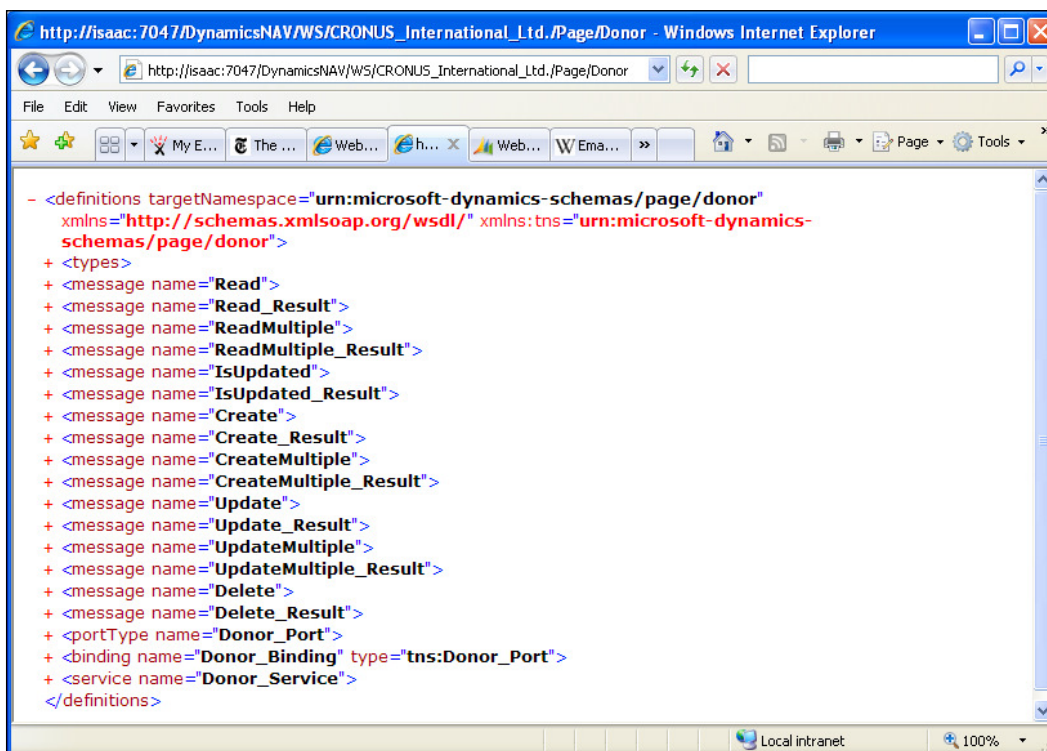
To see the methods (that is, functions) that have been exposed as web services by publishing these two objects, we can enter other similar URLs in our browser address bar. To see the web services exposed by our codeunit, we change the URL used earlier to replace the word *Services* with *Codeunit/Donor\_Export*. If we hadn't included the optional company name earlier, we do have to include that in this URL. The revised URL gives us the information shown in the following screenshot. When this display comes up initially, all the XML sections are expanded.

The following display has all of the sections collapsed to make the information easier to read:



Our Donor Export has two functions in it, `ApplyDateFormula` and `TestForWebService`. Both of those are visible here as methods. Each method also has a `Result`, which contains whatever information the method returns back to the calling routine.

To see the web services exposed by our page, we put the string `Page/Donor` following the company name in the URL. The revised URL gives us the following information onscreen. Once again, the XML sections have been collapsed for easier reading.



In this screenshot, you can see the eight data access methods that we reviewed earlier along with the associated Results.

The actual consumption (meaning "use of") of a web service is also fairly simple, but that process occurs in another application developed with another, non-NAV 2009, toolset. As our focus here is strictly the NAV 2009 side of the equation, we're not going to dig into the technical aspects of the web service consumption process. There are a number of relatively detailed examples in the Internet blogs about NAV and in the **C/SIDE Reference Guide** Help about routines that consume NAV 2009 web services.

Tools that can be used to consume NAV Web Services include, among many others, Microsoft InfoPath, Microsoft Excel, Microsoft Sharepoint, applications written in C#, other .NET languages, open source PHP, and a myriad of other application development tools. Remember, web services are a standard interface for dissimilar systems.

While it is technically feasible to publish any of the pages and codeunits that make up NAV 2009 (some marketing literature mentions something akin to that), many of those object would not be particularly useful published as web services. A variety of reasons exist for that including duplications and access constraints (such as not editable, and so on), which narrow a functionality to be useful and others.

As with any other enhancement to the system functionality, serious thought needs to be given to the design of what data is to be exchanged and what functions within NAV 2009 should be invoked for the application to work properly. In many cases, you will want to provide some simple routines to perform standard NAV processing or validation tasks without exposing the full complexity of NAV internals as web services.

Perhaps you will want to provide just two or three functions from a Codeunit that contains 30 or 50 functions. Or you would want to expose a function that is contained within a Report object. In each of these instances and others, it will be necessary to create a basic library of C/AL functions that can be published as web services.

Use of web services carries with it some additional issues that must be dealt with in any production environment. In addition to the basic application functionality, you will have to plan for security, access, and communications issues. These are also outside the scope of this text, but nevertheless, critical to the implementation of a successful application solution.

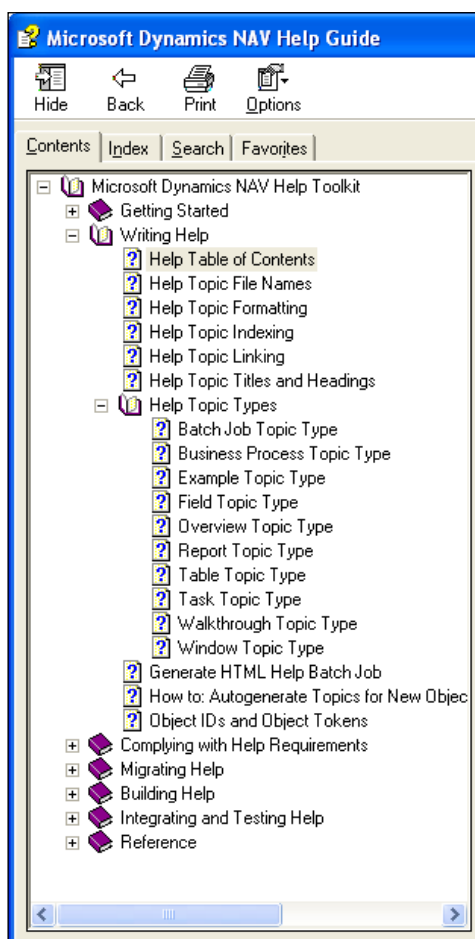
## Customizing Help

One of the several auxiliary toolkits that's available to help you deliver better solutions to clients is the Microsoft Dynamics NAV Help Toolkit. Once installed on your development system, this toolkit allows you to create customized Help for the application customizations that you deliver.

Part of the Help Toolkit is the *Help Toolkit Help Guide*. Once you have downloaded and unzipped the Help Toolkit, you will find it useful to immediately find the file `NAV_HelpGuide.chm` and double-click on it to see the Help contents. In this Help file are instructions on installing and using the Help Toolkit. Reading through the installation instructions, perhaps even printing them off for ready reference, will make installing the toolkit much easier.

A sample display of a portion of the Table of Contents of the Help Toolkit is shown in the image following. The **Getting Started** section contains the installation and setup instructions. You can see that the following sections include a great deal of information on the actual tasks involved in creating customized Help for your customized application.





Before you begin your first Help customization effort, you should use the information in this guide as a tutorial. A good place to start is the section entitled **Walkthrough: Creating and Building a New Help Project**.

Developing customized Help is similar in many ways to developing customizations to the application software. In most of the cases, the top-level style of new material should be essentially similar to the original product material. If the customizations are too different from the base material, it will create additional operational challenges for the users and thus training and support challenges for the Partner.



Help customizations should be designed to make it as easy as possible to port them to new versions of the product when upgrading occurs. Again, very similar to the software development, whenever possible, Help revisions should be done on a copy of the original Help topic, leaving the original unchanged. For example, if the Customer Card is modified, make the Help changes in a copy of the Customer Card Help.

By taking advantage of the Help Toolkit, you have additional abilities to make your extensions and customization of the NAV system more completely and invisibly integrated into the product. This makes it easier for your users and therefore easier for you.

## **NAV development projects**

When you start a new project, the goals and constraints for the project must be defined. The degree to which you meet these will be a significant part of the measure of your success. Some examples:

- What are the functional requirements and what flexibility exists within these?
- What are the user interface standards?
- What are the coding standards?
- What are the calendar and financial budgets?
- What existing capabilities within NAV will be used?

Now that you understand the basic workings of the NAV C/SIDE development environment and C/AL, we'll review the process of software design for NAV enhancements and modifications. Designing for NAV requires more forethought and knowledge of the operating details of the application than was needed with traditional models of ERP systems.

New material should be as compatible and as consistent with the original as possible. After you are done, it should be difficult (or impossible) for a new user to tell what's original and what's a modification. As we discussed previously, when the modifications don't look the same or don't work in the same fashion as the base product, it often increases training time, decreases user efficiency, and requires more support. Thus, even though you may feel you have a "better way", be very cautious about applying it.

## Knowledge is key

In order to respect the existing system being modified, you have to understand and appreciate the overall structure and design philosophy of that system. As we have seen, NAV has unique data structure tools (SIFT and FlowFields), quite a number of NAV-specific decimal and date functions, which make it easier to program business applications, and a data structure (journal, ledger, and so on), which is inherently an accounting structure. The learning curve to become expert in the way NAV works is significant, both because of the differences between NAV and other business software packages, and because there is very little product design documentation.

## Different approaches for different scopes

When you embark on a new project, you should consider whether you are creating software for a new functional area or creating a wholly integrated change of existing NAV functionality, that is, a modification. This is significant because different approaches are appropriate and feasible for different scopes of work. If you are adding an entry to the main Navigation Page or at the top level of one of the primary Action menus, then your work is likely a new functional area. If there is no action entry involved in your enhancement, then it is probably not a new functional area. We'll briefly discuss some of the issues that make working on these two task types different from one another.

## Advantages of designing new functionality

When creating a new functionality, you typically have more leeway in the design of your data structure, pages, reporting, processing flow, and user interface subtleties than you do when you are enhancing existing functionality. That doesn't eliminate the importance for consistency with the design of the original system.

Menu structure, page structure, invoking of reports, indeed, the whole user experience needs to be designed and implemented in a manner consistent with the out of the box product. When you don't maintain consistency, training requirements, error rates, and overall user frustration may all be increased. Significant user interface differences from one portion of the system to another often result in extra challenge and expense.

New functionality will likely have a representation of new instances of most types of NAV objects, almost certainly including one or more tables, reports, and pages along with a menu entry or two. You may have XMLports and other external interfaces as well. Some developers find developing new functionality an easier task than modifying existing functionality. There are several reasons for this.

First, you have more freedom in your user interface design. Second, much of what you are creating involves completely new objects. Creating a new object can be easier because you don't have to study, understand, and integrate with all the complexities of an existing object. Even documenting a new object is easier because you can put most, if not all, of your internal documentation comments in one place, rather than having to identify and comment individual modifications.

Third, adding new objects generally makes upgrading easier. A new object can often be moved to an upgraded version with little or no change. In fact, this aspect of system maintainability sometimes justifies utilizing new objects for modifications even when they are not otherwise required.

The possibility of easier upgrading should not be interpreted to mean that custom objects don't need to be carefully examined during an upgrade. There is always the possibility that the data structure or flow on which the modification design was built has been changed in the new version. In that case, the custom object will need to be revised just as would the embedded customization.

## **Modifying an existing functional area**

When you are modifying existing functionality, you should operate with a very light touch, changing as little as possible to accomplish your goals. Your new code should look like NAV code and your user interface will work like the standard product interface. Naturally you will leave behind a well documented trail. Your new fields will fit neatly and tightly on the pages. Your new feature will operate so similarly to existing standard NAV functions that little or no user training will be required except to inform users that the new feature exists and what it does.

In contrast to the pros and cons of creating a new functional area with your modification, there is a different set of pros and cons when your modification is localized and tightly integrated into existing NAV structure and code. Because your modification should closely resemble the design and structure of what already exists, you don't have nearly as much creative freedom.

Modifying an existing function means you may not need to learn as much about optional approaches, that is, you have fewer choices. In addition, while you should be intimately familiar with the processes and logic in the parts of the system you are affecting, you can usually focus your study on a relatively small segment of the original system, thus reducing the breadth of your study of the existing system. Finally, when you are modifying an existing function, you have the distinct advantage of knowing that the foundation code and objects on which you are building are already debugged and working. You may need little or no change to user procedures and documentation.

Testing can be easier in a smaller modification. Often you will have your choice of testing using the Cronus demo database or using a test copy of the production database. Depending on exactly what your modification does, you may be able to use existing setups, master table data, and perhaps even existing in-process transactions to test your modification. As the creation of a consistent set of test data can be a very time-consuming process, this is a significant advantage.

Advantages of using a copy of the live database for testing include having a fully set up system that matches the customer's production environment as well as data that matches the real production world. This can make it quite a bit easier to compare new results to previous results and to spot speed bottlenecks that relate to historical data volumes. This applies to any type or scope of modification.

## NAV development time planning

For many years, those responsible for software development technology have been promising to increase the ratio of design time and effort to the coding time and effort (that is, more time spent designing and less in coding). But very few systems or toolsets have fulfilled that promise, particularly in the business application marketplace. In a majority of non-NAV customization projects, much more time and effort is spent writing and debugging code than in the design of that code.

The NAV structure, code functions, and IDE tools actually begin to fulfill that long-delayed promise of significantly reducing the coding effort/design effort ratio. When working on a NAV modification, it is not unusual to spend several hours studying existing code, testing logic, and flow with a variety of data samples, before making a code change of just two, three, or four lines to effect a significant change.

This is due to the combination of the very tight code structure and the powerful set of language functions NAV provides. That's not to say that NAV is perfectly constructed and has no code could be improved. But if it were perfect, there wouldn't be much work left for us to do, would there?

## Data-focused design

Any new application design must begin with certain basic analysis and design tasks. That is just as applicable when our design is for new functionality to be integrated into an existing standard software package such as NAV.

What information must the new application functionality make available to the users in order to accomplish its goals? How do the users expect that data to be presented? What type of constraints exist due to practice or practicality? What actions are to be performed by the system when presented with the appropriate and expected data or user actions?

## **Determining the data needs**

First, we must determine what underlying data is required. What will it take to construct the material the users need to see? What level of detail and in what structural format must the data be stored so that it may be satisfactorily retrieved? Having defined the data and other inputs that are required, we must define the sources of all this material. Some may be input manually, some may be forwarded from other systems, some may be derived from historical accumulations of data, and some will be computed from combinations of all these, and more. In any case, every component of the information needed must have a clearly defined point of origin, schedule of arrival, and format.

## **Defining the needed data views**

Define how the data should be presented. How does it need to be "sliced and diced"? What levels of detail and summary? What sequences and segmentations? What visual formats? What media will be used? Will the users be local or remote? Of course, ultimately many other issues also need to be considered in the full design, including user interface specifications, data and access security, accounting standards and controls, and so on.

## **Designing the data tables**

Data table definition includes the data fields, the keys to control the sequence of data access and to ensure rapid processing, frequently used totals (which are likely to be set up as **SumIndex Fields**), references to lookup tables for allowed values, and relationships to other primary data tables. It is important not to just define the primary data tables (for example, those holding the data being processed in the course of business activity) but also to include any related lookup tables and controlling "setup" information tables. The design effort must also consider what "backward looking" references to these new tables will be added to the already existing portions of the system. These connections are often the finishing touch that makes the new functionality operate in a truly seamlessly integrated fashion with the original system.

## **Designing the user data access interface**

Design the pages and reports to be used to display or interrogate the data. Define what keys are to be used or available to the users. Define what fields will be allowed to be visible, what are the totaling fields, how the totaling will be accomplished (for example, FlowFields, on-the-fly), and what dynamic display options will be available. Define what type of filtering will be needed. Some filtering needs may be

beyond the ability of the built-in filtering function and may require auxiliary code functions. Determine whether external data analysis tools will be needed and will therefore need to be interfaced. Design considerations at this stage often result in returning to the previous data structure definition stage to add additional data fields, keys, SIFT fields, or references to other tables.

## Designing the data validation

Define exactly how the data must be validated before it is accepted upon entry into a table. There are likely to be multiple levels of validation. There will be a minimum level, which defines the minimum set of information required before a new record is accepted. The minimum may be no more than an identifying number or it may include several data fields. At the least, it must include all the fields that make up the Primary Key to the table.

Subsequent levels of validation may exist relating to particular subsets of data, which are, in turn, tied to specific optional uses of the table. For example, in the base NAV system, if the manufacturing functionality is not being used, the manufacturing-related fields in the Item Master table do not need to be filled in. But if they are filled in, then they must satisfy certain validation criteria.

As mentioned earlier, the sum total of all the validations that are applied to data when it is entered into a table may not be sufficient to completely validate the data. Depending on the use of the data, there may be additional validations performed during the processing, reporting, or inquiries.

## Data design review and revision

Perform the above three steps for the permanent data (Masters and Ledgers) and then for the transactions (Journals). As a general rule, once all the supporting tables and references have been defined for the permanent data tables, there are not likely to be many, if any, such new definitions required for the Journal tables. If any significant new supporting tables or new table relationships are identified during the design of Journal tables, you should go back and re-examine the earlier definitions. Why? Because there is a high likelihood that this new requirement should have been defined for the permanent data and was overlooked.

## **Designing the Posting processes**

First define the final data validations, then define and design all the ledger and auxiliary tables (for example, Registers, Posted Document tables, and so on). At this point, you are determining what the permanent content of the Posted data will be. If you identify any new supporting table or table reference requirements at this point, you should go back to the first step to make sure that this requirement didn't need to be included at that earlier design definition stage.

Whatever variations in data are permitted to be Posted must be assumed to be acceptable in the final instance of the data. Any information or relationships that are necessary in the final Posted data must be ensured before Posting is allowed to proceed.

Part of the Posting design is to determine whether data records will be accepted or rejected individually or in complete batches. If the latter happens, you must also define what constitutes a batch; if the former, it is quite likely that the makeup of a Posting Batch will be flexible.

## **Designing the supporting processes**

Design the processes necessary to validate, process, extract, and format data for the desired output. In earlier steps, these processes can be defined as "black boxes" with specified inputs and required outputs, but without overdue regard for the details of the internal processes. That allows you to work on the several preceding definition and design steps without being sidetracked into the inner workings of the output related processes.

These processes are the cogs and gears of the functional application. They are necessary, but often not pretty. By leaving design of these processes in the application design as late as possible, you increase the likelihood that you will be able to create common routines and to standardize how similar tasks are handled across a variety of parent processes. At this point, you may identify opportunities or requirements for improvement in material defined in a previous design step. In that case, you should return to that step relative to the newly identified issue. In turn, you should also review the effect of such changes for each subsequent step's area of focus.

## **Double-check everything**

Do one last review of all the defined reference, setup, and other control tables to make sure that the primary tables and all defined processes have all the information available when needed. This is a final design quality control step.

It is important to realize that returning to a previous step to address a previously unidentified issue is not a failure of the process, it is a success. An appropriate quote used in one form or another by construction people the world over is *Measure twice, cut once*. It is much cheaper and more efficient to find and fix design issues during the design phase rather than after the system is in testing or, worse yet, in production (it's quieter too).

## Design for efficiency

Whenever you are designing a new modification, you not only need to design to address the defined needs, but also to provide a solution that processes efficiently. An inefficient solution carries unnecessary ongoing costs. Many of the things that you can do to design an efficient solution are relatively simple.

## Disk I/O

The slowest thing in any computer system is the disk I/O. Disk I/O almost always takes more time than any other system processing activity. Therefore, when you begin concentrating your design efforts on efficiency. Focus first on minimizing the disk I/O.

The most critical elements are the design of the keys, the number of keys, the design of the SIFT fields, the number of SIFT fields, the design of the filters, and the frequency of accesses of data (especially FlowFields). If your system is going to have five or ten users, processing a couple of thousand order lines per day and is not heavily modified, you probably won't have much trouble. But if you are installing a system with one or more of the following attributes, any of which can have a significant effect on the amount of disk I/O, you will need to be very careful with your design and implementation.

- Critical attributes:
  - Large number of users
  - High transaction volumes
  - Large stored data volumes
  - Significant modifications
- Very complex business rules



## Locking

One important aspect of the design of an integrated system such as NAV that is often overlooked until it rears its ugly head after the system goes into production, is the issue of "**Locking**". Locking occurs when one process has control of a data element, record, or group of records (in other words, part or all of a table) for the purpose of updating the data within the range of the locked data and, at the same time, another process requests the use of some portion of that data but finds it to be locked by the first process.

In the worst case, a "**deadlock**", there is a design flaw; each process has data locked that the other process needs and neither process can proceed. One of your jobs, as a developer or system implementer, is to minimize the locking problems and eliminate any deadlocks.

Locking interference between processes in an asynchronous processing environment is inevitable. There are always going to be points in the system where one process instance locks out another one momentarily. The secret to success is to minimize the frequency of these and the time length of each lock. Locking becomes a problem when the locks are held too long and the other locked-out processes are unreasonably delayed.

You might ask *What is an unreasonably delay?* For the most of the part, a delay becomes unreasonable when the users can tell that it's happening. If the users see stopped processes or simply experience counter-intuitive processing time lengths (that is, a process that seems like it should take 10 seconds actually takes two minutes), then the delays will seem unreasonable. Of course, the ultimate unreasonable delay is the one that does not allow the work to get done in the available time.

The obvious question is how to avoid locking problems. The best solution is simply to speed up the processing. That will reduce the number of lock conflicts that arise. Important recommendations for speed include:

- Restricting the number of active keys
- Restricting the number of active SIFT fields, eliminating them when feasible
- Carefully reviewing the keys, not necessarily using the "factory default" options
- Making sure that all disk accessing code is SQL Server optimized

Some additional steps that can be taken to minimize locking problems are:

- Always process tables in the same relative order.
- When a common set of tables will be accessed and updated, lock a "standard" master table first (for example, when working on Orders, always lock the Order Header table first).
- Process data in small quantities (for example, process 10 records or one order, then `COMMIT`, which releases the lock). This approach should be very, very cautiously applied.
- In long process loops, process a `SLEEP` command in combination with an appropriate `COMMIT` command to allow other processes to gain control (see the preceding caution).
- Shift long-running processes to off-hours.

You should also refer to the relevant documentation in the C/SIDE Reference Guide and the applicable NAV SQL Server documents.

## Design for updating

One must differentiate between "updating" a system and "upgrading" a system. In general, most of the NAV development work we will do is modifying individual NAV systems to provide tailored functions for end-user firms. Some of those modifications will be created as part of an initial system configuration and implementation, that is, before the NAV system is in production use. Other such modifications will be targeted at a system that is being used for day to day production. All these cases are "Updating".

Upgrading is when you implement a new version of the base C/AL application code and port all the previously existing modifications into that new version. We will cover issues involved in upgrading later.

Any time you are updating a production system by applying modifications to it, a considerable amount of care is required. Many of the disciplines that should be followed in such an instance are the same for an NAV system as with any other production application system. But some of the disciplines are specific to NAV and the C/SIDE environment. We'll review a representative list of both the types.

## **Customization project recommendations**

Some of these recommendations may seem patently obvious. That might be a measure of your experience and your own common sense. Even so, it is surprising that a number of projects go sour because one (or many) of the following suggestions are not considered in the process of developing modifications:

- One modification at a time
- Designing thoroughly before coding
- Designing the testing in parallel with the modification
- Using the C/AL Testability feature extensively
- Multi-stage testing:
  - Cronus for individual objects
  - Special test database for functional tests
  - Copy of production database for final testing as appropriate
  - Setups and implementation
- Testing full features:
  - User interface tests
  - System load tests
  - User Training
- Documenting and delivering
- Following up and moving on

## **One change at a time**

It is very important that changes made to the objects should be made in a very well organized and tightly controlled manner. In most situations, only one developer at a time will make changes to an object. If an object needs to be changed for multiple purposes, the first set of changes should be fully tested (at least through development testing) before the object is released to be modified for a second purpose.

If the project in hand is so large and complex or deadlines are so tight that this approach is not feasible, then you should consider use of a software development version control system such as Microsoft's Visual SourceSafe. Because version control systems don't interface smoothly with C/SIDE, some significant effort is required to use such a tool with NAV development, but sometimes the benefits are worth the effort.

Similarly, as a developer working on a system, you should only be working on one functional change at a time. As a developer, you might be working on changes in two different systems in parallel, but you shouldn't be working on multiple changes in a single system in parallel. It's challenging enough to keep all the aspects of a single modification to a system under control without having incomplete pieces of several tasks, all floating around in the same system.

If multiple changes need to be made simultaneously to a single system, one approach is to assign multiple developers, each with their own individual components to address. Another approach is for each developer to work on their own copy of the development database, with a project librarian assigned to resolve overlapping updates. This is one area where we should learn from the past. In mainframe development environments, having multiple developers working on the same system at the same time was common. Then the coordination problems were addressed and well-documented in professional literature. Similar solutions would still apply.

## Testing thoroughly

As you know, there is no substitute for complete and thorough testing. Fortunately, NAV provides some useful tools to help you to be more efficient than you might be in some other environment.



Now that NAV has the C/AL Testability Tools, all testing should utilize those tools to the greatest extent possible.

## Database testing approaches

If your modifications are not tied to previous modifications and specific customer data, then you may be able to use the Cronus database as a test platform. This works well when your target is a database that is not heavily modified in the area on which you are currently working. As the Cronus database is small, you will not get lost in deep piles of data. Most of the master tables are populated, so you don't have to create and populate this information. Setups are done and generally contain reasonably generic information.

If you are operating with an unmodified version of Cronus, you have the advantage that your test is not affected by other pre-existing modifications. The disadvantage, of course, is that you are not testing in a wholly realistic situation. Because the data volume in Cronus is so small, you will generally not detect a potential performance problem when testing in a Cronus database.

Even when your modification is targeted at a highly modified system where those other modifications will affect what you are doing, it's often useful to test a version of your modification initially in Cronus. This may allow you to determine if your change has internal integrity before you move on to testing in the context of the fully modified copy of the production system.

If the target database for your modifications is an active customer database, then there is no substitute for doing complete and final testing in a copy of the production database. You should also be using a copy of the customer's license. This way, you will be testing the compatibility of your work with the production setup, the full set of existing modifications, and of course, live data content and volumes. The only way to get a good feeling for possible performance issues is to test in a copy of the production database.

## **Testing in production**

While it is always a good idea to thoroughly test before adding your changes to the production system. Sometimes, you can safely do your testing inside the production environment. If the modifications consist of functions that do not change any data and can be tested without affecting any ongoing production activity, it may be feasible to test within the production system.

Examples of modifications that may be able to be tested in the live production system can range from a simple inquiry page or a new analysis report or export of data that is to be processed outside the system to a completely new subsystem that does not change any existing data. There are also situations where the only changes to the existing system are the addition of fields to existing tables. In such a case, you may be able to test just a part of the modification outside production (we'll discuss that mode of testing a little later), and then implement the table changes to complete the rest of the testing in the context of the production system.

## **Using a testing database**

From a testing point of view, the most realistic testing environment is a copy of actual production database. There are often good excuses about why it is just too difficult to test using a copy of the actual production database.



Don't give in to excuses—use a testing database!

Remember, when you implement your modifications, they are going to receive the "test of fire" in the environment of production. You need to do everything within reason to assure success. Let's review some of the potential problems involved in testing with a copy of the production database and how to cope with them:

- *It's too big*—is not a good argument relative to disk space. With USB disk drives available for less than \$0.15 US per GB, you can easily afford to have plenty of spare disk space.
- *It's too big*—is a better argument if you are doing file processing of some of the larger files (for example, Item Ledger, Value Entry, and so on). But NAV's filtering capabilities are so strong that you should relatively easily be able to carve out manageable size test data groups with which to work.
- *There's no data that's useful*—might be true. But it would be just as true for a test database, probably even more so, unless it were created expressly for this set of tests. By definition, whatever data is in a copy of the production database is what you will encounter when you eventually implement the enhancements on which you are working. If you build useful test data within the context of a copy of the production database, your tests will be much more realistic and, therefore, of better quality. In addition, the act of building workable test data will help to define what will be needed to set up the production system to utilize the new enhancements.
- *Production data will get in the way*—may be true. If this is especially true, then perhaps the database must be preprocessed in some way to begin testing or testing must begin with some other database, Cronus or a special testing-only mockup. As stated earlier, all the issues that exist in the production database must be dealt with when you put the enhancements into production. Therefore, you should test in that environment. The meeting and overcoming of challenges will prepare you for doing a better job at the critical time of going live with the newly modified objects.
- *We need to test repeatedly from the same baseline* or *We must do regression testing*—both are good points, but don't have much to do with what type of database you're using for the testing. Both the cases are addressed by properly managing the setup of your test data and keeping incremental backups of your pre-test and post-test data at every step of the way. In addition, the C/AL Testability Tools are explicitly designed to support regression testing.

Disk space is not a valid excuse for not making every possible useful intermediate stage backup. Staying organized and making lots of backups may be time consuming, but done well and done correctly, it is less expensive to restore from a backup than to recover from being disorganized or having to redo the job. Most of all, doing the testing job well is much less expensive than implementing a buggy modification.

## **Testing techniques**

As you are an experienced developer, you are already familiar with good testing practice. Even so, it never hurts to be reminded about some of the more critical habits to maintain.

First, any modification greater than trivial should be tested in one way or another by at least two people. The people assigned should not be a part of the team who created the design or code of the modification. It would be best if one of the testers is a sharp user because users seem to have a knack (for obvious reasons) of relating how the modification acts compared to how the rest of the system operates relative to the realities of the day-to-day work.

One of the testing goals is to supply unexpected data and make sure that the modification can deal with it properly. Unfortunately, those who were involved in creating the design will have a very difficult time being creative in supplying the unexpected. Users often enter data the designer or programmer didn't expect. For that reason, testing by experienced users is good. Another goal this approach addresses is that of obtaining meaningful feedback on the user interface before stepping into production.

The C/AL Testability Tools provide features to support testing how system functions deal with problem data. If possible, it would be very good to have users help to define test data, then use the Testability Tools to assure the modifications properly handle the data.

Second, after you cover the mainstream issues (whatever it is that the modification is intended to accomplish) you need to plan your testing to cover all boundary conditions. Boundary conditions are the data items that are exactly equal to the maximum or minimum or other range limit. More specifically, boundaries are the points at which input data values change from valid to invalid. Boundary condition checking in the code is where programmer logic often goes astray. Testing at these points is often very effective for uncovering data-related errors.

## **Deliverables**

Create useful documentation and keep good records of the complete testing. Testing scripts, both human-oriented and C/AL Testability Tool-based, are an important part of the records to be kept. Retain these records for future reference. Identify the purpose of the modifications from a business point of view. Add a brief, but complete, technical explanation of what must be done from a functional design and coding point of view to accomplish the business purpose. Record briefly the testing that was done. The scope of the record keeping should be directly proportional to the business value of the modification being made and the potential cost of not

having good records. All such investments are a form of insurance and preventative medicine. You hope they won't be needed but you have to allow for the possibility they will be needed.

More complex modifications will be delivered and installed by experienced implementers, maybe even by the developers themselves. With NAV, small modifications may be transmitted electronically to the customer site for installation by a skilled super-user. Any time this is done, all the proper and normal actions must occur, including those actions regarding backup before importing changes, user instruction (preferably written) on what to expect from the change, and written instruction on how to correctly apply the change. As a responsible developer, whenever you supply objects for installation by others, you must make sure that you always supply .fob format files (compiled objects), not text objects. This is because the import process for text objects simply does not have the same safeguards as does the import process for compiled objects.

## Finishing the project

Bring projects to conclusion, don't let them drag on through inaction and inattention — open issues get forgotten and then don't get addressed. Get it done, wrap it up, and then review what went well and what didn't, both for remediation and for application to future projects.

Set up ongoing support services as appropriate and move on to the next project. With the flexibility of the Role Tailored Client allowing page layout changes by both super users (configuration) and users (personalization), the challenge of User support has increased. No longer can the support person expect to know what display the user is viewing today.

Consequently, support services are almost certainly going to require an ability for the support person to view the user's display. Without that, it will be much more difficult, time consuming, and frustrating for the support personnel <--> user communication to take place. In many instances, this capability will have to be added to the Partner's support organization tool set and practices. There may be communications and security issues that need to be addressed at both the support service and the user site.

## Plan for upgrading

The ability to upgrade a customized system is a very important feature of NAV. Most complex corporate systems are very difficult to customize at the database-structure and process-flow levels. NAV readily offers this capability. This is a significant difference between NAV and the competitive products in the market.



Beyond the ability to customize is the ability to upgrade a customized system. While not a trivial task, at least it is possible with NAV. For other such systems, the only reasonable path to an upgrade is often to discard the old version and re-implement with the new version, recreating all customizations.

You may say, *Why should a developer care about upgrades?* There are at least two good reasons you should care about upgrades. First, because your design and coding of your modifications can have a considerable impact on the amount of effort require to upgrade a system. Second, because as a skilled developer doing NAV customizations, you might well be asked to be involved in an upgrade project. Since the ability to upgrade is important and because you are likely to be involved one way or another, we will review a number of factors that relate to upgrades.

## **Benefits of upgrading**

Just so we are on common ground about why upgrading is important to both the client and the MBS Partner, the following is a brief list of some of the benefits available when a system is upgraded:

- Easier support of a more current version
- Access to new features and capabilities
- Continued access to fixes and regulatory updates
- Improvements in speed, security, reliability, and user interface
- Assured continuation of support availability
- Compatibility with necessary infrastructure changes
- Opportunity to do needed training, data cleaning, and process improvement
- Opportunity to resolve old problems, to do postponed "housekeeping", create a known system reference point

This list is representative, not complete. Obviously, not every possible benefit will be realized in any one situation.

## **Coding considerations**

The toughest part of an upgrade is porting code and data modifications from the older version of a system to the new version. Sometimes the challenges inherent in that process cannot be avoided. When the new version has major design or data structure changes in an area that you have customized, it is quite possible that your modification structure will have to be re-designed and perhaps even be re-coded from scratch.

On the other hand, a large portion of the Microsoft created changes in a new version of a product such as NAV are often relatively modest in terms of their effect on existing code, at least on the base logic. That means, if modifications are done properly, it's not too difficult to port custom code from the older version into the new version. By applying what some refer to as "low-impact coding" techniques, you can make the upgrade job easier and thereby less costly.

## Careful naming

Most of the programming languages have Reserved Words. These are the words or phrases that are reserved for use by the system, generally because they are used by the compiler to reference predefined functions or system-maintained values. C/AL is no exception to this general rule. Only recently, due in part to the research for this book, a list of C/AL Reserved Words has been published as part of the NAV documentation. The list may still be incomplete, but, nevertheless, it should be referenced liberally.

If you choose a variable name which is the same as a C/AL Reserved Word, the compiler will generally recognize that fact. If, under some circumstance it does not, then it will provide unintended results. Such a possibility is slim, but it is relatively easy to avoid by prefacing all variable names with a two or three character string that will clearly identify the variable as part of your modification. You must be careful with this technique when naming variables to be used in conjunction with Automation Controllers. C/SIDE creates some of its own Automation Controller related variables by combining your variable names with suffixes. The combined names are then truncated to 30 characters—the maximum limit allowed for a C/SIDE variable name. If the names you have created are too long, this suffixing plus truncating process may result in some duplicate names.

Confusion can also result in the case of global and local variables with the same name or working storage and table variables of the same name. Two actions can minimize the possibility. First, minimize the instances where two variables have the same name. Second, whenever there is a possibility of name confusion, variable names should be fully qualified with the table name or the object name.

## Good documentation

In the earlier chapters, we discussed some documentation practices that are good to follow when making C/AL modifications. The following is a brief list of few practices that should be used:

- Identify every project with its own unique project tag
- Use the project tag in all documentation relating to the modification

- Include a brief but complete description of the purpose of the modification in a related `Documentation()` section (aka trigger)
- Include a description of the related modifications to each object in the `Documentation()` trigger of that object, including changes to properties, Global and Local variables, functions, and so on
- Add the project tag to the version code of all modified objects
- Bracket all C/AL code changes with inline comments so that they can be easily identified
- Retain all replaced code within comments, using `//` or `{ }`
- Identify all new table fields with the project tag

## Low-impact coding

We have already discussed most of these practices in other chapters. Nevertheless, it is useful to review them relative to our focus here on coding to make it easier to upgrade. You won't be able to follow each and every one of these, but will have to choose the degree to which you can implement low-impact code and which options to choose:

- As much as feasible, separate and isolate new code
- Create functions for significant amounts of new code by using single code line function calls
- Either add independent Codeunits as repositories of modification functions or, if that is overkill, place the modification functions within the modified objects
- If possible, add new data fields; don't change the usage of existing fields
- When the functionality is new, add new tables rather than modifying existing tables
- For minor changes, modify the existing pages, else copy and change the clone
- Create and modify copies of reports and XMLports rather than modifying the original versions in place
- Don't change field names in objects, just change captions and labels if necessary

In any modification, you will have conflicting priorities regarding doing today's job in the easiest and least expensive way versus doing the best you can do to plan for the future. The right decision is never a black and white choice, but must be guided by subjective guidelines as to which choice is really in the customer's best interest.

## The upgrade process

We won't dwell here on the actual process of doing an upgrade except to describe the process at the highest level and the executables-only option.

### Upgrade executables only

The executables are the programs that run under the operating system. They are individually visible in a disk directory and include `.exe` and `.dll` files. Since the Navision Windows product was first shipped in 1995, the executables delivered with each new version of NAV (or Navision) have been backward compatible with the previous versions of the objects. In other words, you could run any version of the database and objects under any subsequent version of the server and client executables.

This backward compatibility allows the option of upgrading only the executables for a client. This is a relatively simple process, which provides access to enhanced compatibility with infrastructure software (for example, Windows desktop and server software, and so on), provides access to added features relating to communications or interfacing, and often provides faster and more reliable processing. Upgrading the executables will also provide access to the C/AL features and user-accessible features that are part of the new version. Some folks use the term "Technical Upgrade" for the act of upgrading the executables.

The process of upgrading the executables just requires replacing all the files related to the executables on the server and clients, typically through doing a standard installation from the distribution CD. This will include the server software, the client software, the executables, and libraries for auxiliary tools such as the Application Server, N/ODBC, and C/FRONT. Then convert the database to be compatible with the new version (preferably through a backup and restore into a new, empty database).



Remember, upgrading the executables is a one-way process. Once done, it cannot be undone.

Like any other change to the production system that affects the data, such an upgrade should be thoroughly tested before implementing it in production.

## Full upgrade

A full upgrade includes the aforesaid executables upgrade, but that is the simplest part of the process. The full upgrade process consists of a clearly defined multi-step project, which is best handled by developers who are specifically experienced in the upgrade process. It is critical to keep in mind that the customer's business may be at stake based on how well and how smoothly an upgrade is carried out.

The following list is a summary of the steps involved in a full upgrade of an NAV system:

- Identify all modifications, enhancements, and add-ons by comparing the full production set of objects against an unmodified set of the same version of the objects as distributed by Microsoft. This is always done by exporting the objects to be compared into text files, then importing the resulting text files into a comparison tool. The comparison tools that are specifically designed to work with C/AL provide capabilities that general-purpose programmer editors don't have, though these can also be used. C/AL-oriented tools include the Developer's Toolkit from Microsoft or the Merge Tool found at [www.mergetool.com](http://www.mergetool.com).
- Plan the upgrade based on which customizations need to be ported, which ones should be discarded because they have been superseded or made obsolete, which ones will need to be re-developed, what special data conversions and new setups and user training will be required, and what upgraded add-ons must be obtained. Identify any license issues.
- Beginning with a current copy of the production database and a distribution of the new version, create a set of new version objects containing all the customizations, enhancements, and add-ons (as previously planned) that were contained in the old version.
- Create modifications to the standard data conversion routines as necessary.
- Convert a full set of (backup) production data, then combine it with the upgraded objects to create a testing database.
- Work with experienced customer super users to set up, then test the upgraded system, identify any flaws, and resolve them. In parallel, address any training requirements.
- Continue testing until the system is certified "ready to use" by both client and Partner test team members.
- Do a final production data conversion, using the final upgraded object set to create a new production database for go-live use.

A minimal upgrade will take two or three weeks, a lightly customized system may take a couple of months, and a highly customized system will take more than that. If multiple sites are involved, then it will significantly add to the complexity of the upgrade process—particularly those parts of the process where users are directly involved.

## Supporting material

Part of the supporting material, relevant to NAV development, are the Implementation tools available from Microsoft. You should review them to know what they do and how to utilize them.

## Sure Step

Sure Step is an implementation methodology developed by Microsoft to aid in the implementation of Microsoft Dynamics products in a standard, cost effective, reliable way. Sure Step consists of a combination of tools such as project management guides, "best practices" information, deployment, and migration templates, as well as product and industry specific tools.

As of late 2009, in a commendable decision by Microsoft management, Sure Step was made available to all Dynamics Partners at no additional charge. All Partners should use Sure Step, even if for no more than access to the huge library of materials it contains.

From the project manager's point of view, Sure Step is a standardized set of comprehensive, editable templates designed to help create a project plan and the important supporting documents. Sure Step collects a considerable amount of implementation experience, along with the related documents, into a tool set designed for customization to fit each specific project. As with all comprehensive general use tools, for any particular project there is a lot of material that won't apply. The important benefits are the completeness of the material combined with the structure that it provides so that implementation project processes can be repeatable and thus predictable.

## RIM

One of the time-consuming and moderately complex parts of implementing a new system, is the gathering and loading of data into the system. Generally, master tables such as Customers, Vendors, and Items must be loaded before you can even begin serious system testing and training. In general, this data is loaded for testing and training and at final cutover for the production use.

In order to assist in this process, Microsoft provides a set of tools called the **Rapid Implementation Methodology (RIM)**. This tool is available to all NAV Partners. The documentation for RIM is in the manual *Dynamics NAV RIM Users Guide 2.0*. As of this writing, Microsoft is in the process of developing documentation to support the integrated use of Sure Step and RIM. This will help users implement the RIM data migration tools in an optimized way in the context of a Sure Step structured project.

Basically RIM consists of a set of questionnaires, some industry-specific data templates, and the associated Import and Export routines. The recommended process has initial data entry occurring in Excel spreadsheets, then exported to XML files and imported into NAV for use. You should review these tools. Even if you find the tools aren't the right answer for your situation, you will learn some useful techniques by studying them.

## Other reference material

With every NAV system distribution there is an included set of reference guides. These are highly recommended. There are a number of other guides available, but sometimes you have to search for them. Some were distributed with previous versions of the product but not with the latest version. Some are posted at various locations on PartnerSource or another Microsoft website. Some may be available on one of the forums. Most are readily available to Partner development personnel, though that may be more by force of habit than as a policy.

In nearly every case you will find these documents a very good starting place, but you will be required to go beyond what is documented, experimenting, and figuring out what is useful for you. If you are working with a single system, you are likely to narrow in on a few things. If you are working with different systems from time to time, then you may find yourself working with one aspect of one tool this month and something entirely different next month.

Here is a list of some documentation you will be interested in (when you look for those with "Navision", they might have been changed to "Dynamics NAV"). A number of filenames are included, especially when they are not easy to interpret. For example, names starting with w1 are from Worldwide product distribution. Because the focus on documentation in Dynamics NAV V2009 has changed from manuals to interactive Help, you may have to search for NAV V5 (or earlier) copies of many of these. Some of these will be (or were) part of the standard system distribution. Some have been distributed on a separate Tools CD, some are only available as downloads from PartnerSource.

- Application Designer's Guide (the C/AL "bible" up through V5)  
—w1w1adg.pdf

- Terminology Handbook—[wlwlterm.pdf](#)
- Installation & System Management of Application Server for Microsoft Dynamics™ NAV—[wlwlatas.pdf](#)
- Installation & System Management of C/SIDE Database Server for Microsoft Dynamics™ NAV—[wlwlism.pdf](#)
- Installation & System Management of SQL Server Option for the C/SIDE Client—[wlwlsql.pdf](#)
- Microsoft Dynamics™ NAV ODBC Driver 5.0 Guide—part of the NAV 2009 system distribution—[wlwlnocbc.pdf](#)
- Making Database Backups in Microsoft Dynamics™ NAV—[wlwlbkup.pdf](#)
- C/FRONT Reference Guide—part of the system distribution—[wlwlcfrent.pdf](#)
- Security Hardening Guide—part of the NAV 2009 system distribution
- Navision Developer's Toolkit
- NAV Tools CD:
  - Microsoft Business Solutions—Navision SQL Server Option Resource Kit—whitepaper
  - Performance Troubleshooting Guide for Microsoft Business Solutions—Navision—[wlwlperftguide.pdf](#)
  - Application Benchmark Toolkit
  - User Rights Setup
- Microsoft Business Solutions—Database Resource Kit
- C/AL Programming Guide
- Dynamics NAV RIM Users Guide 2.0
- Microsoft Dynamics™ NAV Training Manuals and Videos (various topics)
- Many whitepapers on specific NAV (Navision) application and technical topics (even the old ones are useful)

There is other documentation that you will find valuable as you move into specialized or advanced areas. But many of the preceding are general purpose and frequently helpful.

Last, but definitely not least, become a regular visitor to websites for more information and advice on C/AL, NAV, and many more related and unrelated topics. The websites [dynamicsuser.net](#) and [www.mibuso.com](#) are especially comprehensive and well attended. Other, smaller or more specialized sites are also available.



An additional group of websites focused on NAV 2009 have come on the scene since 2008. These are the weblogs of a variety of NAV experts and aficionados. Many of the bloggers are part of the NAV development team at Microsoft, some are by NAV MVPs and others are by those who simply want to share their knowledge of and affection for NAV with the rest of us.

Some of the ones available as of the writing of this book are:

- Microsoft Dynamics NAV Team Blog: [blogs.msdn.com/nav/](http://blogs.msdn.com/nav/)
- Clausl's Dynamics NAV Blog: [blogs.msdn.com/clausl/](http://blogs.msdn.com/clausl/)
- Christian's Blog: [blogs.msdn.com/cabeln/](http://blogs.msdn.com/cabeln/)
- Freddy's Blog: [blogs.msdn.com/freddyk/](http://blogs.msdn.com/freddyk/)
- Kine's info: [msmvps.com/blogs/kine/](http://msmvps.com/blogs/kine/)
- NAV Reporting: [blogs.msdn.com/nav-reporting/](http://blogs.msdn.com/nav-reporting/)

More such useful blogs appear fairly often. The author's firm has one planned for the near future to discuss all things about NAV.

## Into the future...

As we've gone through our study of NAV 2009, you have seen a number of instances where Microsoft tools outside of the C/AL-C/SIDE environment have become an integral and necessary part of the NAV toolset. An immediately obvious example is the Visual Studio Report Designer layout tool. The second obvious example is the C# language. While it's not necessary to learn C# to development complete and complex application enhancements for NAV, it will certainly prove to be handy for a number of tasks. Examples of some C# oriented tasks are:

- Debugging in Visual Studio
- Creating new Client Add-ins
- Creating web services consuming code

The future, the direction of the Microsoft Dynamics NAV product is obviously moving away from the insular, self-contained approach of the past. Product development changes are taking us down twin paths of both being more integrated with the Microsoft family of products and being a more open system. Here are a few changes that one might predict for Dynamics NAV in the foreseeable (or guessable) future:



**Disclaimer:** These are not announcements nor are they based on information from Microsoft. These are extrapolations based on extrapolations, guesses, and rumors.

- A Sharepoint client
- More support for various display devices or device types as rendering targets
- Dropping the C/SIDE Server, focusing on SQL Server
- Eventually dropping the Classic Client
- More integration with the .NET/Visual Studio development environment
- More integration with C#, at some point C/AL and C# code partially or completely interchangeable
- Integration with "the Cloud"
- More documentation integrated and online
- Expansion of the web services functionality
- Enhancement of the Client Add-in style capability to other parts of the system (that is, an "opening up" to more ways of integrating externally created material)
- Expansion of the integration with Microsoft tools for Business Intelligence, Web based activities, and other areas

None of these future projections are sure things, but some of them are inevitable. The most significant point here is that as a NAV developer, you will want to expand your skills as time goes by. As soon as you feel comfortable in C/AL and C/SIDE development, you may want to develop some expertise in the use of Visual Studio and basic knowledge of C# programming. It doesn't appear that you should worry about getting bored in the near future.

## Summary

We have covered a lot of topics in this book with the goal of helping you to become productive in C/AL development. Hopefully that has happened much more quickly than if you hadn't invested your time here. From this point on, your assignments are to continue exploring and learning, enjoy NAV, C/SIDE, and C/AL, and to do your best in all things.

*The nearest way to glory is to strive to be what you wish to be thought to be – Socrates*

## Review questions

1. Which of the following is not an option for integrating a NAV 2009 system and another external system? Choose one:
  - a. XMLports
  - b. Linked SQL Server databases
  - c. Text file import and export
  - d. Email messages
2. In the Role Tailored Client, XMLports can be used to process XML formatted files as well as other text file formats. True or False?
3. XML data files are becoming obsolete and will soon be replaced by a more advanced data structure for interfacing systems. True or False?
4. XMLports must be run from a Codeunit object in the Role Tailored Client. True or False?
5. XMLports cannot contain C/AL code. All data manipulation must occur outside of the XMLport object. True or False?
6. Which of the following are interface tools for NAV? Choose three:
  - a. C/OCX
  - b. C/AL
  - c. C/SIDE
  - d. C/FRONT
  - e. NAS
7. Which of the following is the Client Add-in feature? Choose one:

The ability to add a new client of your own design to NAV 2009

  - a. A tool to provide for extending the Role Tailored Client User Interface behavior
  - b. A special calculator feature for the RTC client
  - c. A new method for mapping Customers to Contacts

8. Web services are an industry standard Application Programming Interface defined by the World Wide Web Consortium. True or False?
9. Software external to NAV that accesses NAV Web Services must be dot NET compatible. True or False?
10. The Help files for NAV cannot be customized by Partner or ISV developers. True or False?
11. When planning a new NAV development project, it is good to focus the design on the data structure, required data accesses, validation and maintenance. True or False?
12. The developer cannot affect the way that "Locking" occurs in a NAV database. True or False?
13. Customized NAV systems cannot be upgraded to new versions. True or False?
14. Which of the following are good coding practices? Choose three:
  - a. Careful naming
  - b. Good documentation
  - c. Liberal use of wildcards
  - d. Design for ease of upgrading



# Answers

The answers to all the review questions, given at the end of after every chapter, are listed here.

## Chapter 1

1. False
2. a, b, d
3. False, it is an object-based system
4. a, d
5. True, though some development can be done within Visual Studio or another report layout tool, that development starts and ends within C/SIDE
6. False, all except MenuSuites
7. True
8. True, but both can be run in parallel
9. False, Ledger data is always permanent
10. True
11. b, c, d
12. False, though there is overlap, they are quite different
13. True
14. False, Tables can be run from the Object Designer screen

## Chapter 2

1. a, b, c
2. False
3. False
4. a, c, d
5. False
6. False
7. True
8. True
9. True
10. True
11. False
12. a, c, d
13. True
14. False
15. False
16. True

## Chapter 3

1. False, max is 30 characters
2. True
3. c
4. a, b, d
5. False
6. b
7. a, c, e
8. False, Primary Key entries must be unique
9. True
10. False, wildcards are a valuable tool
11. c, d, e

## Chapter 4

1. False
2. False
3. b, d, e
4. True
5. c
6. True
7. Home and Departments
8. True
9. a, c
10. False, in almost all instances you can only do basic layout in the Wizard and must do the rest of the development work within the Page Designer
11. False, this would be totally contrary to the design philosophy of NAV as a customizable system

## Chapter 5

1. False, NAV 2009 reports can be displayed onscreen, and can have dynamic features such as sorting and sections which expand/collapse with a mouse click
2. False, the definition must be done in the Sections area and will then appear in the Data Sources area, but cannot be added to the Data Sources area
3. True
4. True
5. b, c
6. True
7. a, c, data in Sections and Request Form apply to Classic Reports only
8. b, d, e
9. b
10. True



## Chapter 6

1. False, MenuSuites, which become the Departments page, cannot contain C/AL code
2. True
3. a, b
4. False
5. a, b, d
6. a
7. False, numbering and naming are very important and must follow the standards set by NAV
8. True
9. c
10. False
11. True
12. Usually False, though exceptions are always possible

## Chapter 7

1. a, b, d
2. False
3. False
4. True
5. True
6. a, b, c
7. a, c
8. b, it should be a WHILE - DO
9. d
10. b
11. True

## Chapter 8

1. False, data should never be entered directly into a Ledger
2. True
3. c
4. False
5. d
6. True
7. Individual answers:
  - a. False
  - b. False
  - c. False
  - d. False
8. a, b, d
9. True
10. True
11. False
12. b

## Chapter 9

1. d
2. True, in the RTC XMLports have taken over the Classic Client Dataport role
3. False, XML interfacing is becoming more common.
4. False, this was true for the Classic Client, but not for the RTC
5. False
6. a, d, e
7. b
8. True
9. False
10. False
11. True
12. False
13. False
14. a, b, d



# Index

## Symbols

- \* 161
- ? 162
- @ symbol 162
- XML port triggers
  - about 514
  - Documentation() 514
  - OnInitXMLport() 514
  - OnPostXMLport() 514
  - OnPreXMLport() 514

## A

- accounting data 111
- advanced interfaces tools
  - about 527
  - Automation Controller 528
  - C/FRONT 529
  - C/OCX 529
  - Linked Server Data Sources 529
  - NAV Application Server (NAS) 529
  - NAV Communication Component 528
- API 528
- application
  - activity tracking tables, adding 97
  - keys 95
  - reference table, adding 98, 102
  - SumIndexFields 103-106
  - tables, creating 82
  - tables, modifying 82
- application design
  - all forms 41
  - Card pages 38
  - Card page, creating 41-45, 218
  - field numbering 34-36

- forms 36
- list form, creating 45, 47
- list format report, creating 52-57
- List page, creating 45
- main/sub forms 39
- Pages 36
- Pages, Card pages 38, 39
- Pages, document pages 39
- Pages, Journal and Worksheet pages 40
- Pages, List pages 38
- Pages, standard elements 41
- Pages, types 37
- reports 50, 51
- tables 32
- tables, creating 33-35
- tables, designing 32, 33
- tabular forms 38

**Application Programming Interface.** *See*

**API**

**automation, complex data types**

- automation data type 142
- OCX 142

**Automation Controller**

- about 528
- use 528

## B

**backups 62**

**BEGIN-END function**

- about 360
- compound statement 361

**BI 18**

**Binary Large Object.** *See* **BLOB**

**BLOB 150**

**Business Intelligence.** *See* **BI**

## C

### C/AL

- about 379, 384
- C/AL Symbol Menu 380
- data conversion functions 390
- DATE functions 391
- filtering functions 407
- flow control 397
- FlowField 394
- functions 350
- INPUT functions 403
- internal documentation 381-383
- InterObject communication 411
- naming convention 328, 329
- SumIndexFields 394
- syntax 343
- Symbol Menu 380
- validation utility functions 384

### C/AL code modifications

- about 363
- report enhancing, by adding code 368-374
- validation, adding to table 363-367

### C/AL functions

- about 354, 361
- BEGIN-END function 360
- code, indenting 362
- CONFIRM function 352, 353
- dialog functions 350
- ERROR function 351, 352
- FIND function 357
- GET function 356
- MESSAGE function 350
- SETCURRENTKEY function 355
- SETRANGE function 356

### C/AL Symbol Menu

- about 380
- Global symbols 381
- Local symbols 381
- use 381

### C/AL syntax

- () operator 346
- [] operator 346
- about 343
- arithmetic operator 347
- assignment 343
- expressions 345

- operators 346
- punctuation 343
- Range operator 347
- Scope operator 347
- single dot operator 346
- Wildcard character 344

### C/Front 529

### C/OCX 529

### C/SIDE

- accessing 23, 24
- C/AL 23
- data definitions 326
- essential practices 325
- functions 335
- navigating through 312
- Object Designer 312
- object numbering 324
- programming 334
- variables 329
- database 25
- Object designer tool icons 24, 25

### C/SIDE Integrated Development

Environment. *See* C/SIDE

### C/SIDE RD tool 245

### CALCFIELDS function 396

### CALCSUMS 396

### CalculateNewDate() function 146

### callable functions

- about 469, 470
- codeunit 358 470
- codeunit 359 471, 472
- codeunit 365 473, 474
- codeunit 396 474, 475
- codeunit 397 475
- codeunit 408 475, 476
- codeunit 412 476, 477

### Card page 188

### careful naming 561

### Client Add-in feature

- about 530
- comments 534
- creating 531-533
- defining 530

### Client Monitor tool 498

### Code Coverage tool 498

### code indenting 362

### codeunit 1 479

- codeunit 228 478**
  - test reports, printing 478
- codeunit 229**
  - about 478
  - documents, printing 478
- codeunit 358**
  - about 470, 471
  - CreateAccountingPeriodFilter function 470
  - CreateFiscalYearFilter function 470
- codeunit 359**
  - about 471, 472
  - CreatePeriodFormat function 472
  - FindDate function 471
  - NextDate function 472
- codeunit 365**
  - about 473, 474
  - address, formatting 473, 474
- codeunit 396**
  - about 474, 475
  - unique identifying number 474, 475
- codeunit 397**
  - about 475
  - mails, managing 475
- codeunit 408**
  - about 475, 476
  - dimension, management 475, 476
- codeunit 412**
  - about 476, 477
  - Common Dialog, managing 476, 477
- codeunit 419 479**
- codeunit 5054 479**
- codeunit 5063 479**
- codeunit 5300 thru 5314 479**
- codeunit 5813 thru 5819 479**
- codeunit 800 479**
- codeunit 801 479**
- codeunit 802 479**
- codeunit 81 479**
- codeunit 82 479**
- codeunit 90 479**
- codeunit 91 479**
- codeunit 92 479**
- codeunits**
  - about 58, 59
  - advantages 58
  - need for 58
- complex data types**
  - about 141
  - Action 149
  - automation 142
  - BigText 150
  - BLOB 150
  - data structure 141
  - DateFormula 143
  - FieldRef 149
  - GUID 150
  - Input/Output 142
  - KeyRef 149
  - objects 142
  - RecordID 149
  - RecordRef 149
  - TableFilter 149
  - Transaction Type 150
  - Variant 149
- components, XML ports**
  - about 510
  - attribute 523, 525
  - data lines 514-519
  - element 523, 525
  - line properties 520, 523
  - line triggers 524, 525
  - NodeType, as an element 524
  - properties 510, 512
  - properties, defining 512
  - Request Page 526, 527
  - TagType, as an attribute 524
  - triggers 514
- Confirmation(Dialog) page 192**
- CONFIRM function 352, 353**
- Content-modifiable tables**
  - about 117
  - System table 117
- controls 26**
- Control triggers 228**

**D**

- database**
  - about 25
  - logical database 25
  - physical database 25

**data conversion functions**

EVALUATE 391

FORMAT 390

**Data Fields 255****data focused design**

about 547

data design sequence 549

data tables, defining 548

data validation, designing 549

posting processes, designing 550

rechecking 550

required data, determining 548

required data views, defining 548

small goals, defining 547

support processes, designing 550

user access interface, designing 548, 549

**data focused design, new functionality**

about 547

data sources, defining 548

data views, defining 548

required data, defining 548

small goals, defining 547

**Data Item properties**

CalcFields 269

DataItemIndent 266

DataItemLink 266, 267

DataItemLinkReference 266

DataItemTable 266

DataItemTableView 266

DataItemVarName 269

GroupTotalFields 268

MaxIteration 269

NewPagePerGroup 268

NewPagePerRecord 268

PrintOnlyIfDetail 269

ReqFilterFields 268

ReqFilterHeader 268

ReqFilterHeadingML 268

TotalFields 268

**data item sections**

run time formatting 280

**Data item triggers**

Documentation() 269

OnAfterGetRecord() 270

OnPostDataItem() 270

OnPreDataItem() 270

**Dataports 59, 60****data structure, complex data types**

file 141

record 142

**DataType as table**

Export::OnAfterGetRecord() 525

Export::OnPreXMLItem() 525

Import::OnAfterInitRecord() 525

Import::OnAfterInsertRecord() 525

Import::OnBeforeInsertRecord() 525

**data types**

about 138

complex 141

fundamental 138

using 150, 151

**Date/Time data, fundamental data types**

date 139, 140

DateTime 141

duration 141

time 141

**date constant 139****DateFormula, complex data types**

alpha time units 143

examples 143

math symbols 143

numeric multipliers 143

positional notation 143

using 144-149

**DATE functions**

CALCDATE 393, 394

DATE2DMY 391

DATE2DWY 392

DMY2DATE 392, 393

DWY2DATE 392, 393

**deadlock 552 552****Debugger 497****Department page 195, 196****design for updating**

about 553

project recommendations, customizing 554

**designing for efficiency**

about 551

disk I/O 551

disk I/O, attributes 551

locking 552

locking problems, avoiding 552

**Developer's Toolkit**

- about 483, 484
- Compare and Merge Tools 483
- object relations 486
- source access 486
- Source Analyzer 484
- table relations 484-486
- testing 488-491
- Where Used 487, 488

**development challenge**

- donor recognition level, developing 419
- ICAN test data, creating 413
- task 413

**development projects**

- creating 544
- examples 544
- existing functional area, modifying 546, 547
- functional area, creating 545
- new functionality design,
  - advantages 545, 546
- overall understanding 545
- time allocation 547

**development tools**

- Client Monitor 498, 499
- implementation tool 565
- reference material 566, 567

**dialog form 192****dialog function, debugging**

- about 500
- CONFIRM statements 500
- DIALOG function 500, 501
- ERROR function 501, 502
- MESSAGE statements 500
- techniques 500-502
- text output 501

**documentation**

- about 62
- Documentation trigger 382
- internal documentation 381-383

**Document page**

- about 188
- FastTabs 189, 190

**document report**

- about 249, 253
- Customer Sales-Invoice document
  - report 249

**donor recognition level, developing**

- about 420
- C/AL coding, starting with 421, 422
- CALCFIELDS processing, code addition
  - 425, 426
- print addresses, adding code to 426-432
- Report Wizard, using 420
- retrofitting date filtering
  - capability 422, 424

**E****Enterprise Resource Planning. *See* ERP system****ERP system**

- about 14
- BI 18
- BI, functions 18
- financial management 16
- financial management, accounts payable 16
- financial management, accounts
  - receivable 16
- financial management, analytical
  - accounting 16
- financial management, cash management
  - and banking 16
- financial management, General Ledger 16
- financial management, inventory and fixed
  - assets 16
- financial management, Multi-Currency and
  - Multi-Language 16
- NAV Human Resource
  - Management (HR) 19
- NAV manufacturing 16
- NAV manufacturing, capacity 17
- NAV manufacturing, functions 17
- NAV manufacturing, Product Design 17
- NAV Project management 19
- NAV Relational Management (RM) 18
- NAV Supply Chain Management (SCM) 17
- NAV Supply Chain Management (SCM),
  - applications 17
- reporting 18

**ERP sysytem**

- components 15
- integrating 15
- viewing, from highest level 15, 16



**ERROR function** 351, 352

**eXtensible Markup Language.** *See* **XML ports**

**external interfaces**

interface tools 527

XMLports 508

## **F**

**F11 key** 325

**FactBoxes, Page parts**

about 198

Card parts 198

Chart pane 199

List parts 198

**field**

about 126

class 152

data structure, examples 136

numbering 132

properties 126

triggers 135

variable, naming 137

**field, properties**

accessing 126

AltSearchField 129

AutoIncrement 130

BlankNumbers 130

BlankZero 130

BLOB properties 128

BLOB properties, compressed 128

BLOB properties, owner 128

BLOB properties, SubType 128

caption 127

Caption Class 129

CaptionML 127

CharAllowed 129

DataLength 129

data type 127

DateFormula 129

DecimalPlaces 130

description 127

Editable 129

enabled 127

ExtendedDataType 130

Field No 127

InitValue 129

MaxValue 130

MinValue 130

name 127

NotBlank 129

Numeric 129

Option data type, OptionCaptionML  
property 132

Option data type, OptionCaption  
property 132

Option data type, OptionString  
property 131

SignDisplacement 130

SQL Data Type 129

TableRelation 130

TestTableRelation 130

ValidateTableRelation 130

ValuesAllowed 129

**field class**

FlowField 152

FlowField, calculation formula 153

FlowField, table filter 155

FlowField, types 153, 154

FlowFilters 155-159

Normal 152

**field numbering**

about 133

Classic Client, using 133

data type, changing 134, 135

for new fields 133

**fields 26**

**fields, page controls**

DecimalPlaces property 216

editable property 216

ExtendedDatatype property 217

HideValue property 216

importance property 216, 217

MultiLine property 216

OptionCaptionML property 216

OptionCaption property 216

visible property XE 216

**field triggers**

OnLookup() trigger 136

OnValidate() trigger 136

**filter controls, filter values**

Classic Client access 172

Classic Client access, Field Filter 172

Classic Client access, Flow Filter 172

- Classic Client access, Show All 173
- Classic Client access, Sort 173
- Find-as-you-type filtering 174
- RoleTailored Client access 173, 174
- filtering**
  - about 158, 159
  - CLEARMARKS function 410
  - COPYFILTER function 409
  - COPYFILTERS function 409
  - filter values, defining 158, 159
  - functions 407
  - GETFILTER function 409
  - MARKEDONLY function 410
  - MARK function 409
  - RESET function 410
  - SETFILTER function 408
  - SETRANGE function 408
  - structure, defining 158, 159
- filtering functions**
  - about 407
  - CLEARMARKS function 410
  - COPYFILTER function 409
  - COPYFILTERS function 409
  - GETFILTER function 409
  - GETFILTERS function 409
  - MARKEDONLY function 410
  - MARK function 409
  - RESET function 410
  - RESET function, filter groups 410, 411
  - SETFILTER function 408
  - SETRANG function 408
- filter syntax, defining.** *See* **filter values, defining**
- filter values, defining**
  - by ranges 160
  - combinations, using 162
  - filter, experimenting with 163-171
  - filter controls, accessing 172
  - on equality 160
  - on inequality 160
  - with Boolean operators 161
  - with wild cards 161, 162
- FIND function**
  - [Which] parameter 358, 359
  - about 357
  - FIND(‘+’) 358
  - FIND(‘-’) 358, 359
  - FINDFIRST 358
  - FINDLAST 358
  - FINDSET 358, 360
  - forms 358
  - GET function, differentiating between 357
  - SQL Server FIND option 359
- Flow control**
  - BREAK function 402
  - CASE - ELSE statement 398, 399
  - EXIT function 402
  - QUIT function 401
  - REPEAT - UNTIL control structure 397
  - SHOWOUTPUT function 403
  - SKIP function 402
  - WHILE - DO control structure 397, 398
  - WITH - DO statement 400
- FlowFields function**
  - CALCFIELDS 395
  - CALCSUMS 396
- form controls**
  - about 205
  - experimenting with 218
  - inheritance 211
- forms**
  - Card page, creating 88-90
  - creating 88
  - List page, creating 90-92
  - TableRelation property, testing 93, 94
  - ZUP file 92
- functional area**
  - advantages 545
  - creating 545
  - data focused design 547
  - enhancing 546
- functions**
  - about 335, 336
  - Codeunits with functions 336
  - creating 337-342
  - DATE2DMY 335
  - designing 339
  - GET 335
  - INSERT 335
  - integrating 341-343
  - MESSAGE 335
  - new functions, need for 337
  - STRPOS 335
  - trigger 335

## **function terminology, NAV 2009**

- Batch 29
- Document 29
- Journal 28
- Ledger 28
- Posting 29
- Register 29

## **fundamental data types**

- Date/Time data 139
- numeric data 138
- string data 139

## **G**

### **GET function 356**

#### **global identifiers**

- about 329
- global functions 330
- global text constants 329

#### **Globally Unique Identifier. See GUID**

#### **Graphical User Interface. See GUI**

#### **Group, page controls**

- editable property 212
- enabled property 212
- FreezeColumnID property 214
- GroupType property 212, 213
- IndentationColumnName property 214
- IndentationControls property 214
- ShowAsTree property 214
- visible property 212

#### **GUI 181**

#### **GUID 150**

## **H**

### **Help Toolkit**

- about 542
- customizing 543, 544

## **I**

### **ICAN, NAV development**

- about 31
- application design 32

### **ICAN application**

- about 229
- activity-tracking tables, adding 97

- Card pages, creating 232, 233

- forms, creating 87

- page object name 229

- reference tables, adding 98

- related list, creating 231

- secondary keys, adding 95, 96

- simple list page, creating 230, 231

- SumIndexFieldS 103, 104

- tables 229

- table, integrating 104-107

- tables, creating 82

- tables, modifying 82

### **ICAN test data, creating**

- C/AL code, defining 417-419

- C/AL Globals screenshot 416

- DataItem, defining 414

- Donor Giving report design 414

- new report, starting with 414

- Request form/page, eliminating 414, 415

### **IDE 14**

### **IF-THEN-ELSE statement 361**

### **Independent Software Vendor. See ISV**

### **Input/Output, complex data types**

- dialog 142

- InStream 143

- Outstream 142

### **INPUT functions**

- DELETEALL 406

- DELETE 405

- INSERT 404

- MODIFYALL 406

- MODIFY 405

- MODIFY, Rec 405

- MODIFY, xRec. 405

- NEXT 403

### **Integrated Development Environment. See**

#### **IDE**

### **integration tools**

- about 61

- automation 61

- C/Front 61

- C/OCX 61

- N/ODBC 61

- web services 61

### **interfaces**

- about 507

- advanced tools 527

## **interface tools**

- about 527
- Automation Controller 528
- C/Front 529
- C/OCX 529
- Linked Server Data Sources 529
- NAV Application Server 529
- NAV Communication Component 528

## **International Community and Neighbors.**

*See* ICAN application

## **InterObject**

- communicating, via data 411
- communicating, via function parameters 411
- communicating, via object calls 412
- communicating, via RUN 412
- communicating, via RUNMODAL 412
- via data 411
- via function parameters 411

## **ISV 27**

## **J**

## **Journal/Worksheet page**

- Sales Journal page 191

## **K**

## **keys**

- about 74
- adding 95
- attributes 77
- Primary Key 75, 76
- secondary keys, adding 95, 96
- SQL Server-specific properties 77

## **L**

## **license 28**

## **line properties, XML ports**

- about 520
- indentation 520
- nodeName 520
- NodeType 520
- SourceType 520
- SourceType as Field 523
- SourceType as Table 522
- SourceType as text 521

## **line triggers, XML ports**

- about 524
- DataType as field 525
- DataType as table 525
- DataType as text 525

## **Linked Server Data Sources 529**

## **List+ page 191**

## **List page, application design**

- creating, steps 46-49
- keyboard shortcuts 49
- Run a table option 50

## **List pages 187**

## **list report 248, 253**

## **local variables**

- about 330
- function local variables 330
- trigger local variables 330

## **locking**

- about 552
- avoiding 552
- deadlock 552
- problems, avoiding 552
- problems, minimizing 553

## **M**

## **management codeunits 479, 480**

## **MenuSuites 59**

## **MenuSuite structure**

- about 463
- configuring 467
- development 463-466
- personalizing 467, 468
- transformation 467

## **MESSAGE functions 350**

## **Microsoft Dynamics NAV. *See* NAV 2009**

## **modifications**

- documenting 480, 481

## **multi-currency 482**

## **multi-currency system 482**

## **multi-language**

- about 481
- features 481

## **multi-language system**

- about 481
- features 481

## **N**

### **NAV**

- advanced development 468
- as ERP 482
- Communication Component 528
- debugging, in Visual Studio 499
- development projects, creating 544
- development time allocation 547
- development with C/AL 379
- filtering tool 158, 159
- form 181
- forms 179
- Help Toolkit 542
- interfaces 507
- modifications, documenting 480, 481
- NAVreport 244
- new C/AL routines, creating 469
- pages 179
- processing flow 438
- Role Center pages 441
- supporting material 565
- tables 67
- tools 483
- variable, naming 137

### **NAV, terminologies**

- complex data type 126
- constant 126
- data element 126
- data type 125
- fundamental (simple) data type 125
- variable 126

### **NAV 2009**

- about 14
- C/SIDE 22
- changes 19
- Client Add-in feature 530
- development tools, accessing 22
- ERP system 14
- function terminology 28
- NAV 2009C/SIDE RD tool 244
- NAV 2009Report Designer tools 244
- NAV 2009Role Tailored Client 259
- object types, defining 22
- object types, Page 22
- possible configuration 20
- prior versions 14

- terminology 25, 26
- user interface 29, 30

### **NAV 2009, as ERP system 14-19**

#### **NAV 2009, modifications**

- compatible Report Viewer 21
- designing 544
- functional area, creating 545
- Role Tailored Client (RTC) 21, 30
- two-tier versus three-tier 20
- web services 21

#### **NAV 2009 Object types**

- Codeunit 22, 58
- Dataport 22, 59
- defining 22
- Form 22
- MenuSuite 22, 59
- Page 22
- Report 22
- Table 22
- XMLports 22, 60

#### **NAV application functionality**

- data design sequence 549
- data tables, defining 548
- data validation, designing 549
- posting processes, designing 550
- support processes, designing 550
- user data access interface, defining 548

#### **NAV Application Server (NAS) 529**

#### **NAV Communication Component 528**

#### **NAV development**

- about 31
- ICAN application 31

#### **NAV enhancement project**

- creating 544

#### **Navigate page**

- about 193, 194
- feature 194
- using 194

#### **Navigate tool**

- about 493
- modifying 496, 497
- testing with 493-496

#### **NAV processing flow**

- about 438
- data, accessing 440
- data, preparing 439
- flow data, accessing 440

- initial setup 438
- Journal Batch, posting 438, 440
- Journal Batch, testing 440
- maintenance 438-441
- transaction entry 438
- transactions, entering 439, 440
- utilize 438
- validate 438

### **NAV Report Designer**

- C/SIDE RD tool 245
- flowchart 246
- Working Storage 246

### **negative testing 503**

### **new object, Object Designer**

- Codeunit Designer 317
- MenuSuite Designer 317-319
- Page Designer 314
- Report Designer 315
- Table Designer 313
- XMLport Designer 316

### **numeric data, fundamental data types**

- bigInteger 139
- binary 139
- boolean 139
- char 139
- decimal 138
- integer 138
- option 139

## **O**

### **object and field numbers**

- about 26, 27
- ranges 27

### **Object Designer**

- about 312, 313
- compiling 326-328
- designer navigation pointers 319, 320
- new object, starting 313
- objects, exporting 320, 321
- objects, importing 322, 323
- saving 326-328
- text objects 324

### **object numbering 324**

### **OLAP 438**

### **Online Analytical Processing. *See* OLAP operators, C/AL syntax**

- arithmetic operator 347
- boolean operator 348
- precedence of operator 349
- relational operator 348, 349

### **OUTPUT function. *See* INPUT functions**

## **P**

### **page**

- about 179, 180, 234
- bound 180
- changes, adding 181
- components 201, 202
- components, controls 201
- components, page trigger 201
- components, properties 202, 203
- Control triggers 228
- controls 180, 205
- designing 179
- dialog page 192
- Journal/Worksheet page 191
- List+ page 191
- List page 187
- naming 199, 200
- naming, Card page 199
- naming, Journal/ Worksheet page 199
- naming, List page 199
- Page Designer, accessing 200
- Page Parts 197
- plagiarism 235
- properties 203, 205
- request page 192, 300
- RTC 182
- tabular 187
- types 180, 187
- unbound 180
- User Experience (UX) Guidelines 234

### **page, properties**

- AutoSplitKey 204
- Caption 203
- CaptionML 203
- CardFormID 204
- DelayedInsert 204
- Description 204
- Editable 204

- ID 203
- MultipleNewLines 204
- Name 203
- PageType 204
- Permissions 204
- SourceTable 204
- SourceTableTemporary 205
- SourceTableView 204
- page, types**
  - card page 188
  - Confirmation (Dialog) page 192
  - Department page 195, 196
  - Document page 188
  - List pages 187
  - List pages, Customer list 187
  - Navigate page 193, 194
  - Request page 192
  - Role Center page 196
- Page 344 - Navigate 479**
- page controls**
  - about 205-209
  - blank slate approach 236, 237
  - Communication group 206
  - container 211
  - container, Fast Tabs 180
  - container, Groups 180
  - Departments buton 210
  - designing 238
  - experimenting with 235-239
  - fields 214, 215
  - General group control 206, 207
  - group 212
  - help, searching 236
  - Home button 210
  - inheritance 211
  - Invoicing group 206
  - Navigation Pane 210
  - Page Parts 224
  - testing 238
  - triggers 228, 229
  - types 211-224
  - using, in Card page 218-224
- Page Parts, page controls**
  - Card Part FactBox, creating 226-228
  - ChartPartID property 225

- FactBoxes 198
- PagePartID property 225
- PartType property 225
- ProviderID property 224
- SubFormLink property 224
- SubFormView property 224
- SystemPartID property 225
- typical properties 224
- using 224
- positive testing 503**
- posting reports 252, 253**
- programming, C/SIDE 334**
- project recommendations, customizing**
  - about 554
  - Cronus database testing 555
  - deliverables 558
  - one at a time 554
  - project, finishing 559
  - testing 555
  - testing, in production 556
  - testing database, using 556, 557
  - testing in production 556
  - testing techniques 558
- properties 26**

## Q

**QUIT function 401**

## R

**Rapid Implementation Methodology. *See* RIM**

**RDLC 247**

**Read-Only Tables**

- about 119
- virtual 119

**records 26**

**report components**

- overview 254
- Report Description 255
- report description 255

**report data flow**

- about 256-259
- Data items 257
- events 256

**Report Definition Language Client-side 247****Report Description, report components**

Data item Properties and Triggers 255

Report Properties 255

Report Properties and Triggers 255

**report elements**

about 259

Data Item properties 264, 265

data item properties 264-269

Data items 264

Data Item Sections 270, 271

Data item triggers 269, 270

inheritance 281

Report Properties 260

Report triggers 263

request page 300

RTC reports, creating via Classical Report

Wizard 271-279

run time formatting 280

**Report Properties**

BottomMargin 261

Caption 260

CaptionML 260

DeviceFontName 261

HorzGrid 261

ID 260

LeftMargin 261

Name 260

Orientation 261

PaperSize 261

PaperSourceFirstPage 261

Permissions 261

ProcessingOnly 261

RightMargin 261

ShowPrintStatus 260

TopMargin 261

TransactionType 261

UseReqForm 261

UseSystemPrinter 261

VertGrid 261

**reports**

about 243, 244

components 254

creating 305

creating, from scratch 305

creative report plagiarism 306

data flow 256-259

document report 249, 253

elements 259

list reports 248, 253

naming 253

NAV reports, look and feel 247

posting report 253

posting reports 252

processing only reports 304

test reports 251, 253

transaction report 250, 253

types 248

types, summary 253

**Report triggers**

Documentation() 263

OnCreateHyperlink() 263

OnHyperlink() 263

OnInitReport() 263

OnPostReport() 263

OnPreReport() 263

**Request page**

about 192, 300

C/AL programming 302-304

FastTabs 301

**RIM 566****Role Center page**

about 441

Activities part 197

Action Menus 455-461

Department, MenuSuite levels 462

Departments button 461

for order processor 197

MenuSuite structure 462

Navigation Pane 455-461

page part 452-454

properties 444

purpose 196

structure 443

system part 451

**Role Center page, structure**

Activities Page 445

Cue Group actions 449, 450

Cue Groups 445

Cues 446



Cue source table 446-449  
representation 443

**Role Tailored Client.** *See* **RTC**

## **RTC**

about 13, 181  
Action Pane 186  
Actions button 184  
address bar 183  
command bar 183-186  
FactBox Pane 187  
Filter Pane 186  
Navigation Pane 186  
Related Information button 185  
Reports button 185  
title bar 182  
travel button 182

## **RTC reports**

changes, making 288-294  
creating, via Classical Report Wizard 271-279  
creatings, ways 281  
existing report, modifying 282, 283  
experimentation 279  
inheritance 281  
Page Header fields 298-300  
report capabilities 295-298  
Report Items 286-288  
run time formatting 280

**RUN function** 412

**RUNMODAL function** 412

## **S**

**SETCURRENTKEY function** 355

**SETRANGE function** 356

**SIFT** 77, 78, 103

**simple system administration** 561

**SQL Server Reporting Service.** *See* **SSRS**

**SSRS** 247

**string data, fundamental data types**

code 139  
text 139

**STRMENU function** 354

**SumIndex Fields** 548

**SumIndexField Technology.** *See* **SIFT supporting material**

reference material 566-568

RIM 566

Sure Step method 565

**Sure Step method** 565

**System table, Content-modifiable tables**

Chart 118

Company 118

Database Key Groups 118

Member Of 117

Permission 118

Profile 118

system internals 118

user 117

User Personalization 118

User Role 118

Web Service 118

Windows Access Control 118

Windows Login 118

## **T**

**Table 330**

about 479

currency conversion functions 479

**Table 370** 479

**Table Elements, Visual Studio Report**

**Designer (VS RD)**

Body Section 285

Element Properties 285

Page Footer 286

Page Header 285

Property Pages Button 286

## **tables**

about 68

activity-tracking tables, adding 97

advantages 68

Content-modifiable tables 117

creating 82-86

Field Groups 79-81

keys 74, 76

modifying 82-86, 93

naming conventions 69

- numbering 69
- overview 67, 68
- properties 70, 72
- Read-Only Tables 119
- reference tables, adding 98-102
- SIFT 77
- SumIndexFields 77
- TableRelation property, assigning 87
- Totally modifiable tables 107
- triggers 72, 73
- types 107
- Wholly modifiable tables 107
- tables, properties**
  - Caption 71
  - CaptionML 71
  - DataCaptionFields 71
  - DataPerCompany 71
  - Description 71
  - DrillDownFormID 71
  - ID 71
  - LinkedObject 72
  - LookupFormID 71
  - Name 71
  - PasteIsValid 72
  - Permissions 71
- tabular 187**
- TagType as Attribute 524**
- TagType as Element 524**
- terminology, NAV**
  - controls 26
  - database 25
  - field numbers 26
  - fields 26
  - license 28
  - object numbers 26
  - properties 26
  - records 26
  - triggers 26
  - work date 27
- terminology, NAV 2009**
  - controls 26
  - database 25
  - field numbers 27
  - fields 26
  - license 28
  - object numbers 27
  - properties 26
  - records 26
  - triggers 26
  - work date 27
- test-driven development**
  - feature goals 503
  - OnAfterTestRun 502
  - OnBeforeTestRun 502
  - TestMethodType property 503, 504
- testing**
  - about 555
  - Cronus database testing 555
  - database, using 556
  - deliverables 558
  - in production 556
  - production database, problems 557
  - project, finishing 559
  - techniques 558
  - testing database, using 556, 557
- test report**
  - about 251, 253
  - General Journal batch example 252
- tools, NAV**
  - Client Monitor 499
  - code analysis tools 483
  - Code Coverage tool 498
  - Debugger 497
  - debugging tools 483
  - Developer's Toolkit 483
  - dialog function debugging 500
  - exported text code, working with 491, 492
  - Navigate 493
  - test-driven development 502
- Totally modifiable tables**
  - about 107
  - ledger 110
  - master tables 107
  - posted document 114
  - reference 112
  - setup 116
  - template 109
  - temporary 117
- transaction report 250, 253**

## **triggers**

- about 26
- documentation triggers 26, 72
- functions triggers 26
- function triggers 26
- OnDelete() trigger 73
- OnInsert() trigger 73
- OnModify() trigger 73
- OnRename() trigger 73

## **U**

**Uniform Resource Name.** *See* URN

### **upgrading plan**

- about 559
- advantages 560
- benefits 560
- executables, upgrading 563
- full upgrade 564
- good documentation 561
- low impact coding 562
- process 563
- tips 560
- tips, careful naming 561
- tips, good documentation 561
- tips, low impact coding 562

### **URN 512**

### **User Experience (UX) Guidelines**

- about 234
- topics 234

**user interface 29, 30**

## **V**

**ValidatePostCode function 366**

### **validation utility functions**

- about 384
- CURRENTDATETIME 388
- Direction value 388
- FIELDERROR 385, 386
- precision values 387
- ROUND 387
- TESTFIELD 384
- TIME 388
- TODAY 388

VALIDATE 386

Work Date 389

### **variables**

- about 329
- arrays 332
- global identifiers 329
- initialization 333
- local variables 330
- system defined variables 334
- temporary tables 331, 332
- working storage variables 331

### **Visual Studio Report Designer (VS RD)**

- about 255
- Data Sources 285
- layout screen 284, 285
- Report Layout 255
- Table Elements 285
- Toolbox Tab 284

## **W**

### **web services**

- about 534
- benefits 534, 535
- exposing 536
- factors 535, 536
- published object, determining 538-542
- publishing 537, 538
- use 535

**Web Services Description.** *See* WSDL

### **Wholly modifiable tables**

- Journal table 108, 109
- Ledger table 110-112
- master table 107, 108
- Posted Document type 114, 115
- Reference table 112, 113
- Register table 114
- Setup table 116
- Template table 109, 110
- Temporary table 117

**wild cards 137**

**Windows Presentation Foundation.** *See*

WPF

**work date 27, 28**

**Working Storage fields 255**

**working storage variables**

- arrays 332
- initialization 333
- system defined variables 334
- temporaray tables 331

**WPF 179****WSDL 539****X****XML port properties**

- about 510
- Caption 511
- CaptionML 511
- DataItemSeparator 513
- DefaultFieldsValidation 511
- DefaultNamespace 512
- Direction 511
- Encoding 512
- FieldDelimiter 513
- FieldSeparator 513
- FileName 513

Format 512

Format/Evaluate 512

ID 511

InlineSchema 512

Name 511

Permissions 513

RecordSeparator 513

TransactionType 513

UseDefaultNamespace 512

UseLax 512

UseRequestForm 513

XMLVersionNo 512

**XMLports**

about 60, 508, 509

components 510

debugging 508

formats 508

**Z****ZUP file 92**





Thank you for buying  
**Programming Microsoft®  
Dynamics™ NAV 2009**

## About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

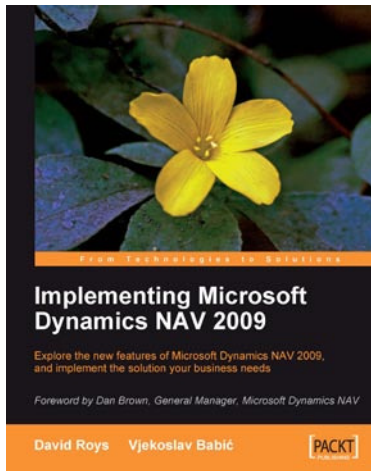
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

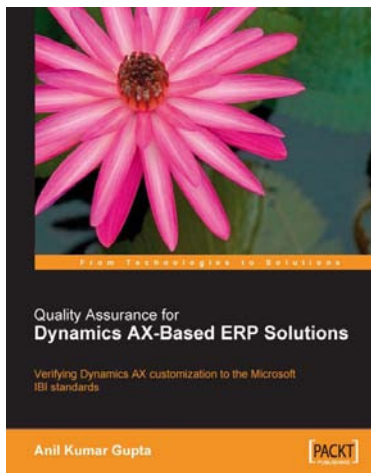


## Implementing Microsoft Dynamics NAV 2009

ISBN: 978-1-847195-82-1      Paperback: 552 pages

Explore the new features of Microsoft Dynamics NAV 2009, and implement the solution your business needs

1. First book to show you how to implement Microsoft Dynamics NAV 2009 in your business
2. Meet the new features in Dynamics NAV 2009 that give your business the flexibility to adapt to new opportunities and growth
3. Easy-to-read style, packed with hard-won practical advice
3. Real-world examples with step-by-step explanations



## Quality Assurance for Dynamics AX-Based ERP Solutions

ISBN: 978-1-847192-91-2      Paperback: 168 pages

Verifying Dynamics AX customization to the Microsoft IBI Standards

1. Learn rapidly how to test Dynamics AX applications
2. Verify Industry Builder Initiative (IBI) compliance of your ERP software
3. Readymade testing templates
4. Code, design, and test a quality Dynamics AX-based ERP solution
5. Customization best practices backed by theory

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



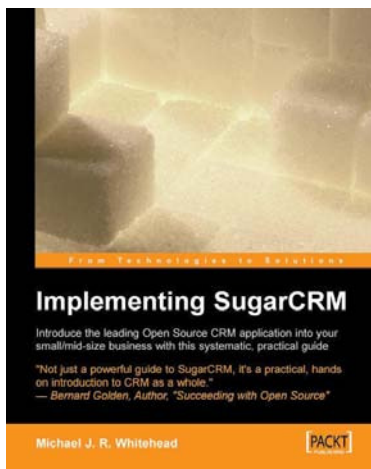
## SAP Business ONE Implementation

ISBN: 978-1-847196-38-5

Paperback: 320 pages

Bring the power of SAP Enterprise Resource Planning to your small-midsize business

1. Get SAP B1 up and running quickly, optimize your business, inventory, and manage your warehouse
2. Understand how to run reports and take advantage of real-time information
3. Complete an express implementation from start to finish
4. Real-world examples with step-by-step explanations



## Implementing SugarCRM

ISBN: 978-1-904811-68-8

Paperback: 328 pages

A step-by-step guide to using this powerful Open Source application in your business.

1. Your complete guide to SugarCRM implementation - assess your needs, install the software, start using it, train users, integrate with existing systems
2. Covers both the free and commercial versions of SugarCRM - get maximum benefit from the free version before paying for add ons

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles