

# Programming Microsoft Dynamics NAV

**Fifth Edition**

Hone your skills and increase your productivity when programming in Microsoft Dynamics NAV 2017



By Mark Brummel, David A. Studebaker,  
Christopher D. Studebaker

**Packt**

[www.packt.com](http://www.packt.com)

# **Programming Microsoft Dynamics NAV**

*Fifth Edition*

Hone your skills and increase your productivity when  
programming in Microsoft Dynamics NAV 2017

**Mark Brummel**

**David A. Studebaker**

**Christopher D. Studebaker**



BIRMINGHAM - MUMBAI

# **Programming Microsoft Dynamics NAV**

## *Fifth Edition*

Copyright © 2017 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(s), 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 Edition: October 2007

Second Edition: November 2009

Third Edition: February 2013

Fourth Edition: July 2015

Fifth published: April 2017

Production reference: 1190417

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78646-819-2

[www.packtpub.com](http://www.packtpub.com)

# Credits

**Authors**                    **Copy Editor**

Mark Brummel                Zainab Bootwala  
David Studebaker  
Christopher Studebaker

**Reviewer**                    **Project Coordinator**

Alex Chow                    Prajakta Naik

**Commissioning Editor**                **Proofreader**

Aaron Lazar                Safis Editing

**Acquisition Editor**                **Indexer**

Nitin Dasan                Mariammal Chettiar

**Content Development Editor**                **Production Coordinator**

Siddhi Chavan                Nilesh Mohite

**Technical Editor**

Dhiraj Chandanshive

# Foreword

In this age where almost any piece of information can be found on the internet, it might seem obsolete to come around with the book that's in front of you. Hundreds of pages of information on what a developer would, and I personally think should, like to know about developing in Dynamics NAV, dozens of useful technical tips and tricks, best practices, and a load of references to more, a great asset to any Dynamics NAV development team, and even more. And this is what makes this book something far beyond all the information out there. This book integrates technical Dynamics NAV with functional Dynamics NAV. On contrary to many other technical platforms, Dynamics NAV is first and foremost an ERP platform serving thousands of customers around the world. Any developer with an ambition to start programming Microsoft Dynamics NAV cannot but become an expert in the functional side of it.

Being the first ever author to have written an independent book on how to program Dynamics NAV, Dave Studebaker became a valuable resource to many and a trendsetter in our community in the late days of his professional live. With the new rhythm of major releases by Microsoft, Dave, and his co-author, and son, Chris, managed to keep his sibling evolve with the subject it covered so eloquently. Now, a decade after the first version of this book, with NAV on the verge of a major development experience change, Dave has called upon some helping hands from the other side of the Atlantic. Both Mark Brummel and I were honored to be included in this project. Thanks for having us in, Dave and Chris.

And thank you reader for buying this book. May it be as valuable to you as was to many before you.

## **Luc van Vugt**

Microsoft MVP and Co-Founder of NAV-Skills

# About the Authors

**Mark Brummel** is a teacher and evangelist for Microsoft Dynamics NAV, focused in helping partners and end users of the product.

With NAV-Skills.com he evangelizes and documents the "NAV way". This is a combination of architectural principles and design best practices formalized in a workshop called Master Class for Microsoft Dynamics NAV Application Architecture and Design Patterns. The methodology helps in creating solutions that are easy to upgrade, recognizable for users, and maintainable outside of the ecosystem of their creators. All three elements apply to the original Navision product that shipped in 1995 and are extracted, updated, and documented in this methodology. In September 2015, his new book *Learning Dynamics NAV Patterns* was published which is about this methodology. He also organizes hands on workshops together with a group of MVPs and MCTs all across the globe.

He started in 1997 as an end user and worked eight years for NAV partners after that. 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. Coaching colleagues and troubleshooting complex problems are his passion and part of his day to day work. His first book, *Dynamics NAV 2009 Application Design*, was published in 2010 and updated to *Dynamics NAV 2013 Application Design* when a new release became available.

Many end users of Microsoft Dynamics NAV struggle with questions about how to upgrade their two-tier solution to a three-tier solution. Mark can help you answer these questions and plot a roadmap to the future, retaining the investment in the solution.

When Microsoft introduced the three-tier architecture in 2009 it meant a major shift for experienced NAV developers and consultant. Mark has trained most of them in The Netherlands and Belgium.

In 2015 Mark got the NAVUG All-Star award on behalf of all the members. He was amongst the first to receive this prestigious award.

In 2010 he started a think tank called Partner Ready Software together with four other Dynamics NAV Experts. Partner Ready Software brings fresh ideas of designing applications in NAV and creates awareness for applying design patterns in creating repeatable solutions.

Mark is an associate in the Liberty Grove Software network, member of the NAVUG advisory board, co-founder of the Dutch Dynamics Community, vice-president of the Association of Dynamics Professionals and advisor for Dynamics HUB, a special project and performance tuning of the Dynamics NAV product on SQL Server. As a unique specialist, he has done breakthrough research in improving the performance of Dynamics NAV on SQL Server.

Mark maintains a blog on <https://nav-skills.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. He also publishes articles on Pulse for LinkedIn.

Since 2006 Mark has been rewarded by Microsoft with the Most Valuable Professional award for his contribution to the online and offline communities. He received the award eleven times.

Mark is a father of four, married, and lives in a small town in The Netherlands.

He is an author of the books, *Microsoft Dynamics NAV 2009 Application Design*, *Microsoft Dynamics 2013 Application Design*, and *Learning Dynamics NAV Patterns*, all for Packt.

**David Studebaker** has been a software consulting entrepreneur and manager most of his career while always maintaining a significant role as an application developer. David has been designing, developing, and teaching software since 1962. He has been a founding partner in five software service firms, most recently Studebaker Technology and Liberty Grove Software. Among his special achievements was the design and development of the very first production SPOOL system, a 1967 AT&T / IBM joint project.

David has been writing for publication since his college days. His published writings include a decade of technical reviews for the ACM's Computing Reviews and a variety of articles and reference material on shop floor data collection. David is the author of four other Packt books on programming in Dynamics NAV C/AL, two of which were co-authored with Christopher Studebaker.

David has a BS in Mechanical Engineering from Purdue University and an MBA from the University of Chicago. He is a life member of the Association for Computing Machinery and was a founding officer of two ACM chapters.

**Christopher Studebaker** has worked as a NAV Developer / Implementer since 1999. He has experience designing, developing, implementing, and selling in the NAV and SQL Server environments, specializing in retail, manufacturing, job shop, and distribution implementations, mostly in high user-count, high data volume applications.

Chris has worked on many NAV implementations with integrations to external databases and third-party add-on products. Some special applications have included high volume order entry, pick-to-light systems, point of sale, procurement analysis, and web front ends.

Chris acts in a consulting and training role for customers and for peer NAV professionals. He provides training both in informal and classroom situations, often developing custom course material to support courses tailored to specific student group needs. Courses have included various NAV functional and development areas. Chris was a co-author of two previous books on programming in NAV C/AL.

Before becoming a certified NAV developer, Chris was a certified environmental consultant. His duties included regulatory reporting, data analysis, project management, and subcontractor oversight in addition to managing projects for hazardous material management and abatement.

Chris is a Microsoft Certified IT Professional, SQL database developer, as well as a Microsoft Certified Business Solutions Professional in NAV Development and NAV Installation and Configuration. He has a Bachelor of Science degree from Northern Illinois University and has done graduate work at Denmark Technical University.

Chris was a co-author of the Packt books, *Programming Microsoft Dynamics NAV 2013* and *Programming Microsoft Dynamics NAV 2015*.

# About the Reviewer

**Alex Chow** has been working with Microsoft Dynamics NAV, formerly Navision, since 1999. Over the years, Alex has conducted hundreds of implementations across multiple industries. His customers range from \$2 million a year small enterprises to \$500 million a year multinational corporations.

Over the course of his Dynamics NAV career, he has often been designated as the primary person responsible for the success and failure of a Dynamics NAV implementation. The fact that Alex is still in the Dynamics NAV business means that he's been pretty lucky so far. His extensive career in the Dynamics NAV business is evidence of his success rate and expertise.

With a background in implementing all functions and modules both in and outside of Microsoft Dynamics NAV, Alex has encountered and resolved the most practical to the most complex requirements and business rules. Through these experiences, he has learned that sometimes you have to be a little crazy to have a competitive edge.

Believing that sharing these experience and knowledge would benefit the Dynamics NAV community, Alex writes about his journey at [www.dynamicsnavconsultant.com](http://www.dynamicsnavconsultant.com). He is also the founder of AP Commerce, Inc. ([www.apcommerce.com](http://www.apcommerce.com)) in 2005, a full service Dynamics NAV service center. In addition, Alex has written two books about Dynamics NAV titled *Getting Started with Dynamics NAV 2013 Application Development* and *Implementing Microsoft Dynamics NAV - Third Edition*, both by Packt.

Alex lives in Southern California with his beautiful wife and two lovely daughters. He considers himself the luckiest man in the world.

# [www.PacktPub.com](http://www.PacktPub.com)

For support files and downloads related to your book, please visit [www.PacktPub.com](http://www.PacktPub.com).

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

# Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1786468190>.

If you'd like to join our team of regular reviewers, you can e-mail us at [customerreviews@packtpub.com](mailto:customerreviews@packtpub.com). We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

# Acknowledgments

## Mark Brummel

This book is the result of working with Microsoft Dynamics NAV® for almost 20 years and having had many discussions, debates, and even arguments with other people who are as passionate about the product as I am.

Being an early adopter of the software, I was very fortunate to be able to make many mistakes during my career and learn from them each time. Fundamental questions such as variable naming, user interface, and where to write code were all discussed many times and changed over the years.

Like many developers, I feel like a creative artist that needs to put his signature on his work. Unfortunately, this does not go well when writing solutions for Microsoft Dynamics NAV. When adopting the style of the base application, you will find that collaboration becomes easier and people will understand your efforts better than when you create your own style.

During the years I worked with the product, I've come to appreciate the structure of the base application. At a certain point in my career, I made an assumption that many of you might recognize. I started to treat what was done in the core product as "always correct".

In the past ten years, ever since I received my first MVP award, I got closer to the development team and was fascinated by the way the product was created and how issues are prioritized. I heard many stories from people who worked on the product in the early 90ies and started to realize that the core product is written by developers just like me and that many of the core components were acquired from partners who had their own development styles.

Over the years, I felt the core NAV product became 'polluted' and code was written in a less structured way, both by Microsoft and by NAV partners. Between 2006 and 2010, my core business was performance troubleshooting and upgrades. The code I saw in that period motivated me to write my first book about Application Design. This was my first effort to document the way Microsoft Dynamics NAV is designed and how application areas work together.

Special thanks go to two very important mentors I've had in my professional career –David Studebaker and Michael Nielsen. Dave and his wife Karen are very special friends and their knowledge and wisdom have been very important. Very often when I discuss an idea with Dave, he says things like, "Yes we did that in 1973 with Cobol." Michael has been my guardian angel at Microsoft for a decade, and now my colleague at ForNAV and personal friend.

I hope this book will inspire you, as it did me, to write software in a generic style that others will recognize and find easy to work with.

I'd also like to thank the reviewers as mentioned in the book but also Luc van Vugt who unofficially helped me with this book.

Last, but definitely not least, I want to thank my wife Dionel and my kids Josefien, Wesley, Saskia, and Daan. Thank you for allowing me to spend time writing these books and travel to the events where I can talk to others about this.

### **David Studebaker**

I offer unbounded appreciation to Karen, my spouse and partner in both life and work, for your unflagging love, support, patience and encouragement in all ways.

Thank you to my co-authors, Mark Brummel and Christopher Studebaker, without whom this book would not have been possible. I am very fortunate to get to work with such knowledgeable experts as you, who are each special to me in your own way.

Thank you to Rebecca and Christopher, my children of whom I am very proud for your own achievements, for your love and support, and for your parenting of my wonderful grandchildren Alec, CeCe and Cole, who are the future.

Special thanks to Michael Nielsen of Microsoft for your wholehearted support of this and my previous four Programming NAV books. Many thanks to Mark Brummel, who knows more about NAV than almost anyone (except maybe Michael), for generously sharing your knowledge.

Thank you to all the people at Microsoft and Packt, as well as our technical reviewers and others, who assisted us with their contributions and advice (especially Luc van Vugt).

Family and friends are what make life worthwhile. All my life, I have had the benefit of inspiration, affection, and help from so many people. Thank you all.

### **Christopher Studebaker**

First and foremost, I would like to thank my parents, Karen and David Studebaker, for giving me the opportunity to start in the NAV world and allowing me the room to grow on my own. Of course, I could not have participated in this book if it weren't for my wife, Beth. Having worked within the NAV community for the past decade, I have worked with many wonderful people, most notably, my parents (of course), Betty Cronin, Kathy Nohr, Tommy Madsen, Susanne Priess, David Podjasek, Joy Bensur , Diane Beck, Chris Pashby, and Anthony Fairclough. Without them, I would not have been the NAV professional I am today.

# Table of Contents

<b>Preface</b>	1
<b>Chapter 1: Introduction to NAV 2017</b>	13
<b>NAV 2017 - An ERP system</b>	14
Financial management	16
Manufacturing	16
Supply chain management	17
Business Intelligence and reporting	18
Artificial Intelligence	19
Relationship Management	19
Human resource management	20
Project management	20
<b>A developer's overview of NAV 2017</b>	20
NAV object types	21
The C/SIDE Integrated Development Environment	21
Object Designer tool icons	22
C/AL programming language	23
NAV object and system elements	25
NAV functional terminology	29
User Interface	29
<b>Hands-on development in NAV 2017</b>	31
NAV 2017 development exercise scenario	31
Getting started with application design	32
Application tables	33
Designing a simple table	33
Creating a simple table	35
Pages	37
Standard elements of pages	38
List pages	38
Card pages	39
Document pages	40
Journal/Worksheet pages	42
Creating a List page	43
Creating a Card page	48
Creating some sample data	53
Creating a list report	54

<b>Other NAV object types</b>	66
Codeunits	67
Queries	68
MenuSuites	68
XMLports	68
<b>Development backups and documentation</b>	69
<b>Review questions</b>	70
<b>Summary</b>	73
<b>Chapter 2: Tables</b>	74
<b>An overview of tables</b>	74
Components of a table	76
Naming tables	77
Table numbering	78
Table properties	78
Table triggers	81
Keys	84
SumIndexFields	87
Field Groups	89
Bricks	93
<b>Enhancing our sample application</b>	94
Creating and modifying tables	95
Assigning a table relation property	101
Assigning an InitValue property	104
Adding a few activity-tracking tables	105
New tables for our WDTU project	106
New list pages for our WDTU project	109
Keys, SumIndexFields, and TableRelations in our examples	109
Secondary keys and SumIndexFields	110
Table Relations	111
Modifying an original NAV table	113
Version List documentation	114
<b>Types of table</b>	115
Fully Modifiable tables	115
Master Data	116
Journal	117
Template	118
Entry tables	119
Subsidiary (Supplementary) tables	122
Register	125
Posted Document	126
Singleton	127

Temporary	128
Content Modifiable tables	129
System table	129
Read-Only tables	130
Virtual	131
<b>Review questions</b>	131
<b>Summary</b>	134
<b>Chapter 3: Data Types and Fields</b>	135
<b>Basic definitions used in NAV</b>	136
<b>Fields</b>	136
Field properties	137
Field triggers	144
Field events	145
Data structure examples	145
Field numbering	145
Field and variable naming	146
<b>Data types</b>	148
Fundamental data types	148
Numeric data	148
String data	150
Date/Time data	151
Complex data types	152
Data structure	153
Objects	153
Automation	154
Input/Output	154
DateFormula	154
References and other data types	161
Data type usage	162
<b>FieldClass property options</b>	164
FieldClass - Normal	164
FieldClass - FlowField	164
FieldClass - FlowFilter	167
FlowFields and a FlowFilter for our application	170
<b>Filtering</b>	176
Experimenting with filters	177
Accessing filter controls	183
Development Environment filter access	184
Role Tailored Client filter access	185
<b>Review questions</b>	187
<b>Summary</b>	191

<b>Chapter 4: Pages - the Interactive Interface</b>	192
<b>Page Design and Structure Overview</b>	193
Page Design guidelines	194
NAV 2017 Page structure	195
<b>Types of pages</b>	198
Role Center page	199
List page	200
Card page	201
Document page	202
FastTabs	203
ListPlus page	203
Worksheet (Journal) page	205
Confirmation Dialog page	206
Standard Dialog page	206
Navigate page	206
Navigate function	207
Special pages	208
Request page	208
Departments page	210
<b>Page parts</b>	211
FactBox Area	211
CardParts and ListParts	211
<b>Charts</b>	213
Chart Part	213
Chart Control Add-In	214
<b>Page names</b>	214
<b>Page Designer</b>	215
New Page wizard	216
<b>Page Components</b>	220
Page Triggers	221
Page properties	222
Page Preview Tool	226
Inheritance	227
<b>WDTU Page Enhancement - part 1</b>	227
<b>Page Controls</b>	234
Control types	237
Container controls	237
Group controls	237
Field controls	242
Page Part controls	245
Page Control triggers	249

Bound and Unbound Pages	249
<b>WDTU Page Enhancement - part 2</b>	
<b>Page Actions</b>	250
Page Action types and subtypes	253
Action Groups	256
Navigation Pane Button actions	257
Actions Summary	261
<b>Learning more</b>	262
Patterns and creative plagiarism	263
Experimenting on your own	263
Experimentation	264
<b>Review questions</b>	267
<b>Summary</b>	270
<b>Chapter 5: Queries and Reports</b>	271
<b>Queries</b>	272
Building a simple Query	273
Query and Query Component properties	278
Query properties	279
DataItem properties	280
Column properties	281
<b>Reports</b>	282
What is a report?	282
Four NAV report designers	284
NAV report types	287
Report types summarized	291
Report naming	292
<b>Report components - overview</b>	292
Report Structure	292
Report Data overview	293
Report Layout overview	295
<b>Report data flow</b>	296
Report components - detail	299
C/SIDE Report Properties	300
Visual Studio - Report Properties	303
Report triggers	306
Request Page Properties	306
Request Page Triggers	307
DataItem properties	309
DataItem triggers	311
<b>Creating a Report in NAV 2017</b>	311

Learn by experimentation	311
Report building - Phase 1	312
Report building - Phase 2	316
Report building - Phase 3	322
Modifying an existing report with Report Designer or Word	328
Runtime rendering	332
Inheritance	333
Interactive report capabilities	333
Interactive sorting	333
Interactive Visible / Not Visible	335
Request Page	336
Add a Request Page option	337
Processing-Only reports	340
Creative report plagiarism and Patterns	340
<b>Review questions</b>	341
<b>Summary</b>	344
<b>Chapter 6: Introduction to C/SIDE and C/AL</b>	345
<hr/>	
<b>Understanding C/SIDE</b>	346
Object Designer	347
Starting a new object	348
Accessing the Table Designer screen	348
Accessing the Page Designer	349
Accessing the Report Dataset Designer	350
Accessing the Codeunit Designer	350
Query Designer	351
XMLport Designer	351
MenuSuite Designer	352
Object Designer Navigation	356
Exporting objects	357
Importing objects	358
Import Table object changes	361
Text objects	362
Shipping changes as an extension	363
Some useful practices	363
Changing data definitions	364
Saving and compiling	365
Some C/AL naming conventions	366
Variables	367
C/AL Globals	367
C/AL Locals	368
Special working storage variables	369
C/SIDE programming	373
Non-modifiable functions	374
Modifiable functions	374

Custom functions	376
Creating a function	377
<b>C/AL syntax</b>	385
Assignment and punctuation	385
Expressions	387
Operators	387
Arithmetic operators and functions	390
Boolean operators	391
Relational operators and functions	391
Precedence of operators	392
Frequently used C/AL functions	393
MESSAGE function	394
ERROR function	395
CONFIRM function	397
STRMENU function	398
Record functions	400
SETCURRENTKEY function	400
SETRANGE function	400
SETFILTER function	401
GET function	402
FIND Functions	402
FIND ([Which]) options and the SQL Server alternates	404
Conditional statements	406
BEGIN-END compound statement	406
IF-THEN-ELSE statement	407
Indenting code	408
<b>Some simple coding modifications</b>	408
Adding field validation to a table	408
Adding code to a report	415
Lay out the new Report Heading	416
Saving and testing	417
Lookup related table data	418
Laying out the new report Body	419
Saving and testing	421
Handling user entered report options	421
Defining the Request Page	424
Finishing the processing code	424
Testing the completed report	426
Output to Excel	427
<b>Review questions</b>	427
<b>Summary</b>	431
<b>Chapter 7: Intermediate C/AL</b>	432
<b>C/AL Symbol Menu</b>	433

<b>Internal documentation</b>	436
<b>Source code management</b>	440
<b>Validation functions</b>	440
TESTFIELD	440
FIELDERROR	441
INIT	443
VALIDATE	443
<b>Date and Time functions</b>	444
TODAY, TIME, and CURRENTDATETIME	444
WORKDATE	445
DATE2DMY function	446
DATE2DWY function	447
DMY2DATE and DWY2DATE functions	447
CALCDATE	448
<b>Data conversion and formatting functions</b>	449
ROUND function	449
FORMAT function	451
EVALUATE function	452
<b>FlowField and SumIndexField functions</b>	452
CALCFIELDS function	453
SETAUTOCALCFIELDS function	454
CALCSUMS function	455
CALCFIELDS and CALCSUMS comparison	456
<b>Flow control</b>	456
REPEAT-UNTIL	457
WHILE-DO	457
FOR-TO or FOR-DOWNTO	458
CASE-ELSE statement	459
WITH-DO statement	460
QUIT, BREAK, EXIT, and SKIP	462
QUIT function	462
BREAK function	462
EXIT function	463
SKIP function	463
<b>Input and Output functions</b>	463
NEXT function with FIND or FINDSET	464
INSERT function	465
MODIFY function	465
Rec and xRec	466
DELETE function	466

MODIFYALL function	467
DELETEALL function	467
<b>Filtering</b>	468
SETFILTER function	469
COPYFILTER and COPYFILTERS functions	470
GETFILTER and GETFILTERS functions	470
FILTERGROUP function	471
MARK function	472
CLEARMARKS function	472
MARKEDONLY function	472
RESET function	473
<b>InterObject communication</b>	473
Communication through data	473
Communication through function parameters	473
Communication via object calls	474
<b>Enhancing the WDTU application</b>	475
Modify table fields	476
Adding validation logic	480
Playlist Header validations	480
Creating the Playlist Subpage	482
Playlist Line validations	487
Creating a function for our Factbox	491
Creating a Factbox Page	495
<b>Review questions</b>	502
<b>Summary</b>	504
<b>Chapter 8: Advanced NAV Development Tools</b>	505
<hr/>	
<b>NAV process flow</b>	506
Initial Setup and Data Preparation	508
Transaction entry	508
Testing and Posting the Journal batch	508
Utilizing and maintaining the data	509
Data maintenance	510
<b>Role Center pages</b>	510
Role Center structure	511
Role Center activities page	514
Cue Groups and Cues	515
Cue source table	516
Cue Group Actions	519
System Part	520
Page Parts	522

Page Parts Not Visible	522
Page Part Charts	523
Page Parts for User Data	525
<b>Navigation Pane and Action Menus</b>	526
Action Designer	528
Creating a WDTU Role Center Ribbon	531
Action Groups / Ribbon Categories	532
Configuration/Personalization	534
Navigation Pane	537
Navigation Home Button	537
Navigation Departments Button	539
Other Navigation Buttons	540
<b>XMLports</b>	540
<b>XMLport components</b>	542
XMLport properties	542
XMLport triggers	546
XMLport data lines	546
The XMLport line properties	547
SourceType as Text	548
SourceType as Table	549
SourceType as Field	551
Element or Attribute	552
NodeType of Element	552
NodeType of Attribute	552
XMLport line triggers	552
DataType as Text	553
DataType as Table	554
DataType as Field	554
XMLport Request Page	554
<b>Web services</b>	555
Exposing a web service	556
Publishing a web service	558
Enabling web services	558
Determining what was published	559
XMLport - Web Services Integration example for WDTU	561
<b>Review questions</b>	569
<b>Summary</b>	571
<b>Chapter 9: Successful Conclusions</b>	572
<b>Creating new C/AL routines</b>	573
Callable functions	574
Codeunit 358 - DateFilterCalc	574
Codeunit 359 - Period Form Management	576
FindDate function	576
NextDate function	577

CreatePeriodFormat function	577
Codeunit 365 - Format Address	578
Codeunit 396 - NoSeriesManagement	579
Function models to review and use	581
Management codeunits	582
<b>Multi-language system</b>	582
<b>Multi-currency system</b>	583
<b>Navigate</b>	584
Modifying for Navigate	587
<b>Debugging in NAV 2017</b>	588
Text Exports of Objects	589
Dialog function debugging techniques	591
Debugging with MESSAGE and CONFIRM	591
Debugging with DIALOG	592
Debugging with text output	593
Debugging with ERROR	593
The NAV 2017 Debugger	594
Activating the Debugger	595
Attaching the Debugger to a Session	597
Creating Break Events	598
The Debugger window	600
Ribbon Actions:	600
Changing code while debugging	603
<b>C/SIDE Test-Driven Development</b>	603
<b>Other interfaces</b>	606
Automation Controller	607
Linked Data Sources	607
<b>NAV Application Server</b>	608
<b>Client Add-ins</b>	608
Client Add-in construction	609
WDTU Client Add-in	611
Client Add-in comments	629
<b>Creating an Extension</b>	629
Table Changes	629
Page Changes	629
Events	630
Creating a WDTU extension	630
Step 1 - Load PowerShell	631
Step 2 - Create Delta files	631
Step 3 - Manifest XML file	631
Remembering the App ID	632
Step 4 - Create the NAVx package	632
Installing the Extension	633

Publishing an Extension	633
Verification	633
Extension installation and setup	634
<b>Customizing Help</b>	636
<b>NAV development projects - general guidance</b>	637
Knowledge is key	637
Data-focused design	637
Defining the required data views	638
Designing the data tables	638
Designing the user data access interface	639
Designing the data validation	639
Data design review and revision	640
Designing the Posting processes	640
Designing the supporting processes	640
Double-check everything	641
<b>Design for efficiency</b>	641
Disk I/O	642
Locking	642
<b>Updating and Upgrading</b>	644
<b>Design for updating</b>	644
Customization project recommendations	645
One change at a time	646
Testing	646
Database testing approaches	647
Testing in production	647
Using a testing database	648
Testing techniques	649
Deliverables	650
Finishing the project	651
<b>Plan for upgrading</b>	651
Benefits of upgrading	652
Coding considerations	652
Good documentation	653
Low-impact coding	654
<b>Supporting material</b>	654
<b>Review questions</b>	655
<b>Summary</b>	658
<b>Index</b>	659

# Preface

Welcome to the worldwide community of Microsoft Dynamics NAV developers. This is a collegial environment populated by C/AL developers who readily and generously share their knowledge. There are formal and informal organizations of NAV-focused users, developers, and vendor firms scattered around the globe and active on the Web. Our community continues to grow and prosper--now including over 110,000 user companies worldwide.

The information in this book will help you shorten your learning curve on how to program for the NAV 2017 ERP system using the C/AL language, the C/SIDE integrated development environment, and their capabilities. We hope you enjoy working with NAV as much as we have.

## A brief history of NAV

Each new version of Microsoft Dynamics NAV is the result of inspiration and hard work along with some good fortune and expert technical investment over the last thirty years.

## 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 the following:

- 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. Also 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 V3, 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 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 quickly followed by a much improved V1.1.

In the next few years, Navision continued to be improved and enhanced. Major new functionalities, such as the following, 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-based 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. The Navision and Great Plains products all received a common rebranding as the Dynamics product line. Navision was renamed Dynamics NAV.

## Continuous enhancement

As early as 2003, research began with the Dynamics NAV development team, planning moves to further enhance NAV and 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 was a determination not to lose the strength and flexibility of the base product.

NAV 2009 was released in late 2008, NAV 2013 in late 2012, followed by NAV 2015 in late 2014. NAV2017, the version on which this book is based, was released in October 2016. The biggest hurdles to the new technologies have been cleared. A new user interface, the Role Tailored Client, was created as part of this renewal. NAV was tightly integrated with Microsoft's SQL Server and other Microsoft products, such as Office, Outlook, and SharePoint. Development is more integrated with Visual Studio and is more .NET compliant. The product is becoming more open and, at the same time, more sophisticated, supporting features such as Web Services access, Web and tablet clients, integration of third-party controls, RDLC and Word based reporting, and so on

Microsoft continues to invest in, enhanced and advanced NAV. More new capabilities and features are yet to come, continuing to build on the successes of the past. We all benefit.

## C/AL's Roots

One of the first questions asked by people new to C/AL is often "what other programming language is it like?" The best 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 analyses. 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 "everyman'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 as a software architect. 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 the following:

- 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, goals of the language and IDE designs were to :

- Allowing the developer to focus on design, not coding, but still allow flexibility
- Providing a syntax based on Pascal stripped of complexities, especially, relating to memory management
- Providing a limited set of predefined object types and reducing the complexity and learning curve

- Implementing database versioning for a consistent and reliable view of the database
- Making the developer and end user more at home by borrowing a large number of concepts from Office, Windows, Access, and other Microsoft products

Michael is now a co-founder and partner at ForNAV. Michael and his fellow team members are all about developing high quality reporting enhancements for Dynamics NAV with the goal of making working with reports easy. Another example of how, once part of the NAV community, most of us want to stay part of that community.

## What you should know

To get the maximum out of this book as a developer, you should have the following attributes:

- Be an experienced developer
- Know more than one programming language
- Have IDE experience
- Be knowledgeable about business applications
- Be good at self-directed study

If you have these attributes, this book will help you become productive with C/AL and NAV much more rapidly.

Even though this book is targeted first at developers, it is also designed to be useful to executives, consultants, managers, business owners, and others who want to learn about the development technology and operational capabilities of Dynamics NAV. If you fit in one of these, or similar, categories, start by studying Chapter 1 for a good overview of NAV and its tools. Then, you should review sections of other chapters as the topics apply to your specific areas of interest.

This book's illustrations are from the W1 Cronus database Dynamics NAV V2017.

## What this book covers

Chapter 1, *Introduction to NAV 2017*, starts with an overview of NAV as a business application system, which is followed by an introduction to the seven types of NAV objects, and the basics of C/AL and C/SIDE. Then, we will do some hands-on work, defining Tables, multiple Page types, and a Report. We'll close with a brief discussion of how backups and documentations are handled in C/SIDE.

Chapter 2, *Tables*, focuses on the foundation level of NAV data structure: Tables and their structures. We will cover Properties, Triggers (where C/AL resides), Field Groups, Table Relations, and SumIndexFields. We'll work our way through hands-on creation of several tables in support of our example application. We will also review the types of tables found in the NAV applications.

Chapter 3, *Data Types and Fields*, teaches you about fields, the basic building blocks of NAV data structure. We will review the different Data Types in NAV and cover all the field properties and triggers in detail, as well as review the three different Field Classes. We'll conclude with a discussion about the concept of filtering and how it should be considered in database structure design.

Chapter 4, *Pages - the Interactive Interface*, reviews the different types of pages, their structures (Triggers, Properties), and general usage. We'll build several pages for our example application using the Page Wizard and Page Designer. We will also study the different types of controls that can be used in pages. In addition, we'll review how and where actions are added to pages.

Chapter 5, *Queries and Reports*, teaches you about both Queries and Reports--two methods of extracting data for presentation to users. For Queries, we will study how they are constructed and some of the ways they are utilized. For Reports, we will walk through report data flow and the variety of different report types. We will study the two Report Designers, the C/SIDE Report Designer and the Visual Studio Report Designer and how a NAV report is constructed using both of these. We'll learn what aspects of reports use one designer and what aspects use the other. As in previously studied objects, we will discuss Properties and Triggers. We will review how reports can be made interactive and will do some hands-on report creation.

Chapter 6, *Introduction to C/SIDE and C/AL*, explains general Object Designer Navigation as well as the individual Designers (Table, Page, Report). We'll study C/AL code construction, syntax, variable types, expressions, operators, and functions. We will then take a closer look at some of the more frequently used built-in functions. This chapter will wrap up with an exercise adding some C/AL code to a report objects created in an earlier exercise.

Chapter 7, *Intermediate C/AL*, digs deeper into C/AL development tools and techniques. We will review some more advanced built-in functions, including those relating to dates and decimal calculations, both critical business application tools. We'll study C/AL functions that support process flow control functions, input/output, and filtering. Then, we'll do a review of methods of communication between objects. Finally, we'll apply some of what we've learned to enhance our example application.

Chapter 8, *Advanced NAV Development Tools*, reviews some of the more important elements of the Role Tailored User Experience; in particular, the Role Center Page construction. We will dig into the components of a Role Center Page and how to build one. We'll also cover two of the powerful ways of connecting NAV applications to the world outside of NAV, XMLports, and Web Services. To better understand these, we will not only review their individual component parts, but we will go through the hands-on effort of building an example of each one.

Chapter 9, *Successful Conclusions*, gives you a detailed study of how NAV functions are constructed and teaches you how to construct your own functions. We will learn more about tools and features built into C/AL and C/SIDE. We will study the new debugger, review the support for Test-Driven Development, and take a look at the ability to integrate .NET Client Add-ins and integrate a .NET Add-in into our example applications. Finally, we will review tips for design efficiency, updating, and upgrading the system, all with the goal of helping us become more productive, high quality NAV developers.

## What you need for this book

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

- A license and database that you can use for development experimentation. The ideal license is a full Developer's license. If your license only contains the 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 Pages and Tables and the C/AL code contained therein.
- A copy of the NAV Cronus demo/test database for your development testing and study. It would be ideal if you also had a copy of a production database at hand for examination as well. This book's illustrations are from the W1 Cronus database for V2017.
- The hardware and software requirements for installing and running Microsoft Dynamics NAV can be found at <https://msdn.microsoft.com/en-us/dynamics-nav/system-requirements-for-microsoft-dynamics-nav>.

Access to other NAV manuals, training materials, websites, and experienced associates will obviously be of benefit, but they are not required for the time with this book to be a good investment.

# Who this book is for

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 type of software required to support those applications
  - Has significant programming experience
  - Has access to a copy of NAV 2017, 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 that 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 experienced business analyst, consultant, or advanced student of applications software development who wants to learn more about NAV because it is one of the most widely used, most flexible business application systems available

The reader of this book:

- Does not need to be an expert in object-oriented programming
- Does not need previous experience with NAV, C/AL, or C/SIDE

## Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The `Customer` table would be filtered to report only customers who have an outstanding balance greater than zero."

A block of code is set as follows:

```
CalculateNewDate;  
"Date Result" := CALCDATE("Date Formula to Test","Reference Date  
for Calculation");
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on **Preview** to see the Report display onscreen."

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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book's title in the subject of your message.

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

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Programming-Microsoft-Dynamics-NAV-Fifth-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## 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 could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## **Piracy**

Piracy of copyrighted 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 website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto: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**

If you have a problem with any aspect of this book, you can contact us at [questions@packtpub.com](mailto:questions@packtpub.com), and we will do our best to address the problem.

# 1

## Introduction to NAV 2017

*"Time changes all things; there is no reason why language should escape this universal law."*

*- Ferdinand de Saussure*

*"When we use a language, we should commit ourselves to knowing it, being able to read it, and writing it idiomatically."*

*- Ron Jeffries*

Microsoft Dynamics NAV has one of the largest installed user bases of any **enterprise resource planning (ERP)** system serving over 100,000 companies and one million plus individual users at the time of this writing. The community of supporting organizations, consultants, implementers, and developers continues to grow and prosper. The capabilities of the off-the-shelf product increase with every release. The selection of the add-on products and services expands both in variety and depth.

The release of Microsoft Dynamics NAV 2017 continues its 20+ year history of continuous product improvement. It provides more user options for access and output formatting. For new installations, NAV 2017 includes tools for rapid implementation. For all installations, it provides enhanced business functionality and more support for ERP computing in the cloud, including integration with Microsoft Office 365.

NAV 2017 is also the base foundation for Microsoft Dynamics 365 for Financials, which is a cloud-based subset of the product that you can customize using extensions. We will discuss the concept of extensions in this book.

Our goal in this chapter is to gain a big picture understanding of NAV 2017. You will be able to envision how NAV can be used by the managers (or owners) of an organization to help manage activities and the resources, whether the organization is for-profit or not-for-profit. You will also be introduced to the technical side of NAV from a developer's point of view.

In this chapter, we will take a look at NAV 2017, including the following:

- A general overview of NAV 2017
- A technical overview of NAV 2017
- A hands-on introduction to C/SIDE development in NAV 2017

## NAV 2017 - An ERP system

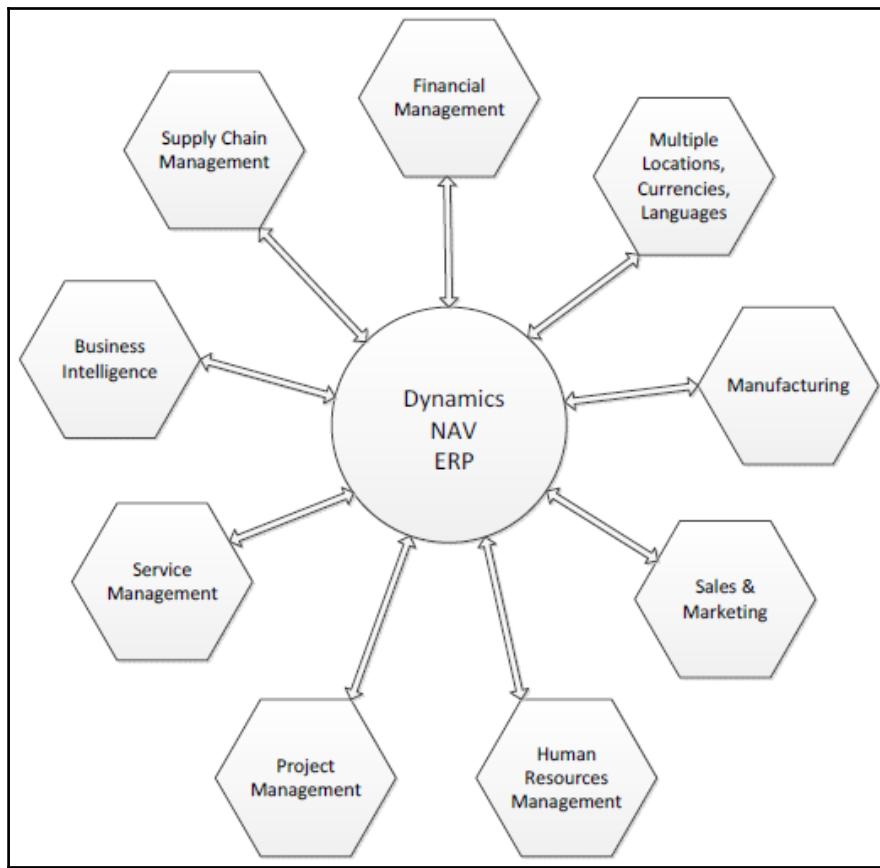
NAV 2017 is an integrated set of business applications designed to service a wide variety of business operations. Microsoft Dynamics NAV 2017 is an ERP system. An ERP system integrates internal and external data across a variety of functional areas including manufacturing, accounting, supply chain management, customer relationships, service operations, and human resources management, as well as the management of other valued resources and activities. By having many related applications well integrated, a full featured ERP system provides an **enter data once, use many ways** information processing toolset.

NAV 2017 ERP addresses the following functional areas (and more):

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

A good ERP system such as NAV 2017 is modular in design, which simplifies implementation, upgrading, modification, integration with third-party products, and expansion for different types of clients. All the modules in the system share a common database and, where appropriate, common data.

The groupings of individual NAV 2017 functions following is based on the **Departments** menu structure, supplemented by information from Microsoft marketing materials. The important thing is to understand the overall components that make up the NAV 2017 ERP system:



NAV 2017 has two quite different styles of **user interface (UI)**. One UI, the development environment, targets developers. The other UI style, the **RoleTailored client**, targets end users. In NAV 2017, there are four instances of the RoleTailored client - for Windows, for web interaction, for tablet use, and a phone client, which was introduced in NAV 2016. The example images in the following module descriptions are from the RoleTailored client's **Departments** menu in the Windows client.

## 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 that help to manage the capital resources of the business are part of NAV 2017's **Financial Management** module. These include all or part of the following application functions:

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

The **Financial Management** section of the **Departments** menu is as follows:

Financial Management	
	General Ledger
	Cash Management
	Cost Accounting
	Cash Flow
	Receivables
	Payables
	Fixed Assets
	Inventory
	Periodic Activities
	Setup

## Manufacturing

NAV 2017 manufacturing is general-purpose enough to be appropriate for **Make to Stock** (MTS), **Make to Order** (MTO), and **Assemble to Order** (ATO), as well as various subsets and combinations of those. Although off-the-shelf NAV is not particularly suitable for most process manufacturing and some of the very high volume assembly line operations, there are third-party add-on and add-in enhancements available for these applications. As with most of the NAV application functions, manufacturing can be implemented to be used in a basic mode or as a full featured system. 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 quantities and costs, plus tracking the planned use manufacturing resources, both on an unconstrained and constrained basis

The **Manufacturing** section of the **Departments** menu is as follows:



## Supply chain management

Obviously, some of the functions categorized as part of NAV 2017 **supply chain management (SCM)**, for example, sales and purchasing) are actively used in almost every NAV implementation. The supply chain applications in NAV include all or parts of the following applications:

- Sales order processing and pricing - supporting the heart of every business
- 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

Even though we might consider **Assembly** part of **Manufacturing**, the standard NAV 2017 **Departments** menu includes it in the **Warehouse** section:

	<b>Sales &amp; Marketing</b>	
	Sales	Marketing
	Order Processing	Inventory & Pricing
	<b>Purchase</b>	
	Planning	Inventory & Costing
	Order Processing	
	<b>Warehouse</b>	
	Orders & Contacts	Goods Handling Multipl...
	Planning & Execution	Inventory
	Goods Handling Order b...	Assembly

As a whole, these functions constitute the base components of a system appropriate for distribution operations, including those that operate on an ATO basis.

## 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 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 BI strategy include the following:

- Standard reports - distributed ready-to-use by end users
- Account schedules and analysis reports - a specialized report writer for **General Ledger** data
- Query, XMLport and Report designers - developer tools to support the creation of a wide variety of report formats, charts, and XML and CSV files
- Analysis by dimensions - a capability embedded in many of the other tools
- Interfaces into Microsoft Office and Microsoft Office 365 including Excel-communications of data either into NAV or out of NAV

- RDLC report viewer - provides the ability to present NAV data in a variety of textual and graphic formats, including user interactive capabilities
- Interface capabilities such as DotNet interoperability and web services - technologies to support interfaces between NAV 2017 and external software products
- NAV 2017 has standard packages for **Power BI**, both integrated on the role center as well as dashboards

## Artificial Intelligence

A new feature of NAV 2017 is integration with Cortana intelligence, which is used for forecasting. The algorithms used in **Artificial Intelligence (AI)** can be used to give users extra information to make business decisions.

## Relationship Management

NAV's **Relationship Management (RM)** functionality is definitely the *little sister* (or, if you prefer, *little brother*) of the fully featured standalone Microsoft CRM system and the new Dynamics 365 for Sales and Dynamics 365 for Marketing. The big advantage of NAV RM is its tight integration with NAV customer and sales data. With NAV 2016, Microsoft introduced a new way of integrating with CRM using **OData**.

Also falling under the heading of the 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:

- **Relationship management:**
  - Marketing campaigns - plan and manage promotions
  - Customer activity tracking - analyze customer orders
  - To do lists - manage what is to be done and track what has been done
- **Service management:**
  - Service contracts - support service operations
  - Labor and part consumption tracking - track the resources consumed by the service business
  - Planning and dispatching - managing service calls

## Human resource management

NAV **Human Resources** is a very small module, but it relates to a critical component of the business, employees. Basic employee data can be stored and reported via the master table (in fact, one can use **human resources (HR)** to manage data about individual contractors in addition to employees). A wide variety of individual employee attributes can be tracked by the 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
- Employee statistics - tracking government required employee attribute data such as age, gender, length of service

## 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 (in other words - billable) or internal. This module is often used by third parties as the base for vertical market add-ons (such as 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

## A developer's overview of NAV 2017

From the point of view of a developer, NAV 2017 consists of about almost five thousand potentially customizable off-the-shelf program objects plus the **integrated development environment (IDE)**-the **Client/Server Integrated Development Environment (C/SIDE)** development tools that allow us to modify existing objects and create new ones.

## NAV object types

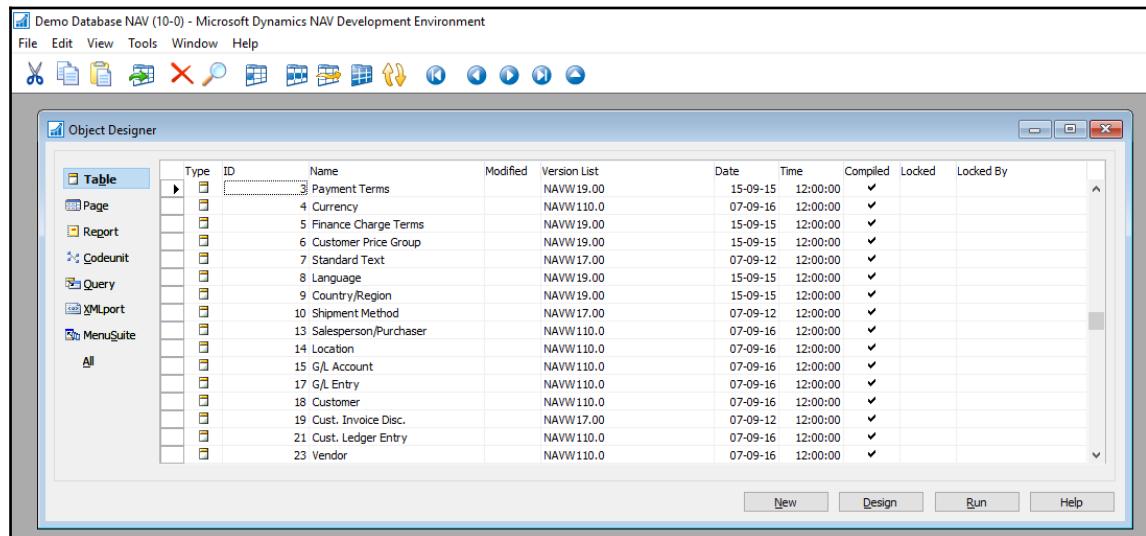
Let's start with basic definitions of the NAV 2017 object types:

- **Table:** Tables serve both to define the data structure and to contain the data records.
- **Page:** Pages are the way data is formatted and displayed appropriately for each of the client types and user roles.
- **Report:** Reports provide for the display of data to the user in hardcopy format, either onscreen (preview mode) or through a printing device. Report objects can also update data in processes with or without data display.
- **Codeunit:** Codeunits are containers for code utilized by other objects. Codeunits are always structured in code segments called functions.
- **Query:** Queries support extracting data from one or more tables, making calculations, and outputting in the form of a new data structure. Queries can output data directly to charts, to Excel, to XML, and to OData. They can be used as an indirect source for pages and reports.
- **XMLport:** XMLports allow the importing and exporting of data to/from external files. The external file structure can be in XML or other file formats.
- **MenuSuite:** MenuSuites contain menu entries that refer to other types of objects. MenuSuites are different from other objects. Menus cannot contain any code or logic. MenuSuite entries display in the **Departments** page in the navigation pane in the Windows client only. In the web and tablet clients, these are used to support search functions.

## The C/SIDE Integrated Development Environment

NAV 2017 includes an extensive set of software development tools. The NAV development tools are accessed through C/SIDE, which runs within the development environment client. This environment and its complement of tools are usually collectively referred to as C/SIDE. C/SIDE includes the **C/AL (Client Application Language)** compiler. All NAV programming uses C/AL. No NAV development can be done without using C/SIDE, but other tools are used to complement C/AL code (such as Visual Studio, .NET, JavaScript, COM controls, and OCX controls, among others).

The C/SIDE IDE is referred to as the **Object Designer** within NAV. It is accessed through a separate shortcut, which is installed as part of a typical full system installation. When we open the **Object Designer**, we see the following screen:



## Object Designer tool icons

When we open an object in the applicable Designer (Table Designer, Page Designer, and so on) for that object, we will see a set of tool icons at the top of the screen. The following table lists those icons and the object types to which they apply. On occasion, an icon will appear when it is of no use:

					
Table	✓	✓	✓ but no response	✓	✓
Page	✓	✓	✓	✓	✓
Report	✓	✓	✓	✓	✓
Codeunit	✓	✓	✓ but empty	✓	✓ Toggles debugger breakpoint
Query	✓	✓	✓	✓	✓
XMLport	✓	✓	✓ but no response	✓	✓
MenuSuite	✓ but of no use	No icon – <i>Alt+Enter</i>			

## C/AL programming language

The language in which NAV is coded is C/AL. A small sample of C/AL code within the **C/AL Editor** is shown here:

```
Table 36 Sales Header - C/AL Editor
64  Document Type - OnValidate()
65  |
66  Document Type - OnLookup()
67  |
68  Sell-to Customer No. - OnValidate()
69  CheckCreditLimitIfLineNotInsertedYet;
70  TESTFIELD(Status,Status::Open);
71  IF ("Sell-to Customer No." <> xRec."Sell-to Customer No.") AND
72  (xRec."Sell-to Customer No." <> '')
73  THEN BEGIN
74  IF ("Opportunity No." <> '') AND ("Document Type" IN ["Document Type":::Quote,"Document Type":::Order]) THEN
75  ERROR(
76    Text062,
77    FIELDCAPTION("Sell-to Customer No."),
78    FIELDCAPTION("Opportunity No."),
79    "Opportunity No.",
80    "Document Type");
81  IF HideValidationDialog OR NOT GUIALLOWED THEN
82    Confirmed := TRUE
83  ELSE
```

C/AL syntax is similar to Pascal syntax. Code readability is always 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.

Good software development focuses on design before coding and accomplishing design goals with a minimum of code. Dynamics NAV facilitates that approach. In 2012, a team made up of Microsoft and NAV community members began the NAV design patterns project. As defined in Wikipedia, *a design pattern is a general reusable solution to a commonly occurring problem*. Links to the NAV design patterns project information follow:

- <http://blogs.msdn.com/b/nav/archive/2013/08/29/what-is-the-nav-design-patterns-project.aspx>
- <https://community.dynamics.com/nav/w/designpatterns/default.aspx>
- <https://www.packtpub.com/big-data-and-business-intelligence/learning-dynamics-nav-patterns>



A primary goal of this project is to document patterns that exist within NAV. In addition, new best practice patterns have been suggested as ways to solve common issues we encounter during our customization efforts. Now, when working on NAV enhancements, we will be aided by reference to the documentation of patterns within NAV. This allows us to spend more of our time designing a good solution using existing, proven functions (the documented patterns), spending less time writing and debugging code. A good reference for NAV design and development using patterns can be found here: <https://www.packtpub.com/application-development/microsoft-dynamics-nav-2013-application-design>

The NAV 2017 *Reusing Code* section states the following:

*"Reusing code makes developing applications both faster and easier. More importantly, if you organize your C/AL code as suggested, your applications will be less prone to errors. By centralizing the code, you will not unintentionally create inconsistencies by performing the same calculation in many places, for example, in several triggers that have the same table field as their source expression. If you have to change the code, you could either forget about some of these triggers or make a mistake when you modify one of them."*

Much of our NAV development work is done by assembling references to previously defined objects and functions, adding new data structure where necessary. As the tools for NAV design and development provided both by Microsoft and the NAV community continue to mature, our development work becomes more oriented to design and less to coding. The end result is that we are more productive and cost effective on behalf of our customers. Everyone wins.

## NAV object and system elements

Here are some important terms used in NAV:

- **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 that 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, usually configured in groups of features. Microsoft Partners have access to full development licenses to provide support and customization services for their clients. End-user firms can purchase licenses allowing them developer access to NAV. A Training License can also be generated, which contains any desired set of features and expires after a specified period of time.

### License limits



The NAV license limits access to the C/AL code within tables, pages, and codeunits differently than the C/AL code buried within reports or XMLports. The latter can be accessed with a lower level license (that is, less expensive). If a customer has license rights to the report designer, which many do, they can access C/AL code within Report and XMLport objects. But access to C/AL code in a table, page, or codeunit requires a more expensive license with developer privileges. As a result, C/AL code within tables, pages, and codeunits is more secure than that within report and XMLport objects.

- **Field:** An individual data item, defined either in a table or in the working storage (temporary storage) of an object.

- **Record:** A group of fields (data items) handled as a unit in many operations. Table data consists of rows (records) with columns (fields).
- **Control:** In MSDN, a control is defined as a component that provides (or enables) user interface (UI) capabilities.
- **Properties:** These are the attributes of the element such as an object, field, record, or control that defines some aspect of its behavior or use. Example property attributes include display captions, relationships, size, position, and whether editable or viewable.
- **Trigger:** Mechanisms that initiate (fire) an action when an event occurs and is communicated to the application object. A trigger in an object is either empty, contains code that is executed when the associated event fires the trigger, or only contains comments (in a few cases, this affects the behavior of the trigger). Each object type, data field, control, and so on may have its own set of predefined triggers. The event trigger name begins with the word `On`, such as `OnInsert`, `OnOpenPage`, or `OnNextRecord`. NAV triggers have similarities to those in SQL, but they are not the same (similarly named triggers may not even serve similar purposes). NAV triggers are locations within objects where a developer can place comments or C/AL code. When we view the C/AL code of an object in its designer, we can see non-trigger code groups that resemble NAV event-based triggers:
  - **Documentation:** This can contain comments only, no executable code. Every object type except **MenuSuite** has a single Documentation section at the beginning of the C/AL code.
  - **Functions:** These can be defined by the developer. They are callable routines that can be accessed by other C/AL code from either inside or outside the object where the called function resides. Many functions are provided as part of the standard product. As developers, we may add our own custom functions as needed.

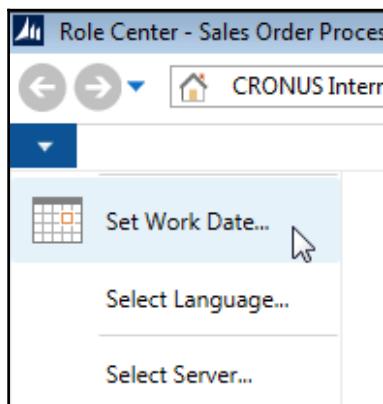
- **Object numbers and field numbers:** All objects of the same object type are assigned a number unique within the object type. All fields within an object are assigned a number unique within the object (that is, the same field number may be repeated within many objects whether referring to similar or different data). In this book, we will generally use comma notation for these numbers (fifty thousand is 50,000). In C/AL, no punctuation is used. 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 these number ranges can be modified or deleted with a developer's license, but cannot be created. Field numbers in standard objects often start with 1 (one). Historically, object and field numbers from 50,000 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. Field numbers from 90,000 to 99,999 should not be used for new fields added to standard tables as those numbers are sometimes used in training materials. Microsoft allocates ranges of object and field numbers to **Independent Software Vendor (ISV)** developers for their add-on enhancements. Some such objects (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 object numbering practice:

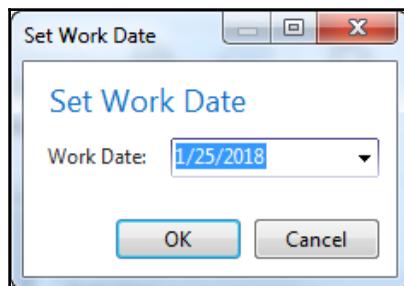
Object number range	Usage
1 - 9,999	Base-application objects
10,000 - 49,999	Country-specific objects
50,000 - 99,999	Customer-specific objects
100,000 - 98,999,999	Partner-created objects
Above 98,999,999	Microsoft territory

- **Events:** Functions can subscribe to events that are raised in the system. NAV 2017 has both platform and manual events. Functions can also be used to raise events.

- **Work Date:** This is a date controlled by the user operator. It 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 the 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. There are settings to allow limiting the range of work dates allowed. The work date can be set by clicking on arrowhead dropdown below the Microsoft Dynamics icon and selecting the **Set Work Date...** option:



- Clicking on **Set Work Date...** in the dropdown options displays the **Set Work Date** screen. Or click on the date in the status bar at the bottom of the RTC window. In either case, we can enter a new **Work Date**:



## NAV functional terminology

For various application functions, NAV uses terminology more similar to accounting than to traditional data processing terminology. Here are some examples:

- **Journal:** A table of unposted 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 posted transaction entries that have been processed. For example, General Ledger, Customer Ledger, Vendor Ledger, Item Ledger, and so on. Some ledgers have subordinate detail ledgers, typically providing a greater level of quantity and/or value detail. With minor exceptions, ledger entries cannot be edited. This maintains auditable data integrity.
- **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 posted at the same time.
- **Register:** An audit trail showing a history, by Entry Number ranges, of posted Journal Batches.
- **Document:** A formatted page such as an Invoice, a Purchase Order, or a Payment Check, typically one page for each primary transaction (a page may require display scrolling to be fully viewed).

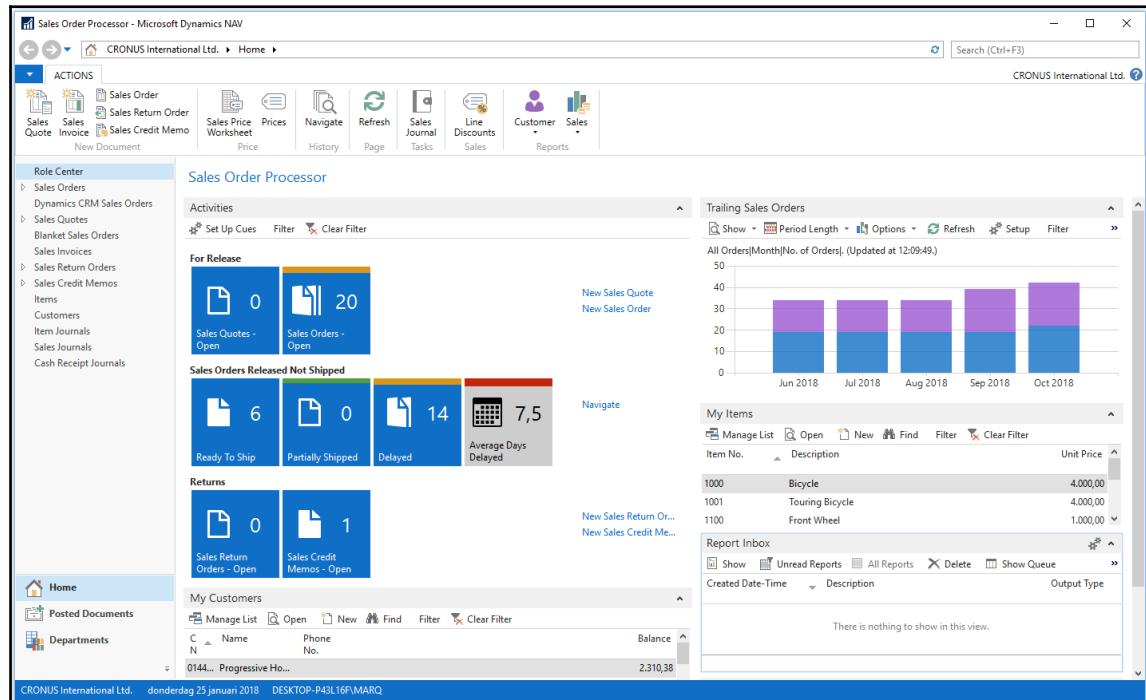
## User Interface

NAV 2017 **user interface (UI)** is designed to be role oriented (also called role tailored). The term role oriented means tailoring the options available to fit the user's specific job tasks and responsibilities. If user access is through one of the clients, the **Role Tailored Client (RTC)** will be employed. If the user access is via a custom-built client, the developer will have more responsibility to make sure the user experience is role tailored.

The first page that a user will see is the **Role Center** page. The **Role Center** page provides the user with a view of work tasks to be done; it acts as the user's home page. The home **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 installation.

Someone whose role is focused on order entry will probably see a different RTC home page than the user whose role focuses on invoicing, even though both user roles are in what we generally think of as Sales & Receivables. The NAV 2017 RTC allows a great deal of flexibility for implementers, system administrators, managers, and individual users to configure and reconfigure screen layouts and the set of functions that are visible.

The following screenshot is the out-of-the-box **Role Center** for a **Sales Order Processor**:



The key to properly designing and implementing any system, especially a role tailored system, is the quality of the User Profile analysis done as the first step in requirements analysis. User Profiles identify the day-to-day needs of each user's responsibilities relative to accomplishing the business' goals. Each user's tasks must be mapped to individual NAV functions or elements, identifying how those tasks will be supported by the system. A successful implementation requires the use of a proven methodology. It is very important that the up-front work is done and done well. The best programming cannot compensate for a bad definition of goals.

In our exercises, we will assume the up-front work has been well done and we will concentrate on addressing the requirements defined by our project team.

## Hands-on development in NAV 2017

One of the best ways to learn a new set of tools, like those that make up a programming language and environment, is to experiment with them. We're going to have some fun doing that throughout this book. We're going to experiment where the cost of errors (otherwise known as learning) is small. Our development work will be a custom application of NAV 2017 for a relatively simple, but realistic, application.

We're going to do our work using the Cronus demo database that is available with all NAV 2017 distributions and is installed by default when we install the NAV 2017 demo system. The simplest way to install the NAV 2017 demo is to locate all the components on a single workstation. A 64-bit system running Windows 10 will suffice. Additional requirements information is available in the MSDN library (<https://msdn.microsoft.com/en-us/dynamics-nav/system-requirements-for-microsoft-dynamics-nav>) under the heading *System Requirements for Microsoft Dynamics NAV 2017*. Other helpful information on installing NAV 2017 (the demo option is a good choice for our purposes) and addressing a variety of setup questions is available in the NAV 2017 area of the MSDN library. In fact, all the help information for NAV 2017 is accessible in the MSDN library.

The Cronus database contains all of the NAV objects and a small, but reasonably complete, set of data populated in most of the system's functional applications areas. Our exercises will interface very slightly with the Cronus data, but will not depend on any specific data values.

To follow along with all our exercises as a developer, you will need a developer license for the system with rights allowing the creation of objects in the 50,000 to 50,099 number range. This license should also allow at least read access to all the objects numbered below 50,000. If you don't have such a license, check with your Partner or your Microsoft sales representatives to see if they will provide a training license for your use.

## NAV 2017 development exercise scenario

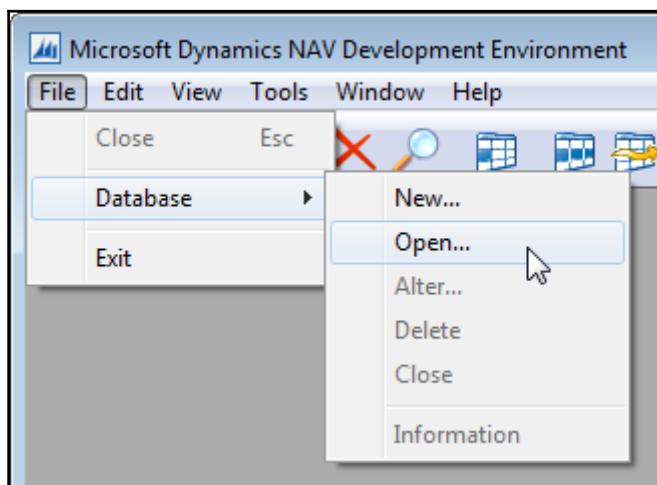
Our business is a small radio station that features a variety of programming, news, music, listener call-ins, and other program types. Our station call letters are WDTU. Our broadcast materials come from several sources and in several formats: vinyl records, CDs, MP3s, and downloaded digital (usually MP3s). While our station has a large library, especially of recorded music, sometimes our program hosts (also called disc jockeys or DJs) want to share material from other sources. For that reason, we need to be able to easily add items to our playlists (the list of what is to be broadcast) and also have an easy-to-access method for our DJs to preview MP3 material.

Like any business, we have accounting and activity tracking requirements. Our income is from selling advertisements. We must pay royalties for music played, fees for purchased materials such as prepared text for news, sports, and weather information, and service charges for our streaming Internet broadcast service. As part of our licensed access to the public airwaves, a radio station is required to broadcast public service programming at no charge. Often, that is in the form of **Public Service Announcements (PSAs)** such as encouraging traffic safety or reduction in tobacco use. Like all radio stations, we must plan what is to be broadcast (create schedules) and track what has been broadcast (such as ads, music, purchased programming, and PSAs) by date and time. We bill our customers for the advertising, pay our vendors their fees and royalties, and report our public service data to the appropriate government agency.

## Getting started with application design

The design for our radio station will start with a Radio Show table, a Radio Show Card page, a Radio Show List page, and a simple Radio Show List report. Along the way, we will review the basics of each NAV object type.

When we open the NAV Development Environment for the first time or to work on a different database, we must define what database should be opened. Click on **File | Database | Open...** and then choose a database:



## Application tables

Table objects are the foundation of every NAV application. Tables contain data structure definitions, as well as properties that describe the behavior of the data, including data validations and constraints.

More business logic is required in complex applications than in simple data type validation, and NAV allows C/AL code to be put in the table to control the insertion, modification, and deletion of records as well as logic on the field level. When the bulk of the business logic is coded on the table level, it is easier to develop, debug, support, modify, and even upgrade. Good design in NAV requires that as much of the business logic as possible resides in the tables. Having the business logic coded on the table level doesn't necessarily mean the code is resident in the table. NAV 2017 **Help** recommends the following guidelines for placing C/AL code:



In general, put the code in codeunits, instead of on the object on which it operates. This promotes a clean design and provides the ability to reuse code. It also helps enforce security. For example, typically users do not have direct access to tables that contain sensitive data, such as the **General Ledger Entry** table, nor do they have permission to modify objects. If you put the code that operates on the general ledger in a codeunit, give the codeunit access to the table, and give the user permission to execute the codeunit, then you will not compromise the security of the table and the user will be able to access the table. If you must put code on an object instead of in a codeunit, then put the code as close as possible to the object on which it operates. For example, put code that modifies records in the triggers of the table fields.

## Designing a simple table

Our primary master data table will be the Radio Show table. This table lists our inventory of shows available to be scheduled.

First, open the NAV Development Environment, click on **Tools | Object Designer** and select **Table**. We can view or modify the design of existing master tables in NAV by highlighting the table (for example, *Table 18 - Customer*, or *Table 27 - Item*) and clicking on **Design**.

Each master table has a standard field for the primary key (a Code data type field of 20 characters called No.) and has standard information regarding the entity the master record represents (for example, Name, Address, City, and so on, for the Customer table and Description, Base Unit of Measure, Unit Cost, and so on for the Item table).

The Radio Show table will have the following field definitions (we may add more later):

Field names	Definitions
No.	20-character text (code)
Radio Show Type	10-character text (code)
Name	50-character text
Run Time	Duration
Host No.	20-character text (code)
Host Name	50-character text
Average Listeners	Decimal
Audience Share	Decimal
Advertising Revenue	Decimal
Royalty Cost	Decimal

In the preceding list, three of the fields are defined as Code fields, which are text fields that limit the alphanumeric characters to upper case values. Code fields are used throughout NAV for primary key values. Code fields are used to reference or be referenced by other tables (foreign keys). No. will be the unique identifier in our table. We will utilize a set of standard internal NAV functions to assign a user-defined No. series range that will auto-increment the value on table insertion and possibly allow for user entry (as long as it is unique in the table) based on a setup value. The Host No. references the standard Resource table and the Radio Show Type field will reference a custom table we will create to allow for flexible Type values.

We will have to design and define the reference properties at the field level in the Table Designer, as well as compile them, before the validation will work. At this point, let's just get started with these field definitions and create the foundation for the Radio Show table.

## Creating a simple table

To invoke the table designer, open the NAV 2017 Development Environment and the database in which we will be doing our development. In the **Object Designer**, click on **Table** (in the left column of buttons) and click **New** (in the bottom row of buttons). Enter the first field number as 1 (the default is 1), and increment each remaining field number by 10 (the default is 1). Sometimes, it is useful to leave large gaps (such as jumping from 80 to 200 or 500) when the next set of fields have a particular purpose not associated with the prior set of fields.

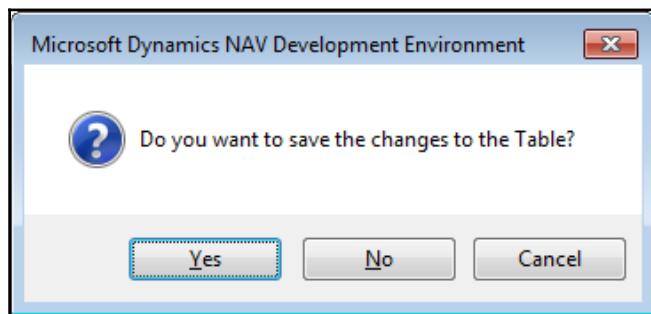


The fields combining the primary key can use the numbers up to 9 since they are very unlikely to change. If this is changed, there is a substantial amount of refactoring to be done.

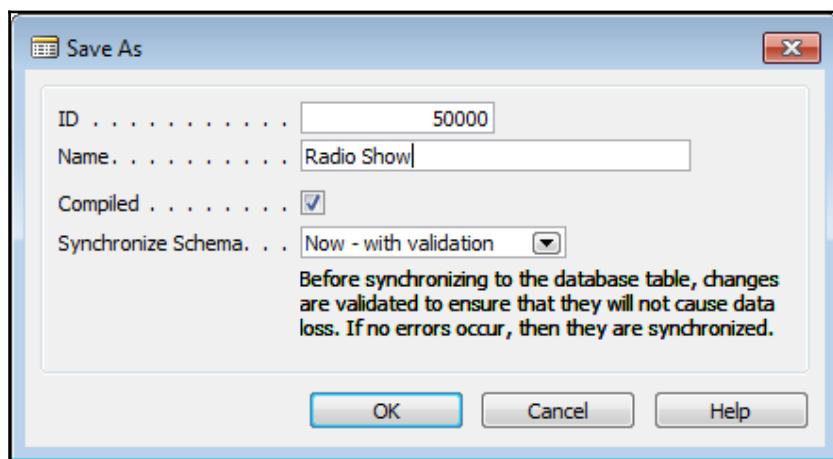
NAV 2017 Help says to not leave gaps in field numbers. Based on many years of experience, the authors disagree. Leaving numbering gaps will allow us to later add additional fields between existing fields, if necessary. The result will be data structures that are (at least initially) easier to read and understand. Once a table is referenced by other objects or contains any data, the field numbers of previously defined fields should not be changed.

The following screenshot shows our new table definition in the Table Designer:

Now we can close the table definition (click on **File | Save** or **Ctrl + S** or press **Esc** or close the window - the first two options are the explicit methods of saving our work). We will see a message reminding us to save our changes:



Click on **Yes**. We must now assign the object number (use 50000) and a unique name (it cannot duplicate the same first 30 characters of another table object in the database). We will name our table **Radio Show** based on the master record to be stored in the table:

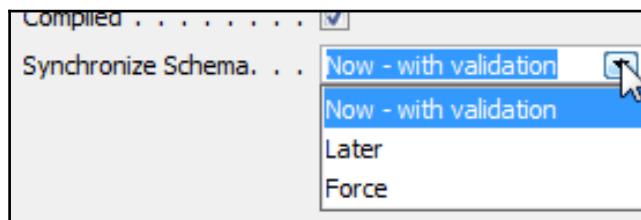


Note that the **Compiled** option is automatically checked and the **Synchronize Schema** option is set to **Now - with validation**, which are the defaults for NAV. Once we press **OK** and the object is successfully compiled, it is immediately ready to be executed within the application. If the object we were working on was not ready to be compiled without error, we could uncheck the **Compiled** option in the **Save As** window.



Uncompiled objects will not be considered by C/SIDE when changes are made to other objects. Be careful. Until we 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 we have compiled an object, we have not confirmed that all is well. We may have made changes that affect other objects which reference the modified object. As a matter of good work habit, we should recompile all objects before we end work for the day.

The **Synchronize Schema** option choice determines how table changes will be applied to the table data in SQL Server. When the changes are validated, any changes that would be destructive to existing data will be detected and handled either according to a previously defined upgrade codeunit or by generating an error message. The **Synchronize Schema** option choices are shown in the following screenshot:



Refer to the documentation (<https://msdn.microsoft.com/en-us/dynamics-nav/synchronizing-table-schemas>) in the section called *Synchronizing Table Schemas* for more detail.

## Pages

Pages provide views of data or processes designed for on-screen display (or exposure as web services) and also allow for user data entry into the system. Pages act as containers for action items (menu options).

There are several basic types of display/entry pages in NAV 2017:

- List
- Card
- Document
- Journal/Worksheet
- List Plus
- Confirmation Dialog
- Standard Dialog

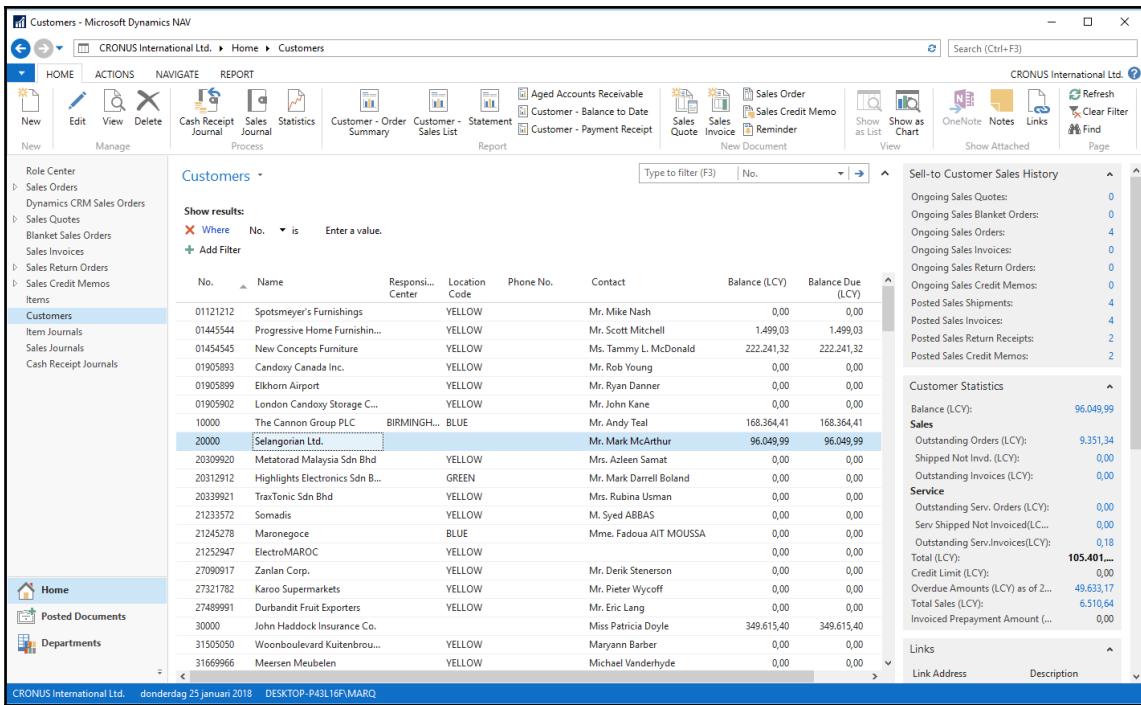
There are also page parts (they look and program like a page, but aren't intended to stand alone) as well as UIs that display like pages, but are not page objects. The latter user interfaces are generated by various dialog functions. In addition, there are special page types such as **Role Center** pages and **Navigate** pages (for Wizards).

## 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 buttons, Action items, and Page Parts. While there are a few instances where we must include C/AL code within page or page control triggers, it is good practice to minimize the amount of code embedded within pages. Any data-related C/AL code should be located in the table object rather than in the page object.

## List pages

List pages display a simple list of any number of records in a single table. The **Customers** List page (with its associated **FactBoxes**) in the following screenshot shows a subset of the data for each customer displayed. 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. Standard list pages are always displayed with the Navigation Pane on the left:

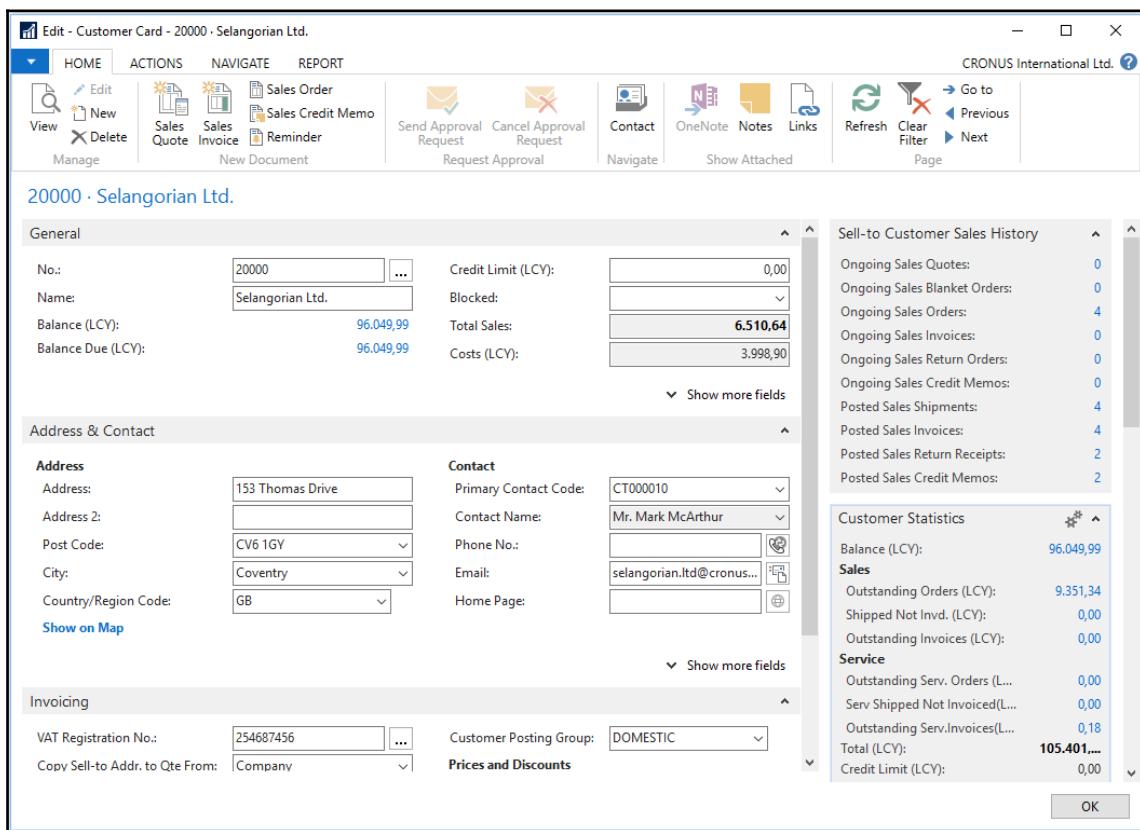


## Card pages

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 have **FastTabs** (a FastTab consists of a group of controls with each tab focusing on a different set of related customer data). FastTabs can be expanded or collapsed dynamically, allowing the data visible at any time to be controlled by the user. Important data elements can be promoted to be visible even when a FastTab is collapsed.

Card pages for master records display all the required data entry fields. If a field is set to **ShowMandatory** (a control property we will discuss in Chapter 4, *Pages - the Interactive Interface*), a red asterisk will display until the field is filled. Typically, card pages also display **FactBoxes** containing summary data about related activity. Thus cards can be used as the primary inquiry point for master records. The following screenshot is a sample of a standard **Customer Card**:



## Document pages

A document page looks like a card page with one tab containing a **List Part** page. An example is the **Sales Order** page shown in the following screenshot. In this example, the first tab and last four tabs are in card page format showing **Sales Order** data fields that have a single occurrence on the page (in other words, do not occur in a repeating column).

The second tab from the top is in List Part page format (all fields are in repeating columns) showing the **Sales Order** line items. **Sales Order** 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 area is related data and computations (FactBoxes) that have been retrieved and formatted. The top FactBox contains information about the ordering customer. The bottom FactBox contains information about the item on the currently highlighted sales line:

The screenshot shows the Microsoft Dynamics NAV interface for editing a Sales Order. The title bar reads "Edit - Sales Order - 101015 - Autohaus Mielberg KG". The ribbon has tabs for HOME, ACTIONS, and NAVIGATE. The ACTIONS tab is selected, showing various options like Release, Reopen, Copy Document..., Order Promising, Statistics, Assembly Orders, Archive Document, Shipments, Invoices, Email Confirmation..., Print Confirmation..., Manage, Release, Prepare, Order, Documents, Order Confirmation, Posting, Request Approval, Show Attached, and Page.

The main area displays the Sales Order details for "101015 - Autohaus Mielberg KG". The General section includes fields for Customer (Autohaus Mielberg KG), Due Date (4-2-2018), Contact, Requested Delivery Date, Posting Date (26-1-2018), External Document No., Order Date (21-1-2018), Status (Released), and a "Show more fields" button.

The Lines section lists the order items:

Type	No.	Description	Location Code	Quantity	Qty. to Assemble to Order
Item	1972-S	MUNICH Swivel Chair, yellow	RED	6	
Item	1968-S	MEXICO Swivel Chair, black	RED	5	
Item	1896-S	ATHENS Desk	RED	12	
Item	1906-S	ATHENS Mobile Pedestal	RED	12	

Below the lines section, there are summary fields for Subtotal Excl. VAT (EUR: 19.395,95), Inv. Discount Amount Excl. VAT (EUR: 0,00), Invoice Discount % (0), Total Excl. VAT (EUR: 19.395,95), Total VAT (EUR: 0,00), and Total Incl. VAT (EUR: 19.395,95).

The right side of the screen contains FactBoxes:

- Sell-to Customer Sales History** (includes Ongoing Sales Quotes, Sales Blanket Orders, Sales Orders, Sales Invoices, Sales Return Orders, Sales Credit Memos, Sales Shipments, Sales Invoices, Sales Return Receipts, Sales Credit Memos).
- Customer Details** (Customer ID: 49633663, Phone No.: autohaus.mielberg.kg@cro..., Email: autohaus.mielberg.kg@cro..., Fax No., Credit Limit: 0,00, Available Balance: 0,00, Payment Term: 14 DAYS, Contact).
- Sales Line Details** (Item No.: 1972-S, Required Quantity: 3, Availability: Shipment Date: 21-1-2018, Item Availability: -7, Available Inventory: -4, Scheduled Receipt: 0).

An "OK" button is located at the bottom right of the FactBox area.

## 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 also have a section at the bottom that shows details about the selected line and/or totals for the displayed data. These pages may 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 (though this may be a matter of personal or site preference). An example of the **General Journal** page in Finance is shown in the following screenshot:

The screenshot shows the Microsoft Dynamics NAV interface for the General Journal. The top navigation bar includes links for HOME, ACTIONS, and NAVIGATE, along with the company name CRONUS International Ltd. The ribbon menu has sections for Manage, Bank, Import Payroll Transactions, Payroll, Apply Entries..., Preview Posting, Insert Conv. LCY Rndg. Lines, Dimensions, Post, Post and Print, Test Report..., Get Standard Journals..., Save as Standard Journal..., Email as Attachment, Microsoft Excel, Ledger Entries History, Refresh, Find, and Page.

The main area displays a list of journal entries. The columns include Posting Date, Document Type, Document No., Account Type, Account No., Description, Gen. Posting Type, Gen. Bus. Posting ..., Gen. Prod. Posting ..., Amount, Bal. Account Type, and Bal. Account No. A specific row is highlighted for "Packing Machine 2018".

On the right side, there are two FactBoxes: "Dimensions" and "Incoming Document Files". The "Dimensions" FactBox shows categories like DEP..., PROD, and Production. The "Incoming Document Files" FactBox shows a table with columns for Name and Type, indicating "There is nothing to show in this view".

At the bottom, a summary table provides totals for Account Name, Bal. Account Name, Balance, and Total Balance. The "OK" button is visible in the bottom right corner.

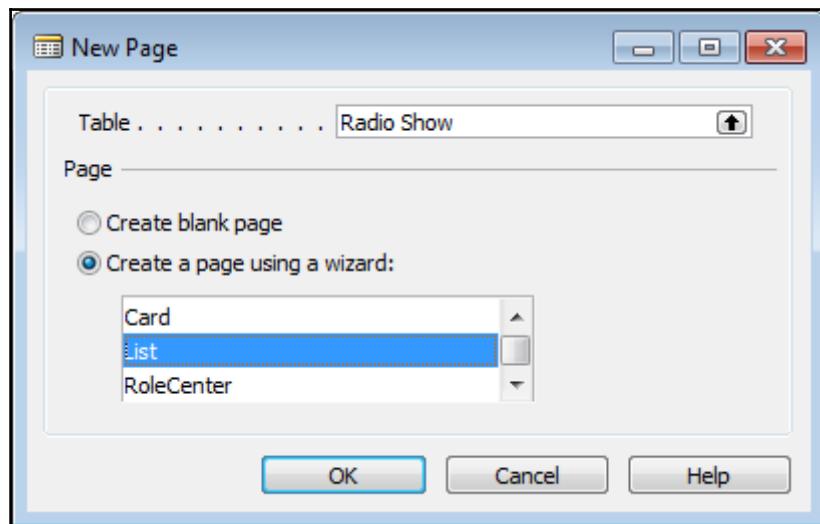
Account Name	Bal. Account Name	Balance	Total Balance
Increases during the Year		110,97	0,00

## Creating a List page

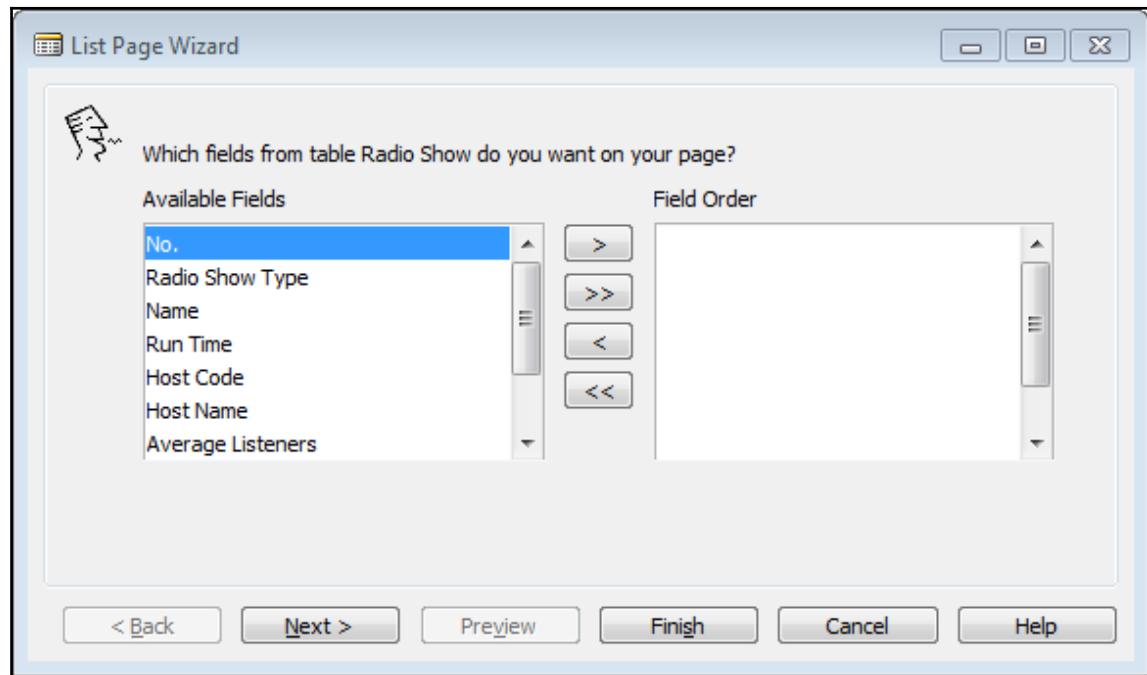
Now we will create a List page for the table we created earlier. A List page is the initial page that is displayed when a user accesses any data table. The NAV Development Environment has Wizards (object generation tools) to help create basic pages. Generally, after our Wizard work is done, we will spend additional time in the Page Design tool to make the layout ready for presentation to users.

Our first List page will be the basis for viewing our Radio Show master records. From the **Object Designer**, click on **Page**, then click on **New**. The **New Page** screen will appear. Enter the name (Radio Show) or table object ID (50000) in the **Table** field. This is the table to which the page will be bound. We can add additional tables to the page object C/AL Global Variables after we close the Wizard, as then we will be working in the Page Designer.

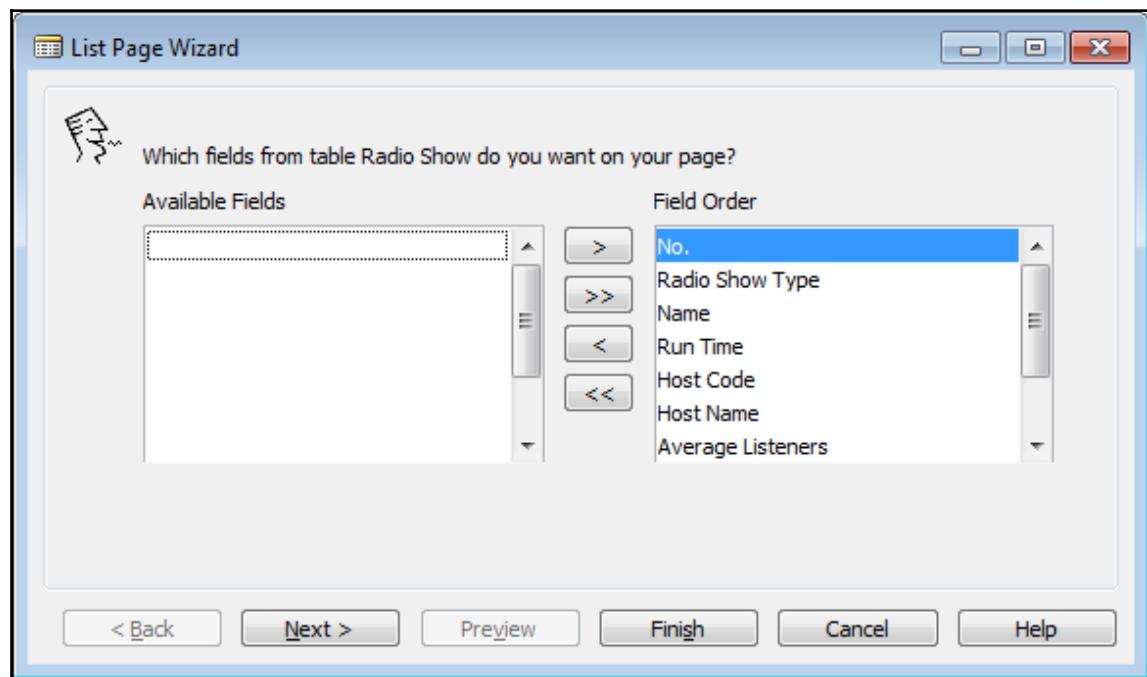
Choose the **Create a page using a wizard:** option and select **List**, as shown in the following screenshot. Click on **OK**:



The next step in the wizard shows the fields available for the List page. We can add or remove any of the field columns using the >, <, >>, and << buttons:



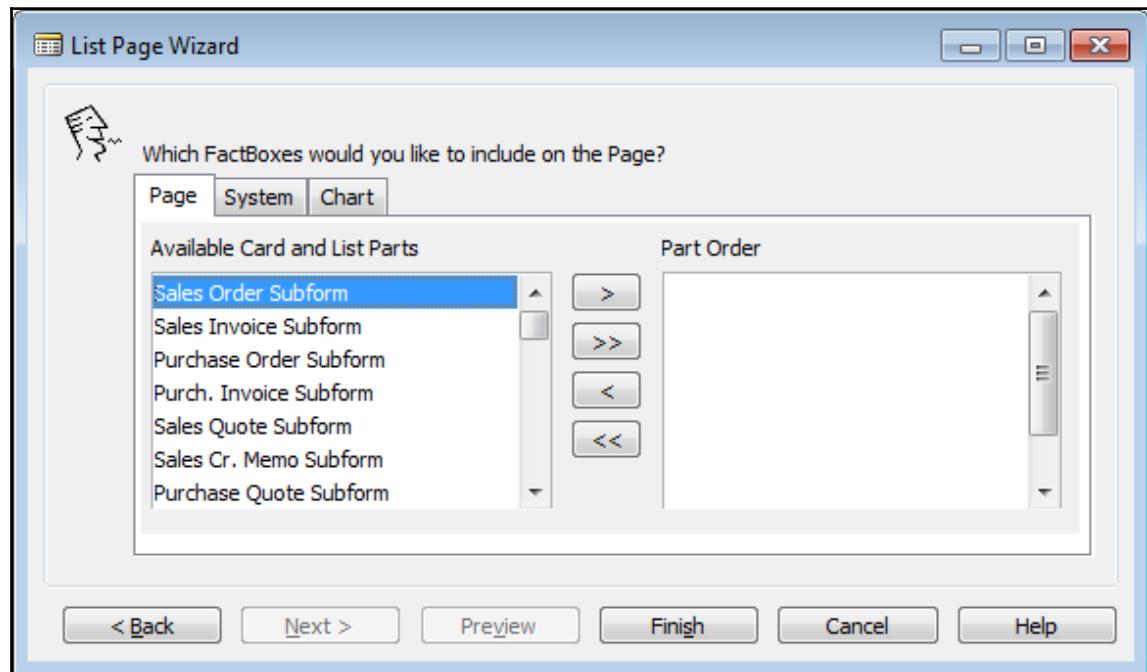
Add all the fields using >> and click on **Next >**:



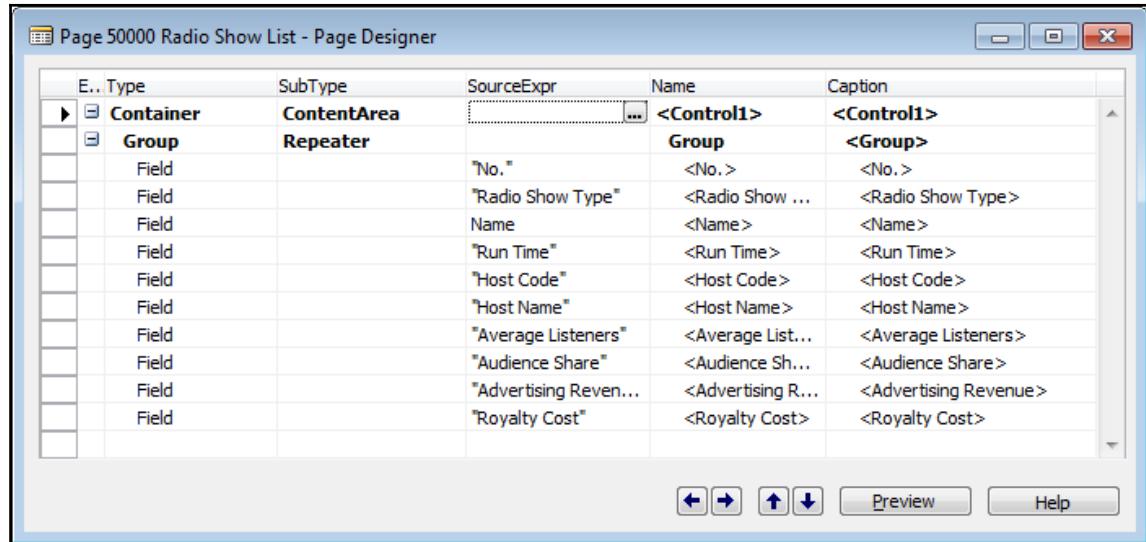
The next Wizard step shows the Subforms, System FactBoxes, and Charts that are available to add to our page:



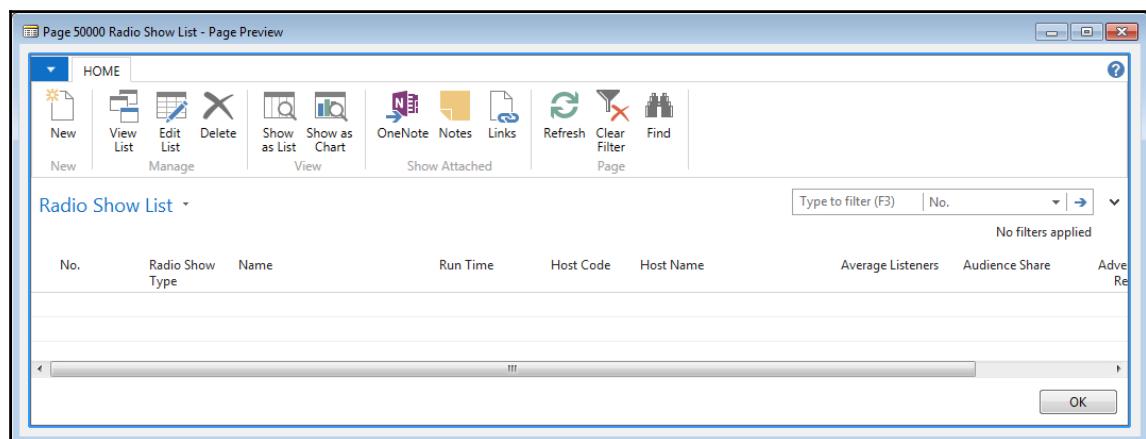
Subforms should properly be named Subpages. Such a change is being considered by Microsoft.



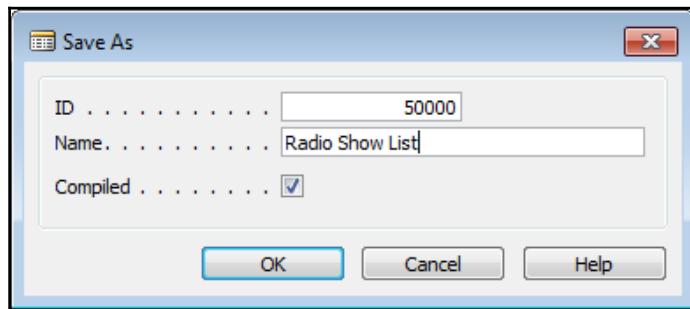
We can add these later in the Page Designer as needed. Click **Finish** to exit the wizard and enter the Page Designer:



Click on **Preview** to view the page with the default ribbon. Note that in **Preview** mode, we cannot insert, modify, or delete any of the layout or enter data. The **Preview** page is not connected to the database data. We need to compile the page and Run it to manipulate data. In the following screenshot, some fields are out of sight on the right-hand side:

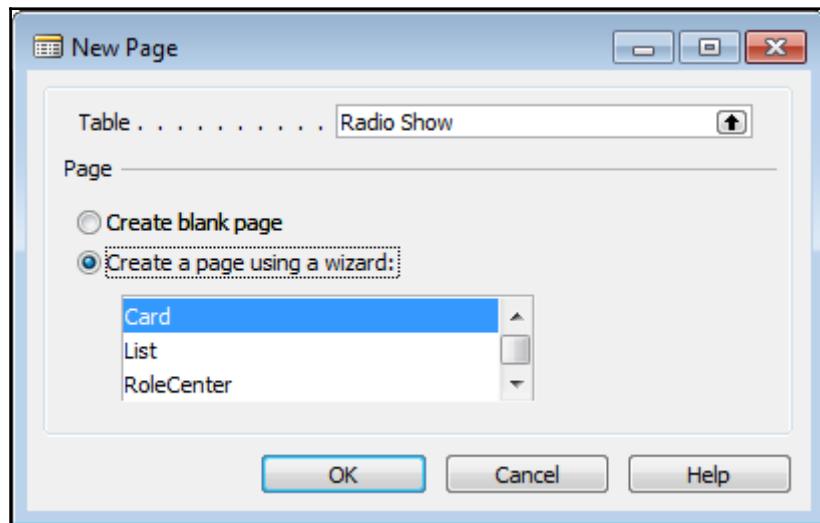


The availability of some capabilities and icons (such as **OneNote**) will depend on what software is installed on our development workstation. Close the preview of the List page and close the window or press *Esc* to save. Number the page 50000 and name the object Radio Show List:

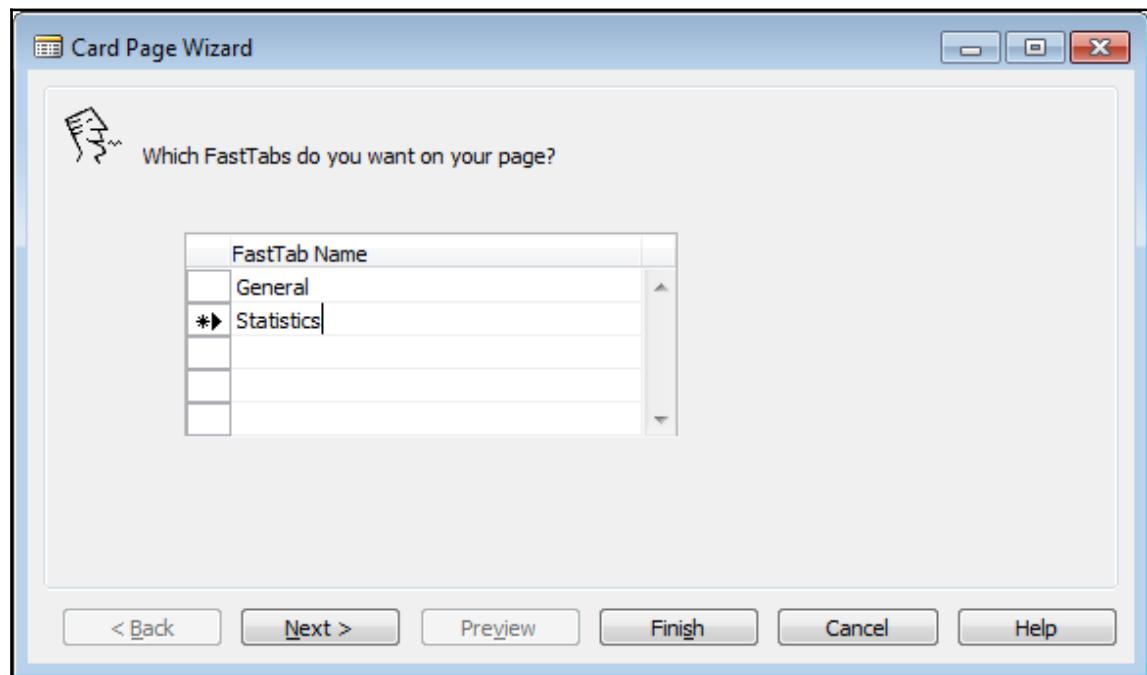


## Creating a Card page

Next, let's create a Card page. The Wizard process for a Card page is almost the same as for a List page, with an additional step. In **Object Designer**, with Page selected, click **New** again. Enter the same table (Radio Show) and make sure the **Create a page using a wizard:** option is selected and **Card** is highlighted:

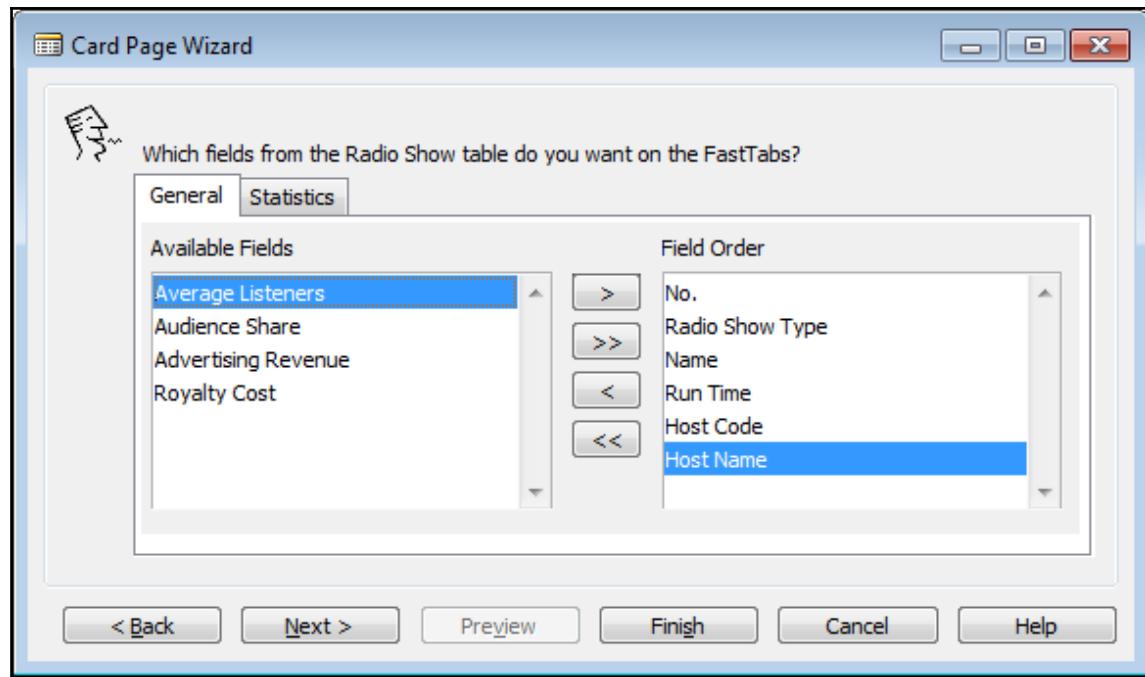


The next step in the wizard is specific to Card pages. It allows us to create FastTabs. These are the display tools that allow the user to expand or collapse window sections for ease of viewing. For our Radio Show card we will divide our table fields into two sections, **General** (primary key, description, resource information, and duration) and **Statistics** (data about the show):

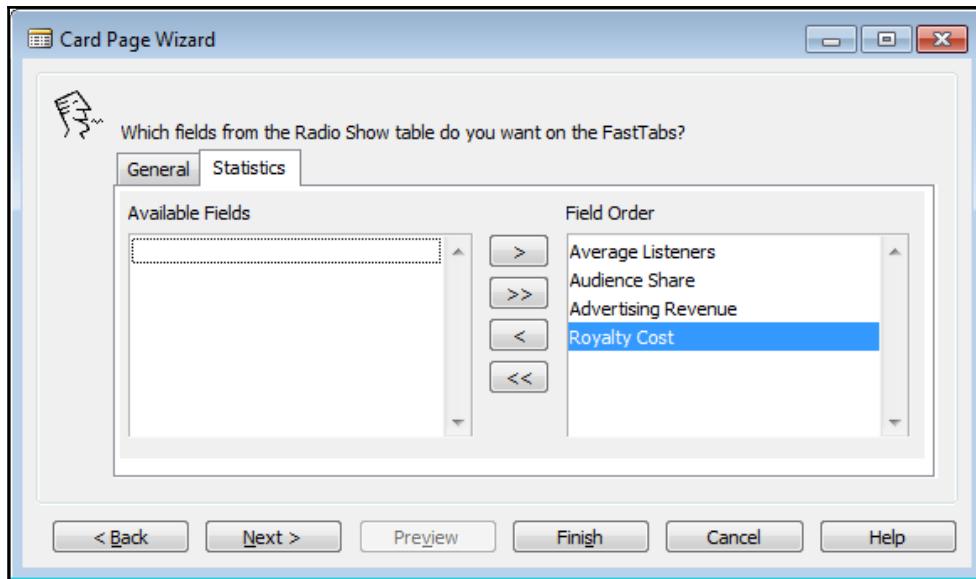


After defining the FastTab names, we must assign the data fields to the tabs on which they are to appear. We will populate the tabs based on the FastTab names we assigned. We can select the fields from the **Available Fields** list and assign the order of appearance as we did in the List Page Wizard. Click on the **Next >** button to proceed.

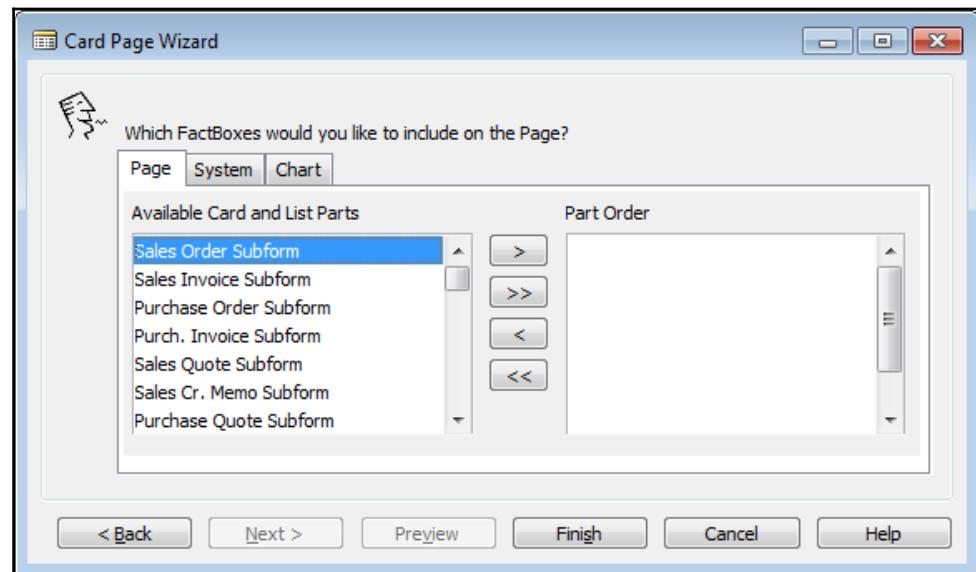
For the **General** FastTab, select the following fields: **No.**, **Show Code**, **Name**, **Run Time**, **Host Code**, and **Host Name**, as shown in the following screenshot:



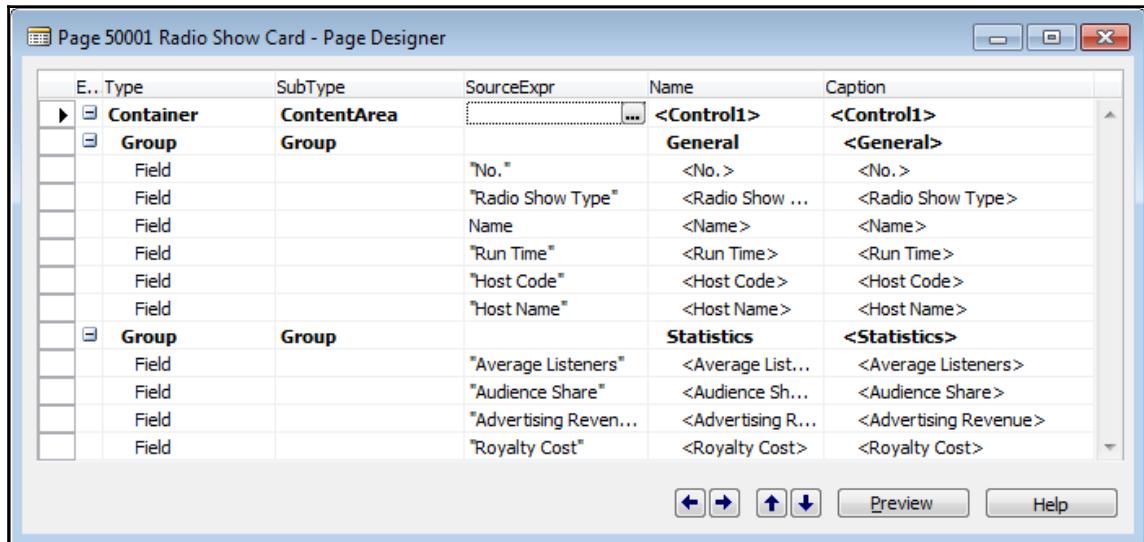
Click on the **Statistics** tab to populate the **Statistics** FastTab, select **Average Listeners**, **Audience Share**, **Advertising Revenue**, and **Royalty Cost**:



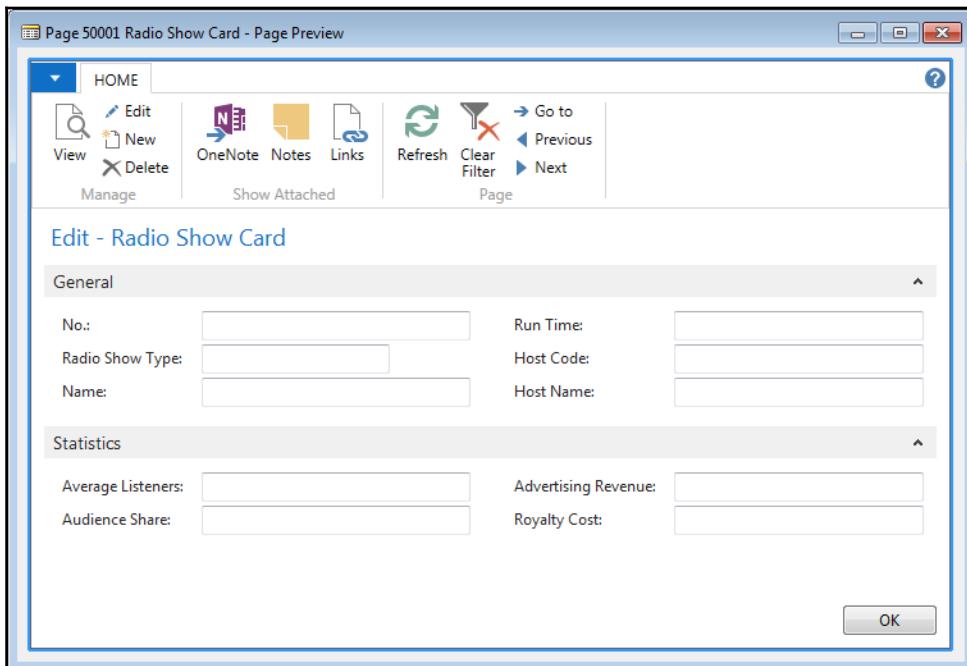
The last **Card Page Wizard** step is to choose from the available Subforms (Subpages), System FactBoxes, and Charts. If we decide later we want any of those, we will add them using the Page Designer:



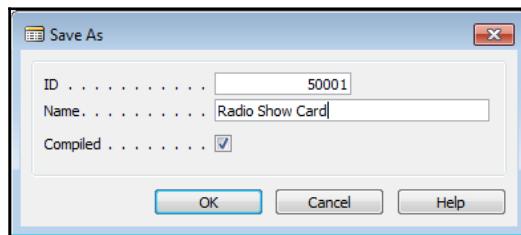
Click **Finish** to view the generated code in the Page Designer:



Click the **Preview** button to show a view-only display of the Card page:



Exit the preview and Page Designer and save the page as **ID** as 50001, and **Name** as Radio Show Card:

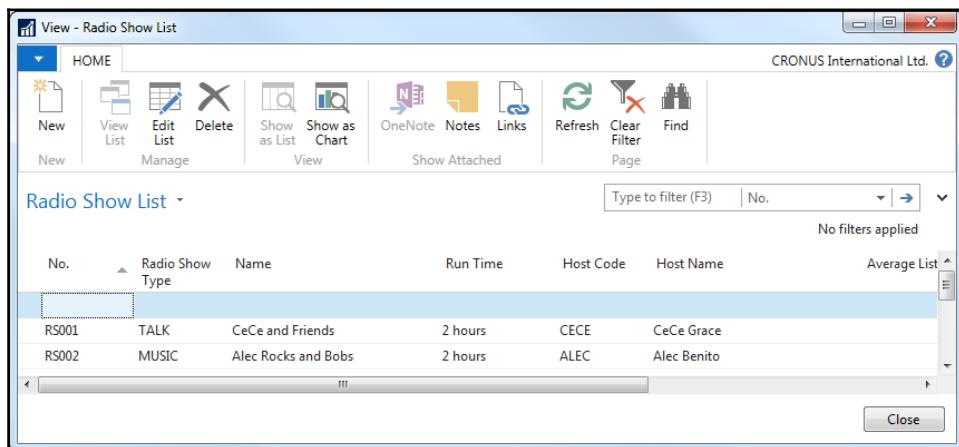


Later on, we can add an action to the List page which will link to the Card page for inserting and editing radio show records and also add the List page to the **Role Center** page for our radio station user.

## Creating some sample data

Even though we haven't added all the bells and whistles to our Radio Show table and pages, we can still use them to enter sample data. The **Radio Show List** page will be the easiest to use for this.

In **Object Designer**, with pages selected, highlight page **50000 - Radio Show List**, and click **Run**. Then click the **New** icon on the ribbon. An empty line will open, where we can enter our sample data. Of course, since our table is very basic at this point, without any validation functionality, table references, function calls, and so on, we will have to be creative (and careful) to enter all the individual data fields accurately and completely without guidance:



The screenshot shows the 'View - Radio Show List' page in Microsoft Dynamics NAV. The ribbon has tabs for HOME, New, View List, Edit List, Delete, Show as List, Show as Chart, OneNote, Notes, Links, Refresh, Clear Filter, Find, and Page. The main area displays a table with columns: No., Radio Show Type, Name, Run Time, Host Code, Host Name, and Average List. There are two entries:

No.	Radio Show Type	Name	Run Time	Host Code	Host Name	Average List
RS001	TALK	CeCe and Friends	2 hours	CECE	CeCe Grace	
RS002	MUSIC	Alec Rocks and Bobs	2 hours	ALEC	Alec Benito	

Enter the data shown in the following table so we can see what the page looks like when it contains data. Later on, after we add more capabilities to our table and pages, some fields will validate, and some will be either automatically entered or available on a lookup basis. But for now, simply key in each field value. If the data we key in now conflicts with the validations we create later (such as data in referenced tables), we may have to delete this test data and enter new test data later:

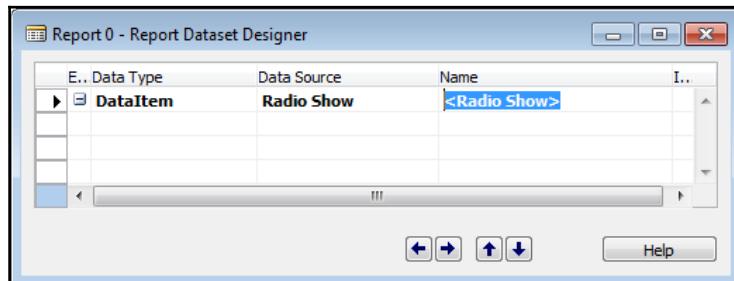


We will use the testability framework for automated testing later in this book. A test codeunit is provided in the downloads, but for now, it is recommended to key in the data manually.

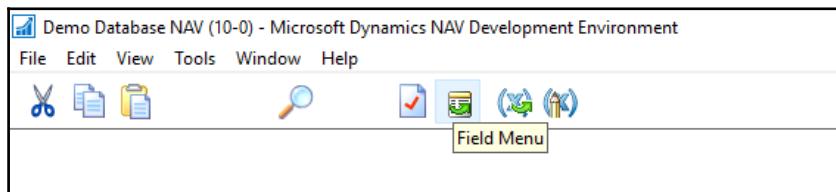
No.	Radio Show Type	Description	Host Code	Host Name	Run Time
RS001	TALK	CeCe and Friends	CECE	CeCe Grace	2 hours
RS002	MUSIC	Alec Rocks and Bops	ALEC	Alec Benito	2 hours
RS003	CALL-IN	Ask Cole!	COLE	Cole Henry	2 hours
RS004	CALL-IN	What do you think?	WESLEY	Wesley Ernest	1 hour
RS005	MUSIC	Quiet Times	SASKIA	Saskia Mae	3 hours
RS006	NEWS	World News	DAAN	Daan White	1 hour
RS007	ROCK	Rock Classics	JOSEPH	Josephine Perisic	2 hours

## Creating a list report

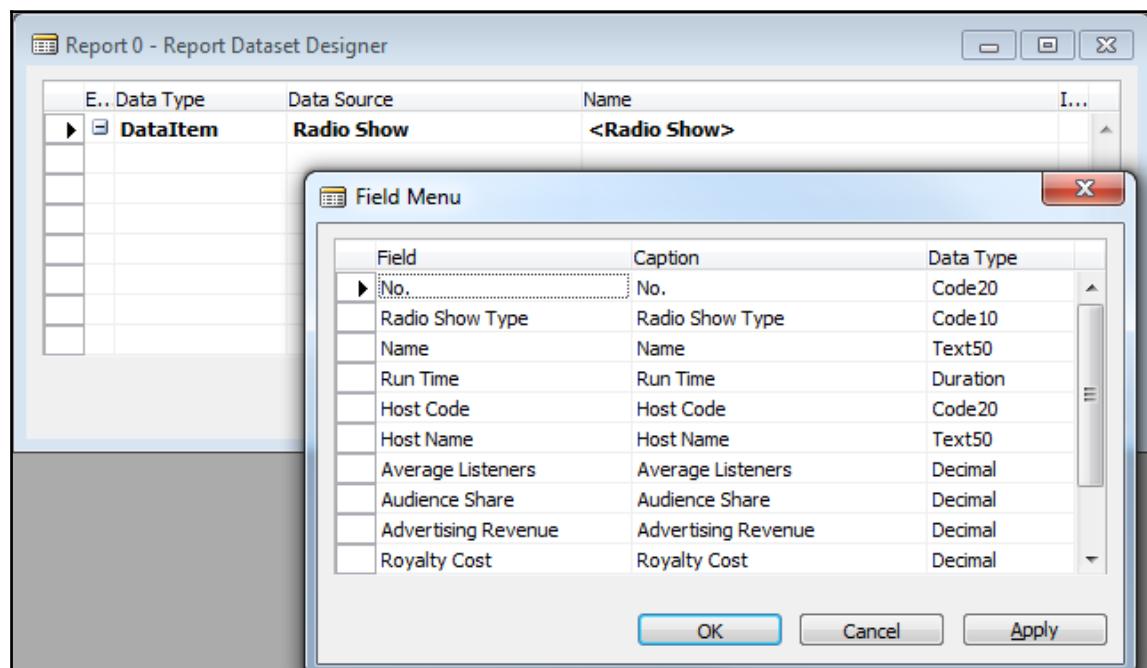
Open Object Designer, select **Report**, and click **New**. The Report Dataset Designer is empty when it displays, so we need to add a **Data Source** (table) to the first blank row. Type 50000 or **Radio Show** into the **Data Source** column:



To add multiple data fields from the table, we can use the **Field Menu**, which is accessed via the icon on the toolbar or the **View | Field Menu** option. The Field Menu will show a list of all the fields in the Radio Show table:



Highlight the first six fields in the **Field Menu**. Then click on next blank line in the Report Dataset Designer:



A confirmation box will appear, asking if we want to add the fields selected. Click **Yes**:

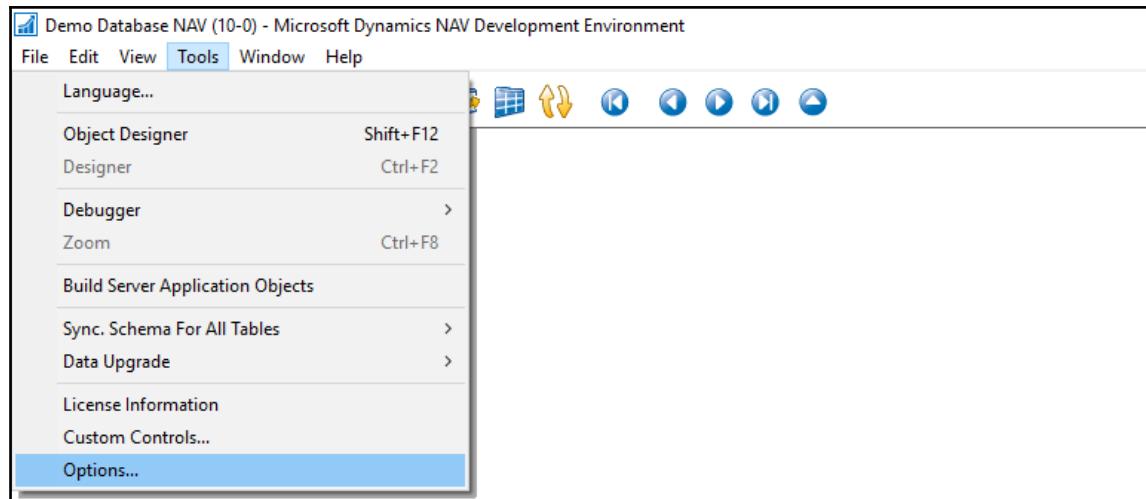


The fields will appear in the Report Dataset Designer without having to type them in manually:

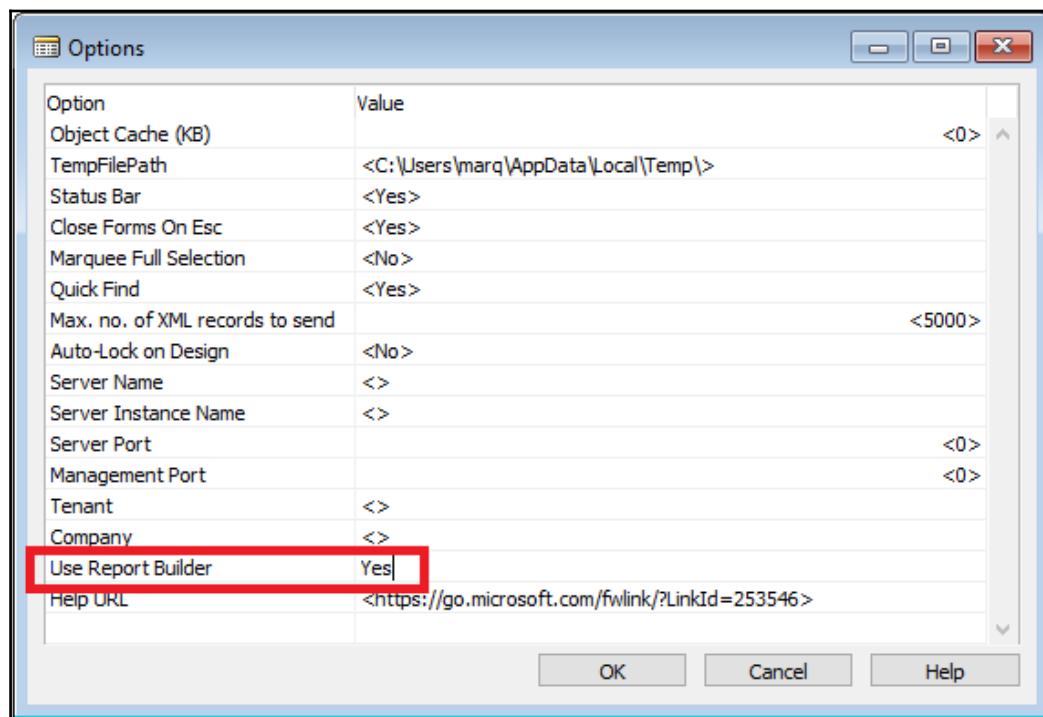
The screenshot shows the "Report 0 - Report Dataset Designer" window. It displays a table of selected fields from a "Radio Show" data source. The columns are labeled "E.. Data Type", "Data Source", "Name", and "Include Cap...". The table rows are as follows:

E.. Data Type	Data Source	Name	Include Cap...
▶ DataItem	Radio Show	<Radio Show>	<input type="checkbox"/>
Column	"Radio Show"."No."	No_RadioShow	
Column	"Radio Show"."Radio Show Type"	RadioShowType_RadioShow	
Column	"Radio Show".Name	Name_RadioShow	
Column	"Radio Show".Run Time"	RunTime_RadioShow	
Column	"Radio Show".Host Code"	HostCode_RadioShow	
Column	"Radio Show".Host Name"	HostName_RadioShow	

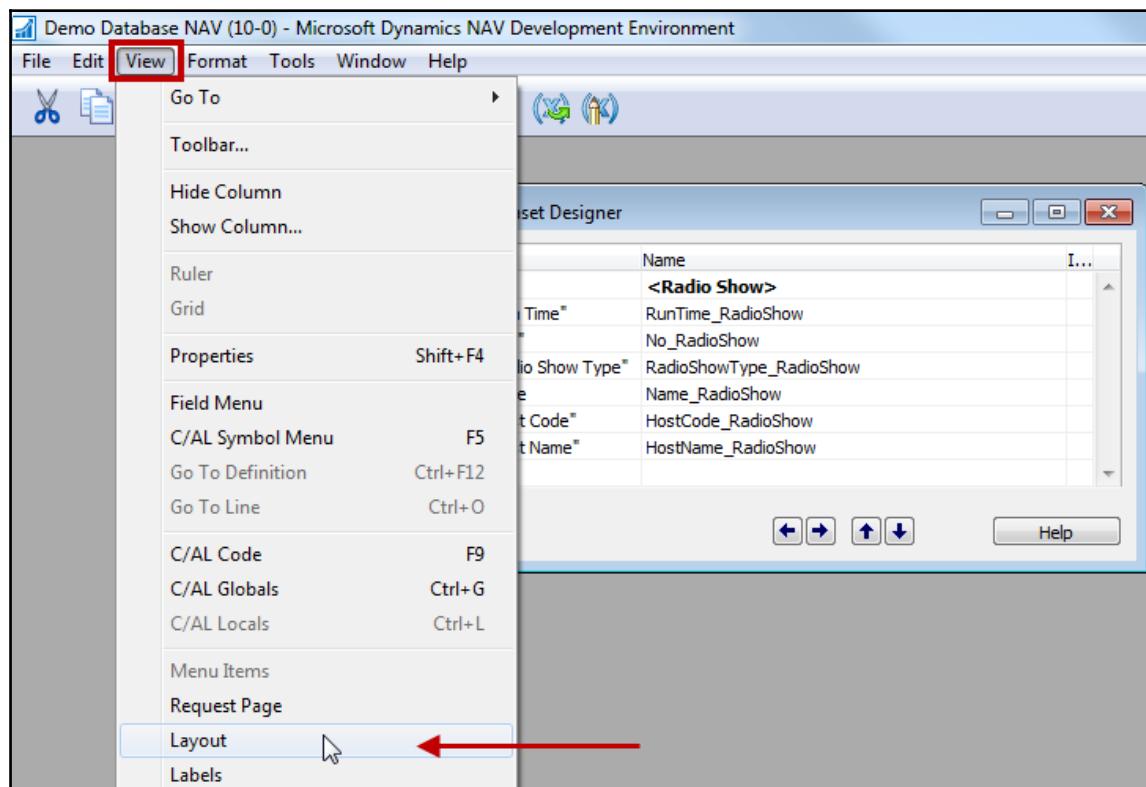
There are two options for RDLC report layout development tools. They are the current version of Visual Studio 2015 Community Edition or the SQL Server Report Builder (which can be downloaded here: <https://www.visualstudio.com/vs/community/>). NAV defaults to using Visual Studio, which we will use in this book. If the free SQL Server Report Builder is installed, it can be activated for NAV 2017 by accessing the **Options...** screen from the **Tools** menu option:



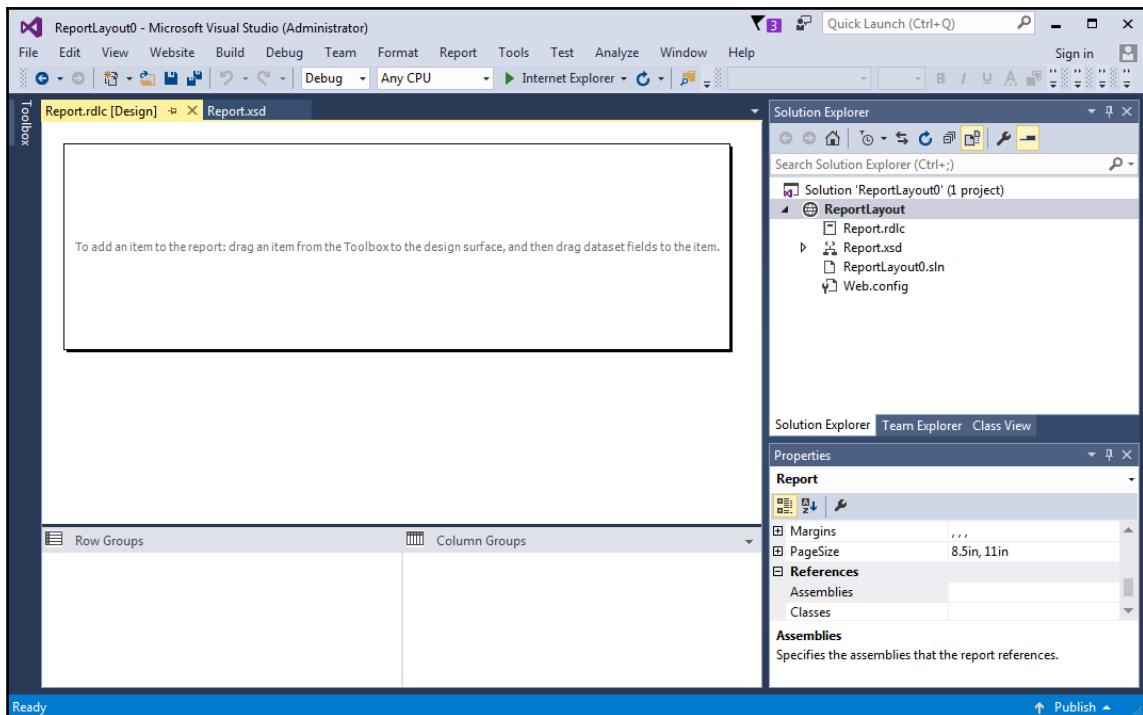
On the **Options** screen, set **Use Report Builder** to Yes:



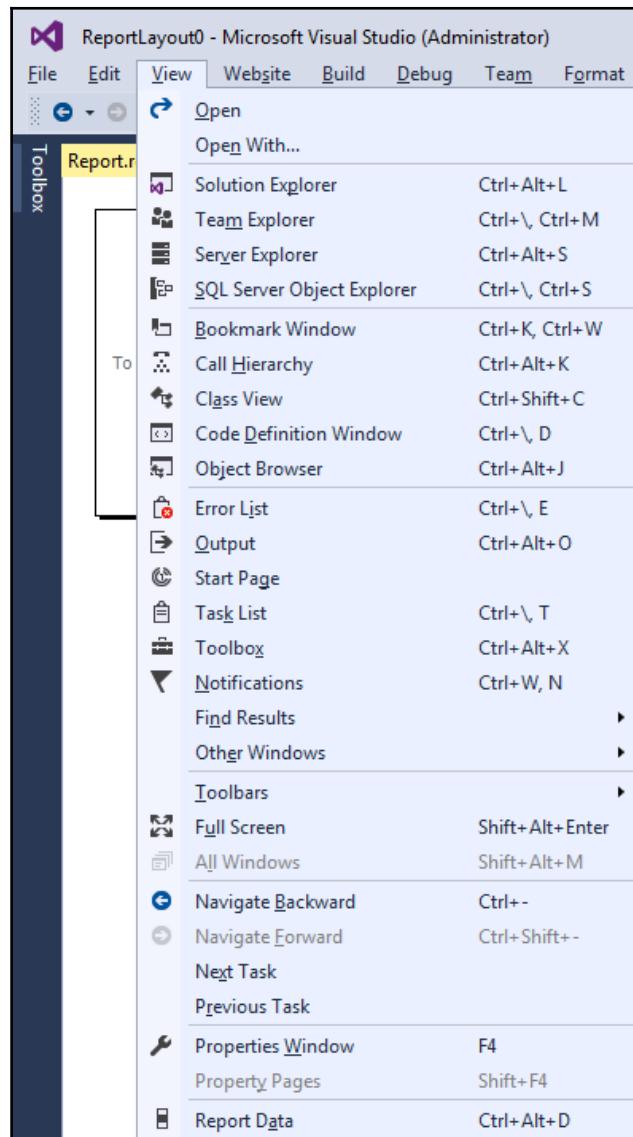
Click on **View | Layout** to proceed to the chosen report layout tool:



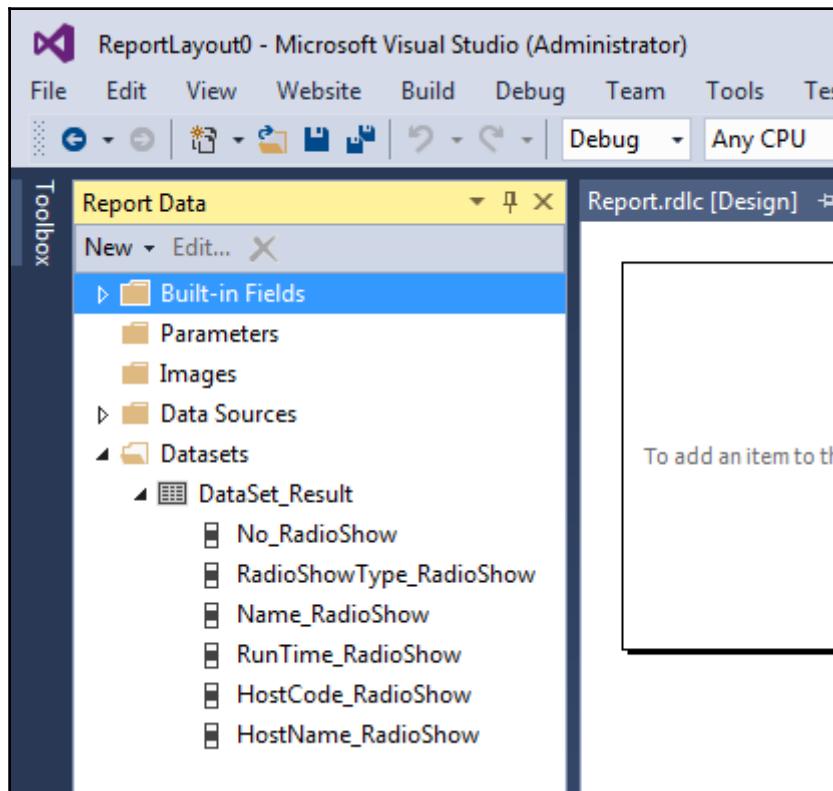
The RDLC report layout tool opens with a blank design surface and no dataset controls visible. Unlike the Page Designer, there is no report wizard to help with the layout of a new report. All the report layout design work must start from scratch with a blank design surface:



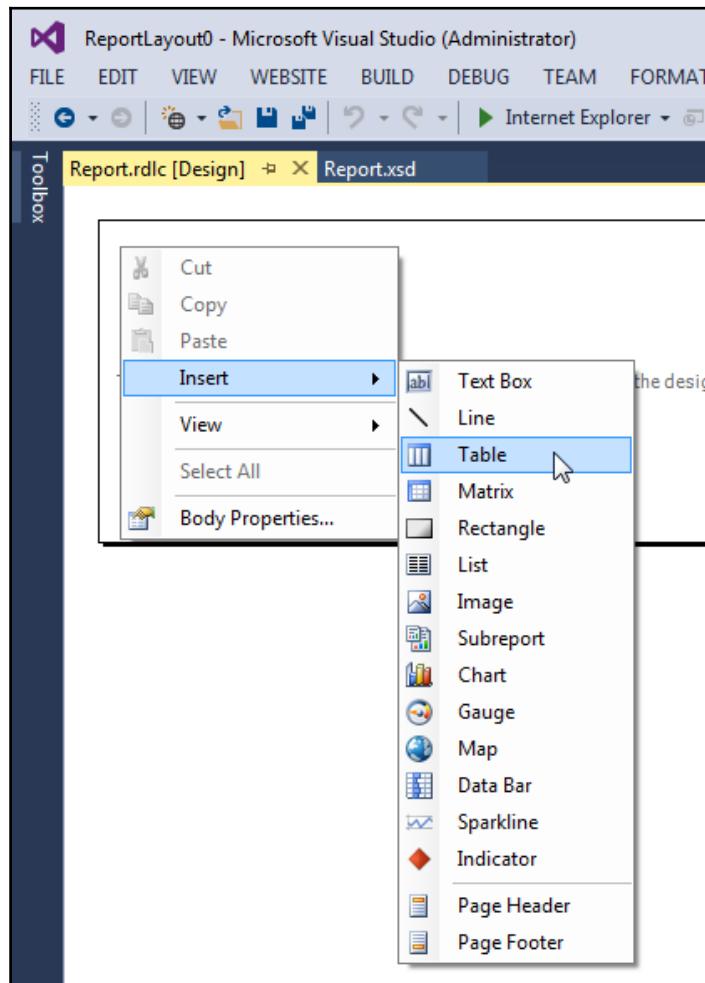
To show the dataset available from NAV, click on **View** and select **Report Data** (the last item on the list):



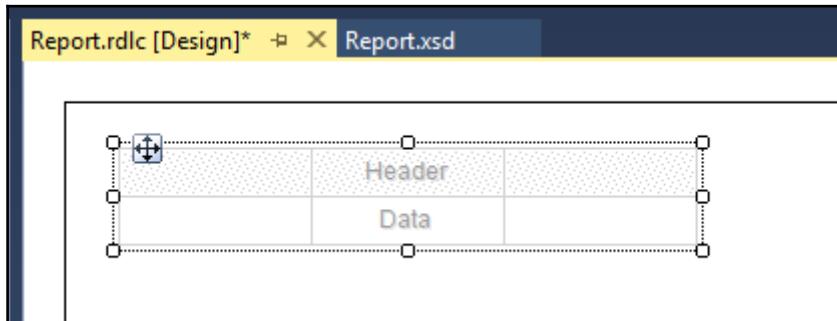
A new **Report Data** pane will show on the left of the Visual Studio layout window:



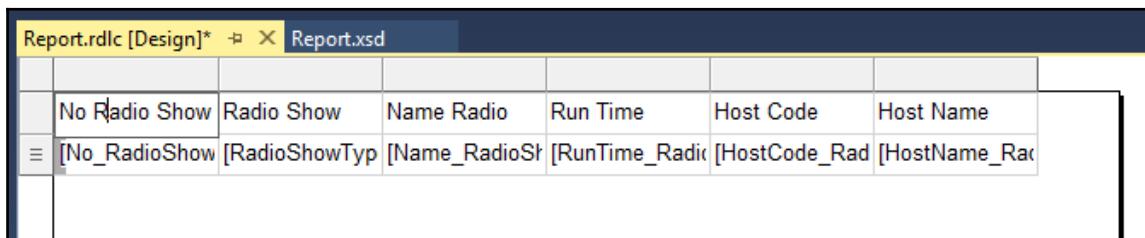
To create our simple list, we will insert a simple table object (a data region with a fixed number of columns but a variable number of rows) in the design surface. Right-click anywhere on the design surface and expand the **Insert** sub-menu to view the tools available on the report. Click the **Table** tool object, then use drag-and-drop to bring a control from the toolbox to the design surface:



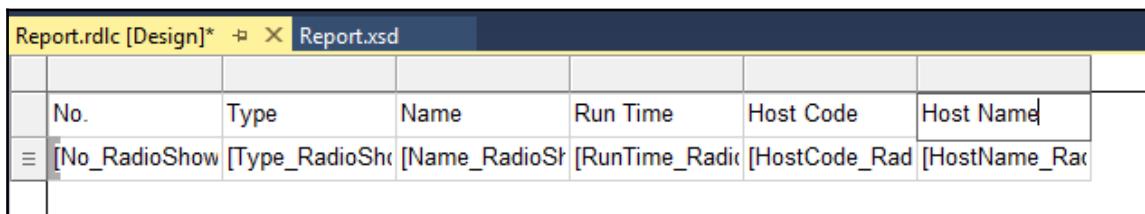
The table layout object defaults to three columns with a header row (repeated once) and a data row (repeated for each row of data retrieved from NAV):



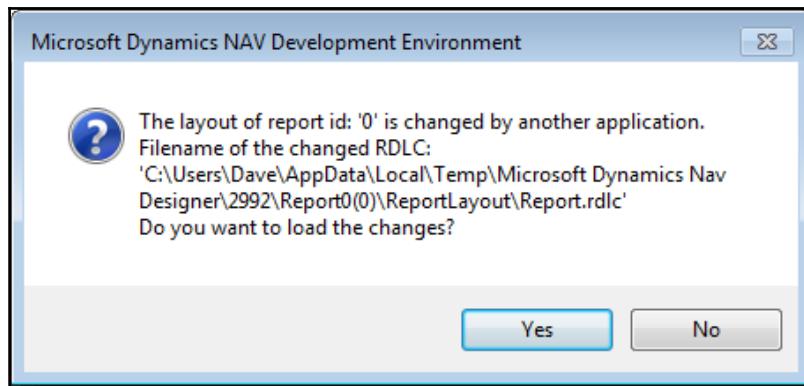
Drag and drop each of the six elements in the `DataSet_Result` into columns in the table object. To add additional columns, right-click on the table object header and select **Insert Column** (we could also drag-and-drop a field from the dataset to the table). The caption with the basic format of *Field Name Table Name* will default into the header row:



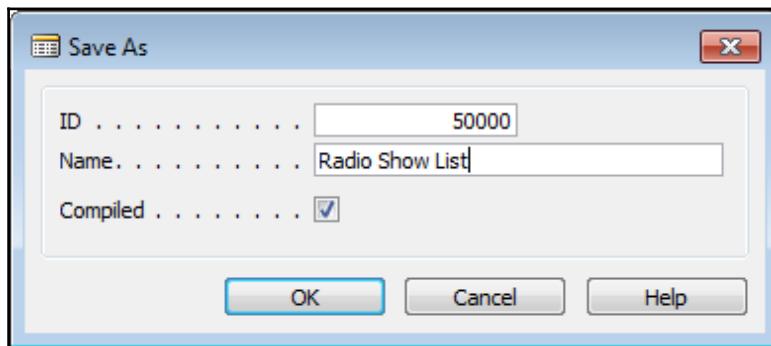
Let's do a little clean-up in the header row by making the captions look like they do in standard NAV reports by manually typing in the field names:



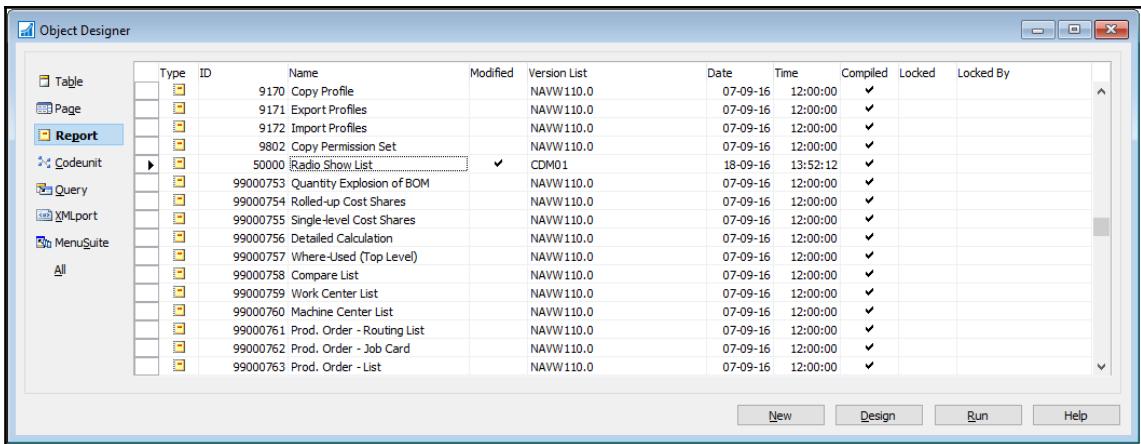
We will save our work by clicking on **File | Save All** (or *Ctrl + Shift + S*) then exit Visual Studio (**File | Exit** or *Alt + F4*). Back in NAV's **Object Designer**, we will exit the report or click **File | Save**; then, we'll need to respond to two confirmation boxes. The first asks if we want to save the report layout from Visual Studio. This allows us to load the RDLC report layout XML into the NAV database report object. Click **Yes**:



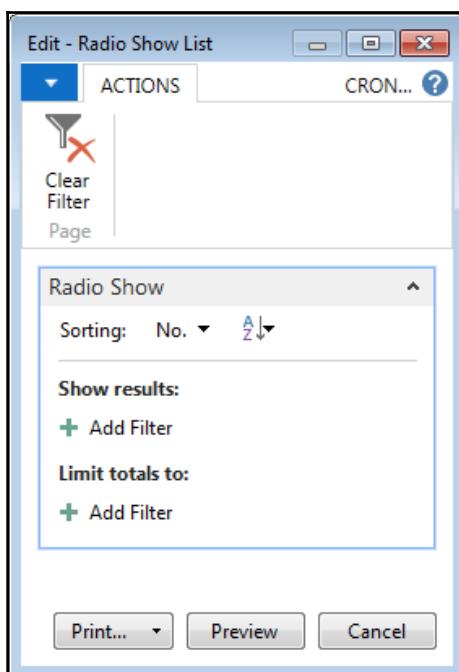
This is followed by the second confirmation screen. Enter **50000** for the **ID**, and **Radio Show List** the **Name** the report Radio Show List. Click **OK** to save:



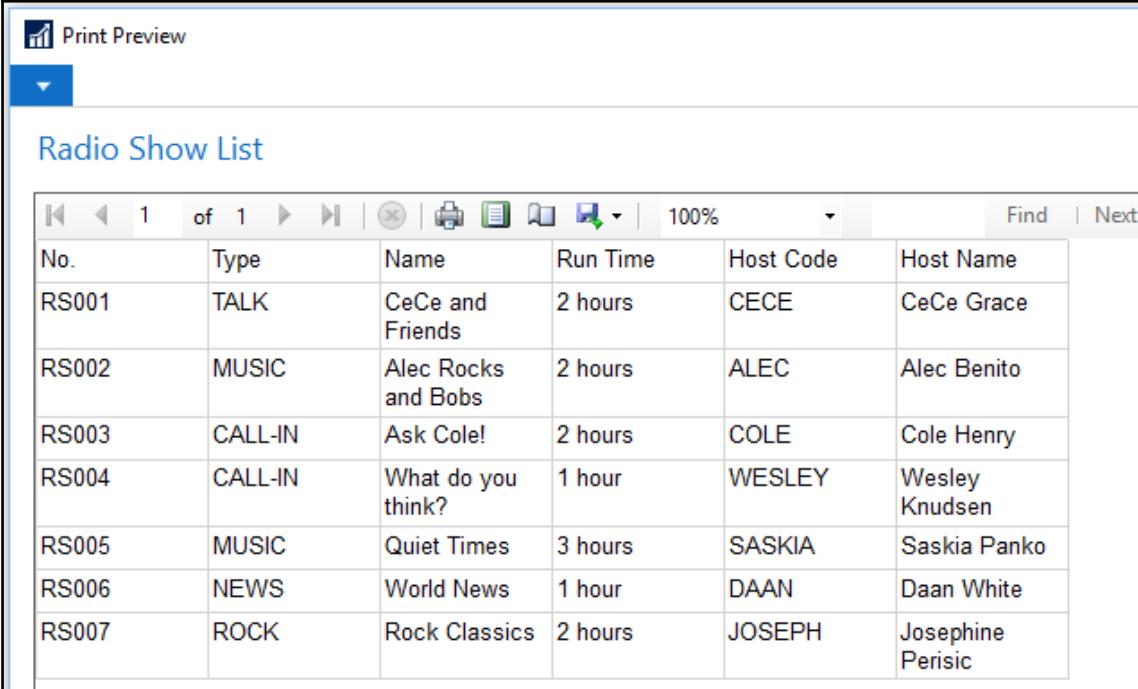
To view the report, make sure the new report object is selected, then click **Run** at the bottom of the **Object Designer** screen:



An RTC instance will open with the Report Request Page showing. This is where the user can set filters, choose a sort sequence, and choose print options:



Click on **Preview** to display the report onscreen. The report will show our simple table layout with the fixed definition column captions showing exactly as we typed them:



No.	Type	Name	Run Time	Host Code	Host Name
RS001	TALK	CeCe and Friends	2 hours	CECE	CeCe Grace
RS002	MUSIC	Alec Rocks and Bobs	2 hours	ALEC	Alec Benito
RS003	CALL-IN	Ask Cole!	2 hours	COLE	Cole Henry
RS004	CALL-IN	What do you think?	1 hour	WESLEY	Wesley Knudsen
RS005	MUSIC	Quiet Times	3 hours	SASKIA	Saskia Panko
RS006	NEWS	World News	1 hour	DAAN	Daan White
RS007	ROCK	Rock Classics	2 hours	JOSEPH	Josephine Perisic

Much more to come. All we've done so far is scratch the surface. But you should have a pretty good overview of the development process for NAV 2017.



You will be in especially good shape if you've been able to follow along in your own system, doing a little experimenting along the way.

## Other NAV object types

Let's finish up our introductory review of NAV's object types.

## Codeunits

A codeunit is a container for **chunks** of C/AL code to be called from other objects. These chunks of code are called Functions. Functions can be called from any of the other NAV object types that can contain C/AL code. Codeunits can also be exposed (published) as Web Services. This allows the functions within a published codeunit to be invoked by external routines.

Codeunits are 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, such as pages, reports, and so on.

Codeunits act only as a container for C/AL coded functions. They have no auxiliary functions, no method of direct user interaction, and no pre-defined processing. Even if we are creating only one or two functions and they are closely related to the primary activity of a particular object, if these functions are needed from both inside and outside of the report, the best practice is still to locate those functions in a codeunit. For more guidance, see *NAV Design Pattern of the Week - the Hooks Pattern*

at <http://blogs.msdn.com/b/nav/archive/2014/03/16/nav-design-pattern-of-the-week-the-hooks-pattern.aspx>.

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 some aspect of **Manufacturing** or some aspect of **Warehouse** management). Many of these can be found by filtering the Codeunit Names on the Management and Mgt strings (the same could be said for some of the tables with the string Buffer in their name). When we customize a system, we should create our own function library codeunits to consolidate our customizations and make software maintenance easier. Some developers create their own libraries of favorite special functions and include a function library codeunit in systems on which they work.

If a codeunit is structured very simply and can operate in a standalone 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. However, most codeunits are more complex and must be tested by a calling routine.

## Queries

Queries are objects whose purpose is to create extracted sets of data from the NAV database and do so very efficiently. NAV 2017 Queries translate directly into T-SQL query statements and run on the SQL Server side rather than on the NAV Service Tier. A query can extract data from a single table or multiple tables. In the process of extracting data, a query can do different types of join (Inner Join, Outer Join, or Cross Join), can filter, can calculate FlowFields (special NAV calculations that are discussed in detail in *Chapter 3, Data Types and Fields*), can sort, and can create sums and averages. Queries obey the NAV data structure business logic.

The output of a query can be a CSV file (useful for Excel charts), an XML file (for charts or external applications), or an Odata file for a web service. Queries can be published for web service access similarly to pages and codeunits. The results of a query can also be viewed by pages (as described in *Chapter 5, Queries and Reports*) and Cues (as described in the *Walkthrough: Creating a Cue Based on a Normal Field and a Query* documentation section (<https://msdn.microsoft.com/en-us/dynamics-nav/walkthrough--creating-a-cue-based-on-a-normal-field-and-a-query>)), but are especially powerful when output to charts. With a little creativity, a query can also be used to feed data to a page or report via use of a temporary table to hold the query results.

## MenuSuites

MenuSuites are the objects that are displayed in the Navigation Menus. They differ considerably from the other object types we have discussed earlier because they have a completely different structure and they are maintained differently. MenuSuite entries do not contain triggers. The only customization that can be done with MenuSuites is to add, delete, or edit menu entries that are made up of a small set of properties.

In the RTC, the data in the MenuSuites object is presented in the **Departments** page.

## XMLports

XMLports are a tool for importing and exporting data. XMLports handle both XML structured data and other external text data formats. **XML (eXtensible Markup Language)** is the de facto standard for exchanging data between dissimilar systems. For example, XMLports could be used to communicate between our NAV ERP system and our accounting firm's financial analysis and tax preparation system.

XML is designed to be extensible, which means that we can create or extend the definition as long as we communicate the defined XML format to our correspondents. There is a standard set of syntax rules to which XML formats must conform. A lot of 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.

The non-XML text data files handled by XMLports fall into two categories. One is known as **comma-separated value** or **comma-delimited** files (usually having a .csv file extension). Of course, the delimiters don't have to be commas. The other category is fixed format, in which the length and relative position of each field is pre-defined.

XMLports can contain C/AL logic for any type of appropriate data manipulation, either when importing or exporting. Functions such as editing, validating, combining, filtering, and so on can be applied to the data as it passes through an XMLport.

## Development backups and documentation

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

When we are working within the **Object Designer**, we 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, to recover the original version of one or more objects.

NAV 2017 introduces **Windows PowerShell Cmdlets** that support backing up data to **NAVData** files. Complementary Cmdlets support getting information or selectively retrieving data from previously created NAVData files. Although these tools promise to be very handy for repetitive development testing, they are challenging (or worse) to use in an environment of changing table or field definitions.

When objects are exported to text files, we can use a standard text editor to read or even change them. If, for example, we wanted to change all the instances of the `Customer` field name to `Patient`, we might export all the objects to text and execute a mass Find and Replace. Making such code changes in a text copy of an object is subject to a high possibility of error, as we won't have any of the many safety features of the C/SIDE environment to limit what we can do.

Internal documentation (that is, inside C/SIDE) 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 (or set of changes) is made in an object, a notation should be added to the version list.

The second area for documentation is the Documentation trigger that appears in every object type except MenuSuites. The Documentation trigger is at the top of the object and is the recommended location for noting a relatively complete description of any changes that have been made to the object. Such descriptions should include a brief description of the purpose of the change as well as technical information.

The third area we can place documentation is in line with modified C/AL code. Individual comment lines can be created by starting the line with double forward slashes, //. Whole sections of comments (or commented out code) can be created by starting and ending the section with a pair of curly braces { }. Depending on the type of object and the nature of the specific changes, we should generally annotate each change inline with forward slashes rather than with curly braces wherever the code is touched, so all the changes can be easily identified by the next developer. Although the use of curly braces is permitted by the software, Microsoft warns of potential conflicts with Powershell upgrade processes.

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

## Review questions

1. An ERP system such as NAV 2017 includes a number of functional areas. Which of the following are part of NAV 2017? Choose four:
  - a) Manufacturing
  - b) Order processing
  - c) Planning
  - d) Computer Aided Design (CAD)
  - e) General accounting

2. New functionality in NAV 2017 includes Choose three:
  - a) Tablet client
  - b) Multi-language
  - c) Document e-mailing
  - d) Spell checker
  - e) Mandatory fields
3. NAV 2017 development is done in the C/SIDE IDE and Visual Studio. True or false?
4. Match the following table types and descriptions for NAV:
  - a) Journals Audit trail
  - b) Ledgers Validation process
  - c) Register Invoice
  - d) Document Transaction entries
  - e) Posting History
5. iPads can be used with NAV 2017 to display the RTC. True or false?
6. Which of the following describe NAV 2017? Choose two:
  - a) Customizable
  - b) Includes a Storefront module
  - c) Object-based
  - d) C# IDE
  - e) Object-oriented
7. What are the seven NAV 2017 object types?
8. All NAV objects except XMLports can contain C/AL code. True or false?
9. NAV 2017 includes support for publishing objects as web services. True or false?

10. The home page for a NAV 2017 User is called what? Choose one:
  - a) Role Home
  - b) Home Center
  - c) Main Page
  - d) Role Center
11. Page previews from the development environment can be used for data entry and maintenance. True or false?
12. Visual Studio is used in NAV 2017 for what work? Choose one:
  - a) Report data definition
  - b) Report layout
  - c) Role Center design
  - d) Query processing
13. Codeunits are the only NAV 2017 objects that can contain functions. True or false?
14. Query output can be used as a data item for reports. True or false?
15. C/AL and C/SIDE are required for NAV 2017 development. True or false?
16. What object number range is available for assignment to customer-specific objects? Choose two:
  - a) 20 - 500
  - b) 50,000 - 60,000
  - c) 150,000 - 200,000
  - d) 50,000 - 99,999
  - e) 10,000 - 100,000
17. XMLports can only process XML formatted data. True or false?
18. The work date can only be changed by the system administrator. True or false?

19. A design pattern is which of the following? Choose two:
  - a) Reusable code
  - b) Stripes and plaid together
  - c) A proven way to solve a common problem
  - d) UI guidelines
20. NAV 2017 reports are often generated automatically through the use of a wizard.  
True or false?

## Summary

In this chapter, we covered some basic definitions of terms related to NAV and C/SIDE. We followed with the introduction of the seven NAV objects types (Tables, Pages, Reports, Codeunits, Queries, MenuSuites, and XMLports). We introduced table, page, and report creation through review and hands-on use, beginning a NAV application for the programming management of the WTDU Radio Show. Finally, we looked briefly at the tools that we use to integrate with external entities and discussed how different types of backups and documentation are handled in C/SIDE. Now that we have covered the basics, we will dive into the detail of the primary object types in the next few chapters.

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

# 2

# Tables

*"If you don't know where you are going, you might wind up someplace else."*

- Yogi Berra

*"Successful design is anticipating the anticipations of others."*

- John Maynard Keynes

The foundation of any system is the data structure definition. In NAV, the building blocks of this foundation are the tables and the individual data fields that the tables contain. Once the functional analysis and process definition has been completed, any new design work must begin with the data structure. For NAV, that means the tables and their contents.

A NAV table includes much more than just the data fields and keys. A NAV table definition also includes data validation rules, processing rules, business rules, and logic to ensure referential integrity. The rules are in the form of properties and C/AL code.



For object-oriented developers, a Dynamics NAV table is best compared to a class with methods and properties stored one-to-one as a table on SQL Server.

In this chapter, we will learn about the structure and creation of tables. Details regarding fields and the components of tables will be covered in the following chapter. Our topics in this chapter include the following:

- An overview of tables, including Properties, Triggers, Keys, SumIndexFields, and Field Groups
- Enhancing our scenario application - creating and modifying tables
- Types of table - Fully Modifiable, Content Modifiable, and Read-Only tables

# An overview of tables

There is a distinction between the table (data definition and data container) and the data (the contents of a table). The table definition describes the identification information, data structure, validation rules, storage, and retrieval of the data that is stored in the table (container). The definition is defined by the design and can only be changed by a developer. The data is the variable content that originates from user activities. The place where we can see the data explicitly referenced independently of the table as a definition of structure is in the Permissions setup data. In the following screenshot, the data is formally referred to as **Table Data**:

The screenshot shows a Microsoft Dynamics 365 application window titled 'Edit - Permission'. The ribbon bar has 'HOME' selected. Below the ribbon are buttons for 'New', 'View List', 'Edit List', 'Delete', 'Refresh', and 'Find'. The main area is titled 'Permission' and shows a list of table data with columns: Role ID, Role Name, Object Type, Object ID, Object Name, Read Permission, Insert Permission, Modify Permission, Delete Permission, and Execute Permission. A filter bar at the top right allows 'Type to filter (F3)' and 'Role ID'. A message 'No filters applied' is displayed. The data table contains 11 rows, all assigned to the 'BASIC' role:

Role ID	Role Name	Object Type	Object ID	Object Name	Read Permission	Insert Permission	Modify Permission	Delete Permission	Execute Permission
BASIC	Basic User (All Inclu... Table Data	Table Data	4	Currency	Yes	Indirect	Indirect		
BASIC	Basic User (All Inclu... Table Data	Table Data	7	Standard Text	Yes				
BASIC	Basic User (All Inclu... Table Data	Table Data	14	Location	Yes	Indirect	Indirect		
BASIC	Basic User (All Inclu... Table Data	Table Data	17	G/L Entry	Yes				
BASIC	Basic User (All Inclu... Table Data	Table Data	18	Customer	Yes	Indirect	Indirect		
BASIC	Basic User (All Inclu... Table Data	Table Data	23	Vendor	Yes	Indirect	Indirect		
BASIC	Basic User (All Inclu... Table Data	Table Data	27	Item	Yes	Indirect	Indirect		
BASIC	Basic User (All Inclu... Table Data	Table Data	30	Item Translation	Yes	Indirect	Indirect		
BASIC	Basic User (All Inclu... Table Data	Table Data	32	Item Ledger Entry	Yes	Indirect	Indirect		
BASIC	Basic User (All Inclu... Table Data	Table Data	36	Sales Header	Yes	Indirect	Indirect		

At the bottom right of the grid is an 'OK' button.

The table is not the data, it is the definition of data contained in the table. Even so, we commonly refer to both the data and the table as if they were one and the same. That is what we will do in this book.

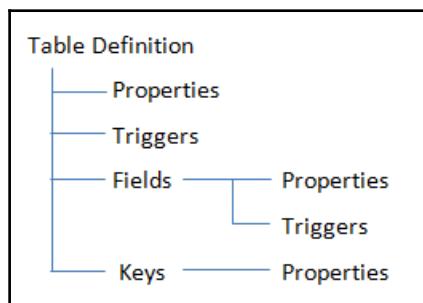
All permanent data must be stored in a table. All tables are defined by the developer working in the Development Environment. As much as possible, critical system design components should be embedded in the tables. Each table should include the code that controls what happens when records are added, changed, or deleted, as well as how data is validated when records are added or changed. That includes functions to maintain the aspects of referential integrity that are not automatically handled. The table object should also include the functions commonly used to manipulate the table and its data, whether for database maintenance or in support of business logic. In the cases where the business logic is either a modification applied to a standard (out-of-the-box) table or also is used elsewhere in the system, the code should be resident in a function library codeunit and called from the table.

The Table Designer in C/SIDE provides tools for the definition of the data structure within tables. We will explore these capabilities through examples and analysis of the structure of table objects. We find the approach of embedding control and business logic within the table object has a number of advantages:

- Clarity of design
- Centralization of rules for data constraints
- 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) and **Keys**. **Fields** also have **Properties** and **Triggers**. **Keys** also have **Properties**:



A table definition that takes full advantage of these capabilities reduces the effort required to construct other parts of the application. Good table design can significantly enhance the application's processing speed, efficiency, and flexibility.

A table can have the following:

- Up to 500 fields
- A defined record size of up to 8,000 bytes (with each field sized at its maximum)
- Up to 40 different keys

## Naming tables

There are standardized naming conventions defined for NAV, which we should follow. Names for tables and other objects should be as descriptive as possible, while keeping to a reasonable length. This makes our work more self-documenting.

Table names should always be singular. The table containing data about customers should not be named `Customers`, but `Customer`. The table we created for our WDTU Radio Station NAV enhancement was named `Radio Show`, even though it will contain data for all of WDTU's radio shows.

In general, we should always name a table so it is easy to identify the relationship between the table and the data it contains. For example, 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 all the lines for the order.

Additional information on table naming can be found in the old, but still useful, *C/AL Programming Guide*, which can be found on the Mergetool.com site



at <http://www.mergetool.com/oldmergetool/CALProgGuide.pdf>. These older documents may be obsolete in some areas. So, of course, we should always refer first to the Developer and IT Pro Help included in NAV and accessible from the Development Environment. The NAV 2017 Help is also available on MSDN at <https://msdn.microsoft.com/en-us/dynamics-nav/index>.

## Table numbering

There are no hard and fast rules for table numbering, except that we must only use the table object numbers that we are licensed to use. If all we have is the basic Table Designer rights, we are generally allowed to create tables numbered from 50000 to 50009 (unless our license was defined differently than the typical one). If we need more table objects, we can purchase licensing for table objects numbered up to 99999. **Independent Software Vendors (ISVs)** can purchase access to tables in other number ranges.

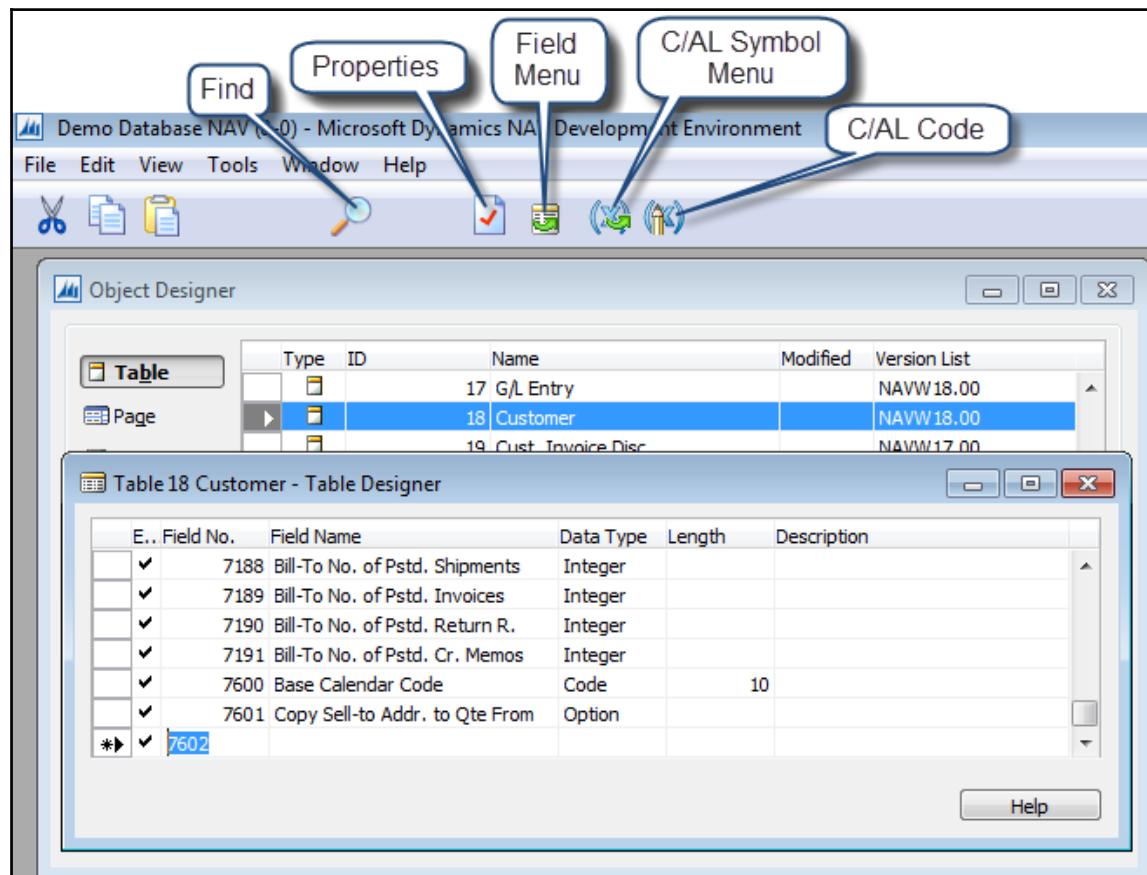
When creating several related tables, ideally we should assign them related numbers in sequential order. We should let common sense be our guide to assigning table numbers. It requires considerable effort to renumber tables containing data.

## Table properties

The first step in studying the internal construction of a table is to open it in Design mode. This is done as follows:

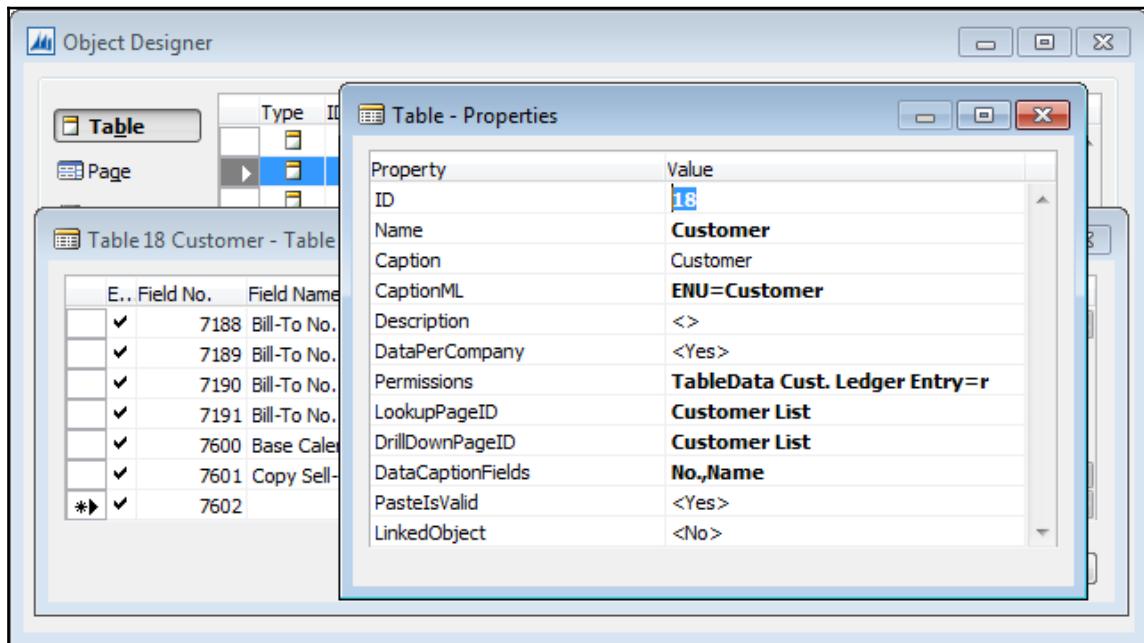
1. Open the Development Environment.
2. Click on the **Table** button in the left column of buttons.
3. Highlight the table to work on (in this case, **Table 18 Customer**).
4. Click on the **Design** button at the bottom-right of the screen.

We now have the Customer table open in the Table Designer in Design mode. In Chapter 1, *Introduction to NAV 2017*, we reviewed the function of the icons across the top of the Table Designer, but they are labeled in this screenshot as a memory aid:



We can access the properties of a table while viewing the table in Design mode. Place the cursor on the empty field line below all the fields (as shown in the preceding screenshot), and click on the Properties icon, or press **Shift + F4**, or navigate through **View | Properties**. If we access properties while the focus is on a filled field line, we will see the properties of that field (not the table).

This will take us to the **Table - Properties** display. The following screenshot is the **Table - Properties** display for the **Customer** table in the demonstration Cronus database:



The table properties are as follows:

- **ID:** This is the object number of the table.
- **Name:** This is used for internal identification of the table object and acts 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 (ENU).
- **CaptionML:** This defines the MultiLanguage caption for the table. For an extended discussion on the language capabilities of NAV, refer to the *Multilanguage Development* section (<https://msdn.microsoft.com/en-us/dynamics-nav/multilanguage-development>) in the online Developer and IT Pro Help.
- **Description:** This property is for optional documentation usage.
- **DataPerCompany:** This lets us 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 generated names of tables within SQL Server (not within NAV) are affected by this choice.

- **Permissions:** This allows us 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.
- **LookupPageID:** This allows us to define what page is the default for looking up data in this table.
- **DrillDownPageID:** This allows us to define what page is the default for drilling down into the supporting detail for data that is summarized in this table.
- **DataCaptionFields:** This allows us 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 title bar at the top of a page showing a customer record.
- **PasteIsValid:** This property (Paste Is Valid) determines if the users are allowed to paste data into the table.
- **LinkedObject:** This lets us 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. When this property is set to Yes, the **LinkedInTransactionProperty** becomes available. The **LinkedInTransactionProperty** should be set to No for any linkage to a SQL Server object outside the current database. The object being linked must have a SQL Server table definition that is compatible with the Microsoft Dynamics NAV table definition.
- **TableType:** This allows you to consume data from an OData web service such as Dynamics CRM or an External SQL as a table.

As developers, we most frequently deal with the **ID**, **Name**, **LookupPageID**, **DrillDownPageID** properties, **Caption**, **CaptionML** (outside the United States), **DataCaptionFields**, and **Permissions** properties. We rarely deal with the others.

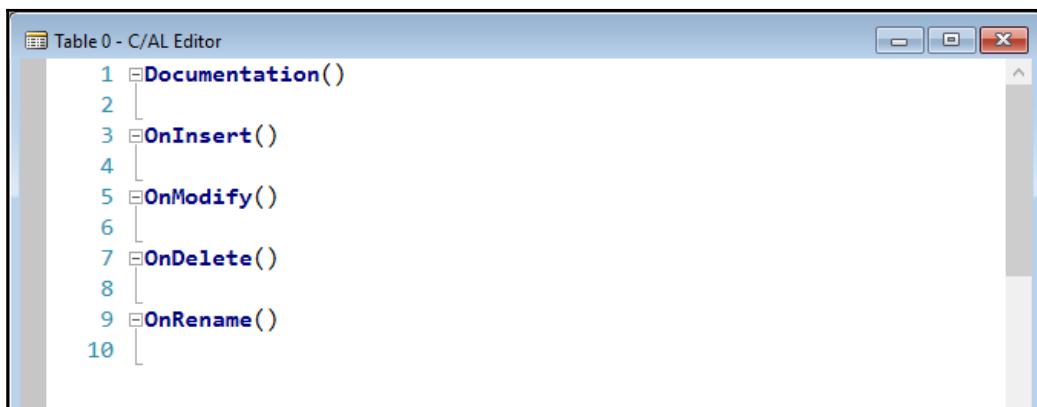
## Table triggers

To display the triggers with the table open in the Table Designer, click on the C/AL Code icon or F9 or View | C/AL Code. The first (top) table trigger is the Documentation trigger. This trigger is somewhat misleadingly named as it only serves as a location for needed documentation. No C/AL code in a Documentation trigger is executed. There are no syntax or format rules here, but we should follow a standard format of some type.

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

```
CD.02 - 2017-03-16 Change to track when new customer added  
- Added field 50012 "Start Date"
```

The Documentation trigger has the same appearance as the four other triggers in a table definition, shown in the following screenshot, each of which can contain C/AL code:



When shipping your solution as an extension as described in [Chapter 6, Introduction to C/SIDE and C/AL](#), the Documentation trigger is not allowed.

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 consistency such as referential integrity. We can even abort the intended action if data inconsistencies or conflicts are found.

These triggers are automatically invoked when record processing occurs as the result of user action. But when table data is changed by C/AL code or by a data import, the C/AL code or import process determines whether or not the code in the applicable trigger is executed.

- `OnInsert()`: This is executed when a new record is to be inserted in the table through the UI. (In general, new records are added when the last field of the primary key is completed and focus leaves that field. See the property in [Chapter 4, Pages- the Interactive Interface](#), on pages for an exception).

- **OnModify ()**: This is executed when a record is rewritten after the contents of any field other than a primary key field has been changed. The change is determined by comparing `xRec` (the image of the record prior to being changed) to `Rec` (the current record copy). During our development work, if we need to see what the **before** value of a record or field is, we can reference the contents of `xRec` and compare that to the equivalent portion of `Rec`. These variables (`Rec` and `xRec`) are System-Defined Variables.
- **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 part of the primary key is a Rename action. This maintains a level of referential integrity. Unlike some systems, NAV allows the primary key of any master record to be changed and automatically maintains all the affected foreign key references from other records.

There is an internal inconsistency in the handling of data integrity by NAV. On one hand, the Rename trigger automatically maintains one level of referential integrity when any part of the primary key is changed (that is, the record is **renamed**). This happens in a black box process, an internal process that we cannot see or touch.

However, if we delete a record, NAV doesn't automatically do anything to maintain referential integrity. For example, child records could be orphaned by a deletion, that is, left without any parent record. Or, if there are references in other records back to the deleted record, they could be left with no target. As developers, we are responsible for ensuring this part of referential integrity in our customizations.

When we write C/AL code in one object that updates data in another (table) object, we can control whether or not the applicable table update trigger fires (executes). For example, if we were adding a record to our `Radio Show` table and used the following C/AL code, the `OnInsert ()` trigger would fire:

```
RadioShow.INSERT(TRUE);
```

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

```
RadioShow.INSERT(FALSE);
```

Alternatively, you can use this:

```
RadioShow.INSERT;
```

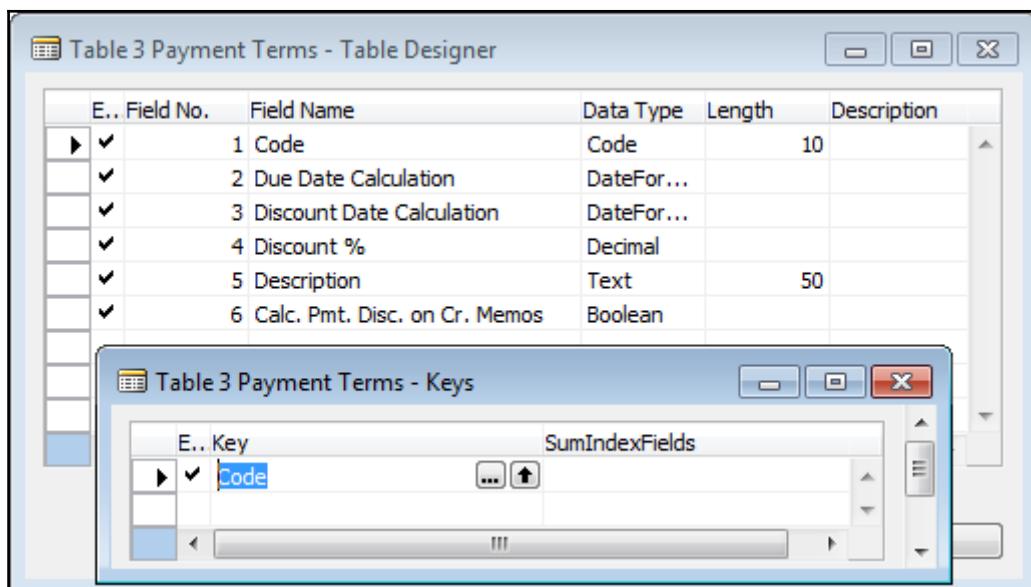
The automatic black box logic enforcing primary key uniqueness will happen whether or not the `OnInsert ()` trigger is fired.

## 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 a key field changes. Key cost is measured primarily in terms of increased index maintenance processing. There is also additional cost in terms of disk storage space (usually not significant) and additional backup/recovery time for the increased database size (sometimes very important).

When a system is optimized for processing speed, it is critical to analyze the SQL Server indexes that are active because that is where the updating and retrieval time are determined. The determination of the proper number and design of keys and indexes 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 tables, there is only one field in the primary key and the only key is the primary key. An example is the Payment Terms table. Highlight Table 3 Payment Terms and click on the **Design** button then to see the **Keys** window, click on **View | Keys**:



The primary key must have a unique value in each table record. We can change the primary key to be any field, or combination of fields up to 16 fields totaling up to 900 bytes, but the uniqueness requirement must be met. It will automatically be enforced by NAV, because NAV will not allow us to add a record with a duplicate primary key to a table.

When we examine the primary keys in the supplied tables, we see that many of them consist only of or terminate in a **Line No.**, an **Entry No.**, or another data field whose contents make the key unique. For example, the G/L Entry table in the following screenshot uses just the **Entry No.** as the primary key. It is a NAV standard that **Entry No.** fields contain a value that is unique for each record:

The screenshot shows a Microsoft Dynamics NAV application window titled "Table 17 G/L Entry - Keys". The window displays a list of fields under the "E... Key" column and their corresponding "SumIndexFields" in the adjacent column. The fields listed are: "Entry No.", "G/L Account No., Posting Date", "G/L Account No., Global Dimension 1 Code, Global Dimension 2 Code, Po...", "G/L Account No., Business Unit Code, Posting Date", "G/L Account No., Business Unit Code, Global Dimension 1 Code, Global Di...", and "Document No., Posting Date". The "SumIndexFields" column shows "Amount,Debit Amount,Credit ..." for most fields, except for the last one which is partially cut off.

E... Key	SumIndexFields
Entry No.	Amount,Debit Amount,Credit ...
G/L Account No., Posting Date	Amount,Debit Amount,Credit ...
G/L Account No., Global Dimension 1 Code, Global Dimension 2 Code, Po...	Amount,Debit Amount,Credit ...
G/L Account No., Business Unit Code, Posting Date	Amount,Debit Amount,Credit ...
G/L Account No., Business Unit Code, Global Dimension 1 Code, Global Di...	Amount,Debit Amount,Credit ...
Document No., Posting Date	

The primary key of the Sales Line table shown in the following screenshot is made up of several fields, with the **Line No.** of each record as the terminating primary 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.**, which relate to the parent header record) makes each primary key entry unique. The logic supporting the assignment of **Line No.** values is done within explicit C/AL code. It is not an automatic feature. The **No. Series** pattern documentation can be found at

<https://community.dynamics.com/nav/w/designpatterns/74.no-series.aspx>:

E.. Key	SumIndexFields
✓ Document Type,Document No.,Line No.	Amount,Amount Including VAT...
✓ Document No.,Line No.,Document Type	
✓ Document Type,Type,No.,Variant Code,Drop Shipment,Location Code,...	Outstanding Qty. (Base)
✓ Document Type,Bill-to Customer No.,Currency Code	Outstanding Amount,Shipped ...
Document Type,Type,No.,Variant Code,Drop Shipment,Shortcut Dime...	Outstanding Qty. (Base)
Document Type,Bill-to Customer No.,Shortcut Dimension 1 Code,Short...	Outstanding Amount,Shipped ...
✓ Document Type,Blanket Order No.,Blanket Order Line No.	

All keys except the primary key are secondary keys. There is no required uniqueness constraint on secondary keys. There is no requirement to have any secondary keys. If we want a secondary key not to have duplicate values, our C/AL code must check for duplication before completing the new entry.

The maximum number of fields that can be used in any one key is 16, with a maximum total length of 900 bytes. At the same time, the total number of different fields that can be used in all of the keys combined cannot exceed 16. If the primary key includes three fields (as in the preceding screenshot), then the secondary keys can utilize up to thirteen other fields (that is  $16 - 3$ ) in various combinations, plus any or all of the fields in the primary key. If the primary key has 16 fields then the secondary keys can only consist of different groupings and sequences of those 16 fields.

Behind the scenes, each secondary key has the primary key appended to the backend. A maximum of 40 keys are allowed per table.

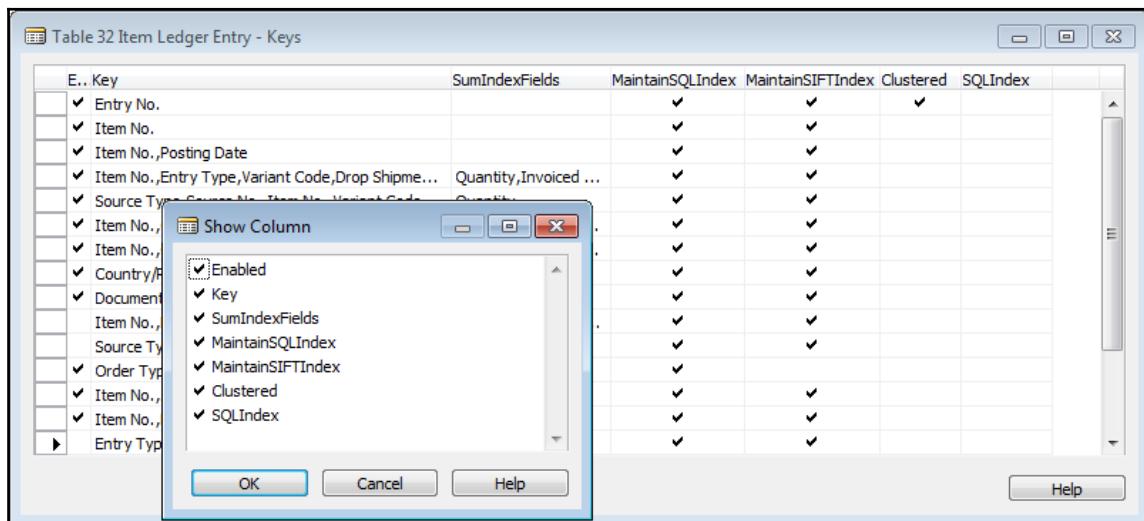


Database maintenance performance is faster with fewer fields in keys, especially the primary key. The same is true for fewer keys. This must be balanced against improved performance in processes by having the optimum key contents and choices.

Other significant key attributes include Key Groups and SQL Server-specific properties:

- A number of SQL Server-specific key-related parameters have been added to NAV. These key properties can be accessed by highlighting a key in the Keys form, then clicking on the Properties icon or pressing *Shift + F4*. We can also have those properties display in the Keys screen by accessing **View | Show Column** and selecting the columns we want displayed. The following screenshot shows both the **Show Column** choice form and the resulting Keys form with all the available columns displayed.

- The **MaintainSQLIndex** and **MaintainSIFTIndex** properties allow the developer and/or system administrator to determine whether or not a particular key or SIFT field will be continuously maintained or it will be recreated only when needed. Indexes that are not maintained minimize record update time but require longer processing time to dynamically create the indexes when they are used. This level of control is useful for managing indexes that are only needed occasionally. For example, a Key or **SumIndexField Technology (SIFT)** Index that is used only for monthly reports can be disabled and no index maintenance processing will be done day to day. If the month end need is for a single report, the particular index will be recreated automatically when the report is run. If the month end need is for a number of reports, the system administrator might enable the index, process the reports, then disable the index again:



## SumIndexFields

Since the beginning of NAV (formerly Navision), one of its unique capabilities has been the SIFT feature. 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, calculated on the fly at runtime, related to the **SumIndexFields**. The cost is primarily that of the time required to maintain the SIFT indexes when a table is updated.

NAV 2017 maintains SIFT totals using SQL Server indexed views. An indexed view is a view that has been preprocessed and stored. NAV 2017 creates one indexed view for each enabled SIFT key. SIFT keys are enabled and disabled through the **MaintainSIFTIndex** property. SQL Server maintains the contents of the view when any changes are made to the base table, unless the **MaintainSIFTIndex** property is set to **No**.

**SumIndexFields** are accumulated sums of individual fields (columns) in tables. When the totals are automatically pre-calculated, they are easy to use and provide very high-speed access for inquiries. If users need to know the total of the `Amount` values in a `Ledger` table, the `Amount` field can be attached as a **SumIndexField** to the appropriate keys. In another table, such as `Customer`, FlowFields can be defined as display fields take the advantage of the **SumIndexFields**. This gives users very rapid response for calculating a total balance amount inquiry based on detailed ledger amounts tied to those keys. We will discuss the various data field types and FlowFields in more detail in a later chapter.

In a typical ERP system, many thousands, millions, or even hundreds of millions of records might have to be processed to give such results, taking considerable time. In NAV, only a few records need to be accessed to provide the requested results. The processing is fast and the programming is greatly simplified.

SQL Server SIFT values are maintained through the use of SQL indexed views. By use of the key property **MaintainSIFTIndex**, we can control whether or not the SIFT index is maintained dynamically (faster response) or only created when needed (less ongoing system performance load). The C/AL code is the same whether the SIFT is maintained dynamically or not.

Having too many 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 update SIFT values.

Conversely, a lack of necessary keys or SIFT definitions can also cause performance problems. Having unnecessary data fields in a SIFT key creates many extra entries, affecting performance. Integer fields usually create an especially large number of unique SIFT index values, and Option fields create a relatively small number of index values.

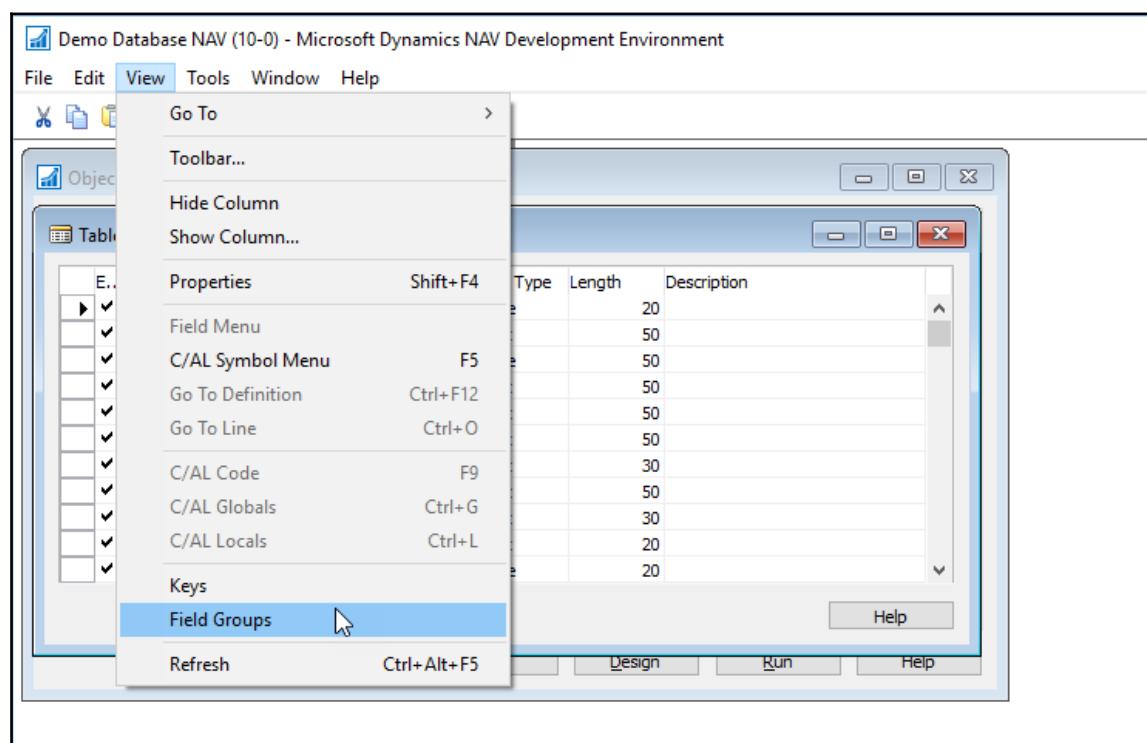
The best design for a SIFT index has the fields that will be used most frequently in queries positioned on the left side of the index in order of descending frequency of use. In a nutshell, we should be careful in our design of keys and SIFT fields. While a system is in production, applicable SQL Server statistics should be monitored regularly and appropriate maintenance actions taken. NAV 2017 automatically maintains a count for all SIFT indexes, thus speeding up all **COUNT** and **AVERAGE** FlowField calculations.

## Field Groups

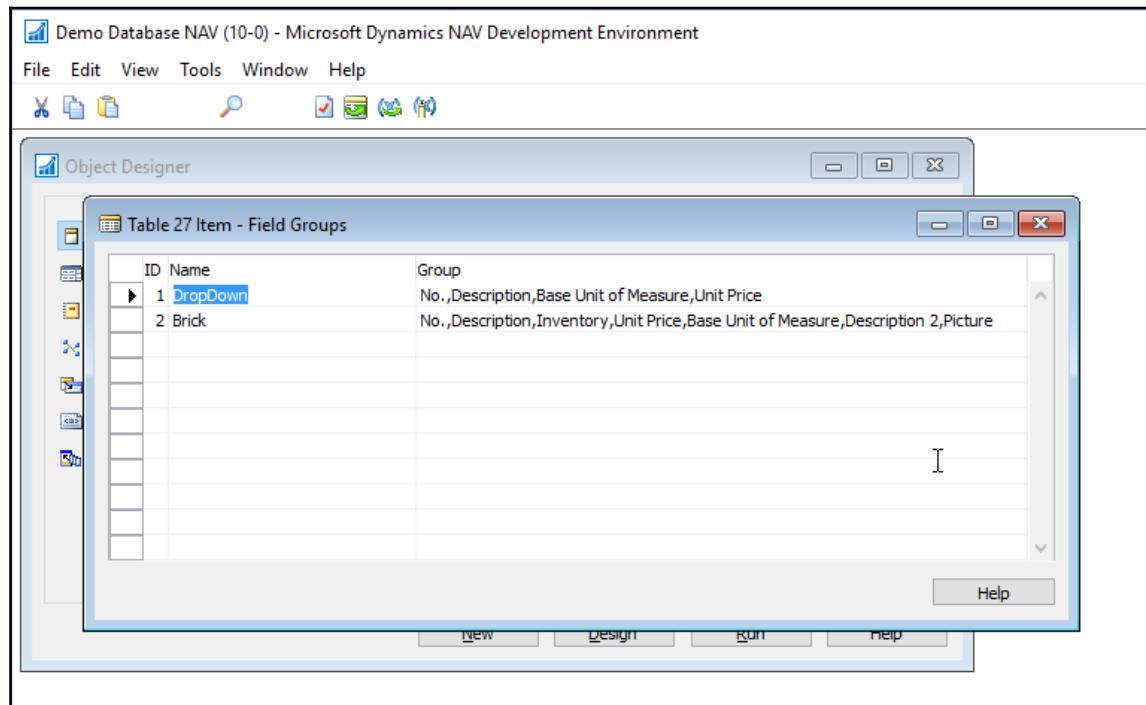
When a user starts to enter data in a field where the choices are constrained to existing data (for example, an **Item No.**, a **Salesperson Code**, a **Unit of Measure** code, a **Customer No.**, and so on), good design dictates that the system will help the user by displaying the universe of acceptable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

In the **Role Tailored Client (RTC)**, the lookup display (a drop-down control) is generated dynamically when its display is requested by the user's effort to enter data in a field that references a table through the **TableRelation** property (which will be discussed in more detail in the next chapter). The format of the dropdown is a basic list. The fields that are included in that list and their left to right display sequence are either defined by default, or by an entry in the **Field Groups** table.

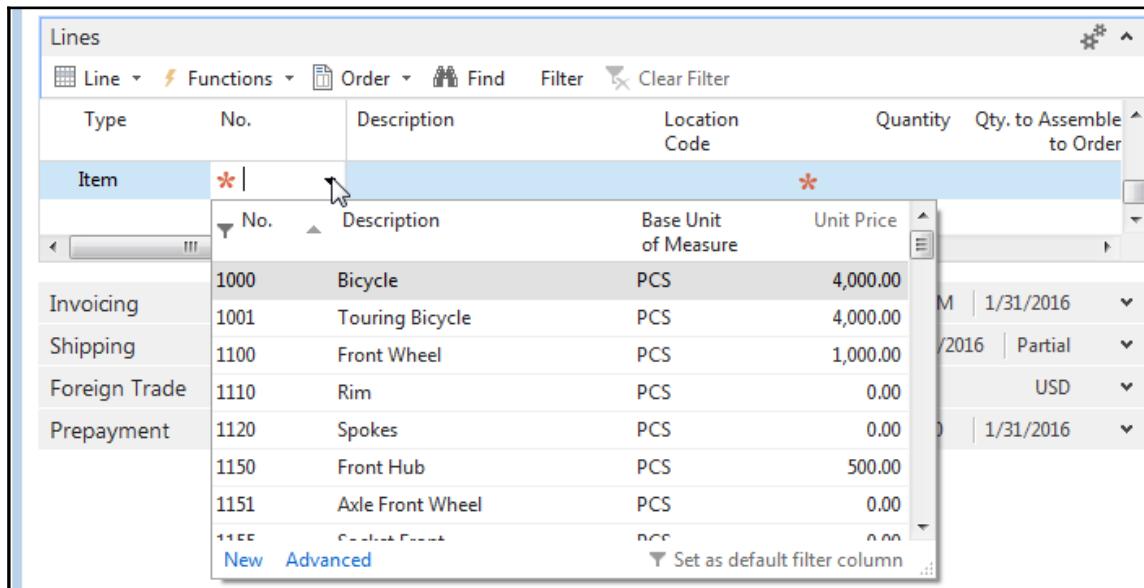
The **Field Groups** table is part of the NAV table definition much like the list of Keys. In fact, the **Field Groups** table is accessed very similar to the Keys, navigating through **View | Field Groups**, as shown in the following screenshot:



If we look at the **Field Groups** for **Table 27 - Item**, we see the drop-down information defined in a field group, which must be named **DropDown** (without a hyphen):



The drop-down display created by this particular field group is shown in the following screenshot on the **Sales Order** page containing fields in the same order of appearance as in the field group definition:



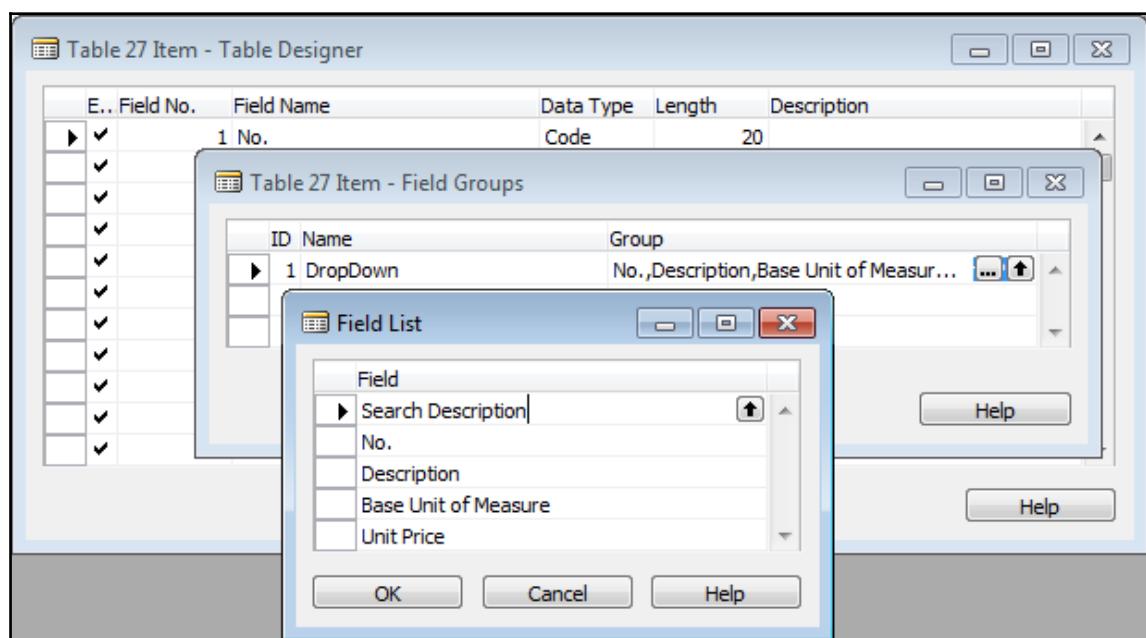
If no field group is defined for a table, the system defaults to using the primary key plus the **Description** field (or **Name** field).

Since **Field Groups** can be modified, they provide another opportunity to tailor the user interface. As we 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 de facto lookup field.

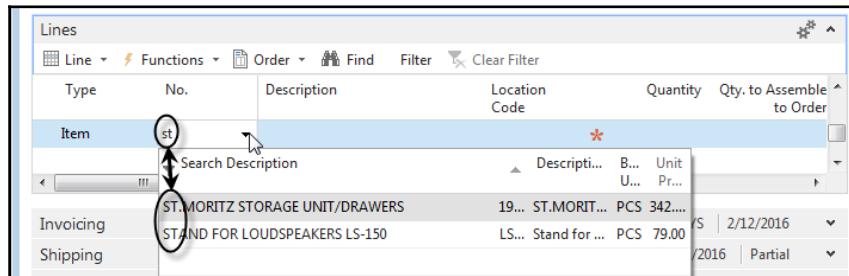
As a system option, the drop-down control provides a find-as-you-type capability, where the set of displayed choices filters and re-displays dynamically as the user types, character by character. The filter applies to the default filter column. Whatever field is used for the lookup, the referential field defined in the page determines what data field contents are copied into the target table. In the preceding screenshot, the reference table/field is the Sales Line table/field **No.** and the target table/field is the Item table/field **No.**.

As developers, we can change the order of appearance of the fields in the drop-down display. We can also add or delete fields for the display by changing the contents of the field group. For example, we could add a capability to our page that provides an alternate search capability (where if the match for an Item No. isn't found in the **No.** field, the system will look for a match based on another field).

Consider this situation: the customer has a system design where the Item No. contains a 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 **Field List**:



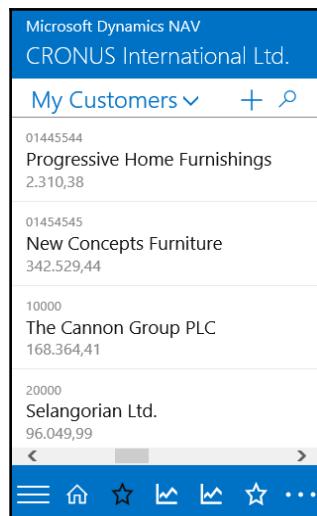
The effect of this change can be seen in the following screenshot, which shows the revised drop-down control. The user has begun entry in the **No.** field, but the lookup is focused on the newly added **Search Description** field. Find-as-you-type filtered the displayed items down to just those that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



The result of our change allows the user to look up items by their **Search Description**, rather than by the harder to remember Item No. Obviously, any field in the Item table could have been used, including our custom fields.

## Bricks

In addition to the `DropDown` field group, we can also define a Brick. Bricks are used in the Web client and Phone client. The Brick for the Item table is rendered like the following screenshot:



## Tables

The same Brick is rendered differently in the Web client, as displayed in this screenshot:

The screenshot shows the Microsoft Dynamics NAV Web client interface. The top navigation bar includes links for HOME, ACTIONS, NAVIGATE, REPORT, and a user profile icon. The main area displays a grid of items under the heading 'Items'. Each item card includes a thumbnail image, the item number, name, quantity, and unit price. To the right of the grid, there are two sections of detailed information for the selected item, which is 'BERLIN Guest Chair, yellow' (Item No. 1000). The first section, 'Item Details - Invoicing', lists fields like Costing Method (Standard), Cost is Adjusted (No), and Standard Cost (350,594). The second section, 'Item Details - Planning', lists fields like Reordering Policy (Fixed Reorder Qty.), Reorder Point (0), and Maximum Inventory (0).

Item No.	1000
Costing Method	Standard
Cost is Adjusted	No
Cost is Posted to G/L	Yes
Standard Cost	350,594
Unit Cost	350,594
Overhead Rate	0,00
Indirect Cost %	0
Last Direct Cost	0,00
Profit %	91,23515
Unit Price	4,000,00

Item No.	1000
Reordering Policy	Fixed Reorder Qty.
Reorder Point	0
Reorder Quantity	100
Maximum Inventory	0
Overflow Level	0
Time Bucket	1W
Lot Accumulation Period	
Rescheduling Period	
Safety Lead Time	
Safety Stock Quantity	0
Minimum Order Quantity	0

## Enhancing our sample application

Now we can take our knowledge of tables and expand our WDTU application. Our base Radio Show table needs to be added to and modified. We also need to create and reference additional tables.

Although we want to have a realistic design in our sample application, we will focus on changes that illustrate features in NAV table design that the authors feel are among the most important. If there are capabilities or functionalities that you feel are missing, feel free to add them. Adjust the examples as much as you wish to make them more meaningful to you.

## Creating and modifying tables

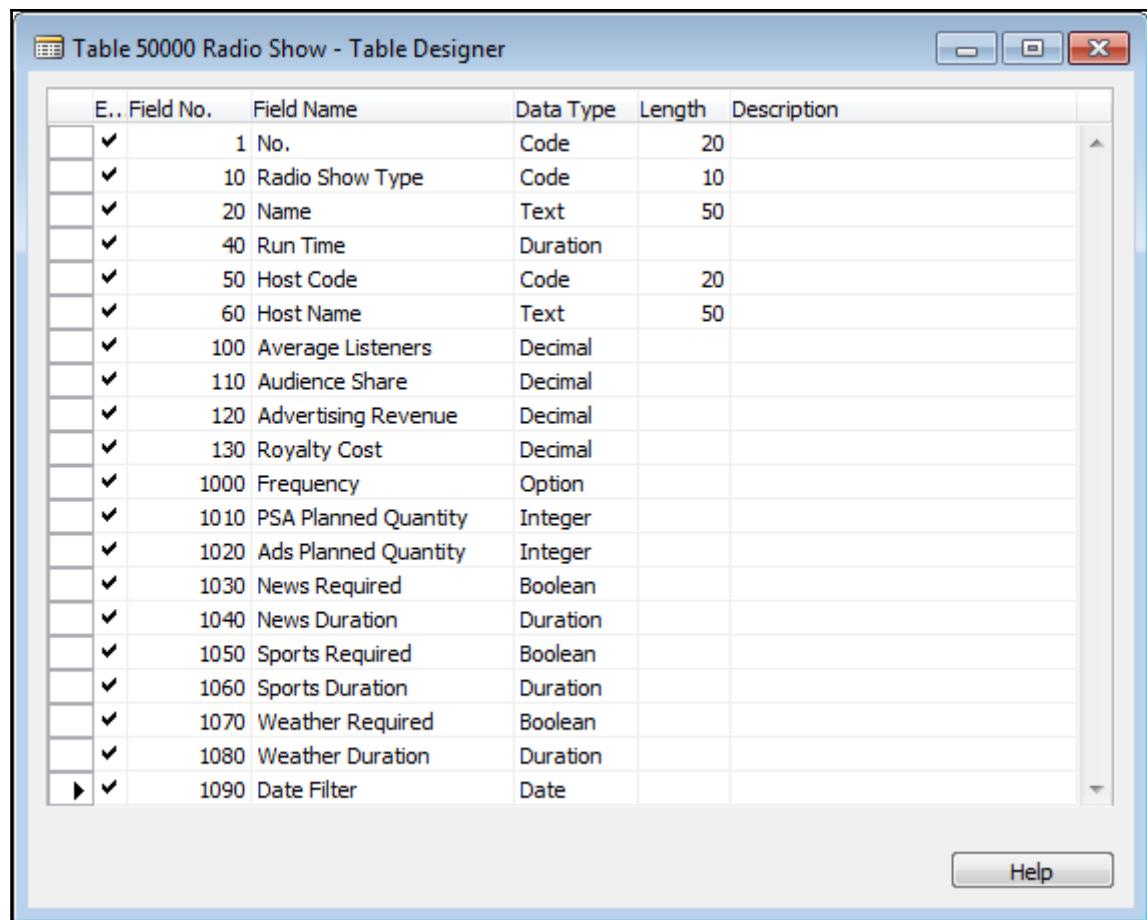
In Chapter 1, *Introduction to NAV 2017*, we created the `Radio Show` table for the WDTU application. At that time, we used the minimum fields that allowed us to usefully define a master record. Now let's set properties on existing data fields, add more data fields, and create an additional data table to which the `Radio Show` table can refer.

Our new data fields are shown in the following layout table:

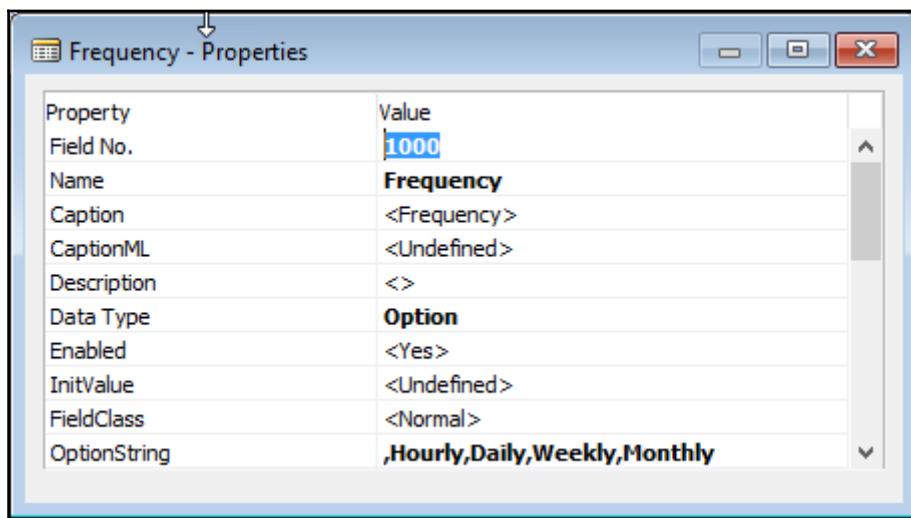
Field no.	Field name	Description
1000	<b>Frequency</b>	An <code>Option</code> data type (Hourly, Daily, Weekly, Monthly) for the frequency of a show; Hourly to be used for a show segment such as news, sports, or weather that is scheduled every hour. A space/blank is used for the first option to allow a valid blank field value.
1010	<b>PSA Planned Quantity</b>	A number (stored as an <code>Integer</code> ) of <b>Public Service Announcements (PSAs)</b> to be played per show; will be used by playlist generation and posting logic.
1020	<b>Ads Planned Quantity</b>	A number (stored as an <code>Integer</code> ) of advertisements to be played per show; will be used by playlist generation and posting logic.
1030	<b>News Required</b>	Is headline news required to be broadcast during the show (a <code>Boolean</code> )?
1040	<b>News Duration</b>	The duration (stored as a <code>Duration</code> ) of the news program embedded within the show.
1050	<b>Sports Required</b>	Is sports news required to be broadcast during the show (a <code>Boolean</code> )?
1060	<b>Sports Duration</b>	The duration (stored as a <code>Duration</code> ) of the sports program embedded within the show.
1070	<b>Weather Required</b>	Is weather news required to be broadcast during the show (a <code>Boolean</code> )?
1080	<b>Weather Duration</b>	The duration (stored as a <code>Duration</code> ) of the weather program embedded within the show.

1090	<b>Date Filter</b>	A date FlowFilter (stored as a Data Type Date, Data Class FlowFilter) that will change the calculations of the flow fields based on the date filter applied. More on FlowFilters in Chapter 3, <i>Data Types and Fields</i> .
------	--------------------	---

After we have completed our Radio Show table, it will look like this:



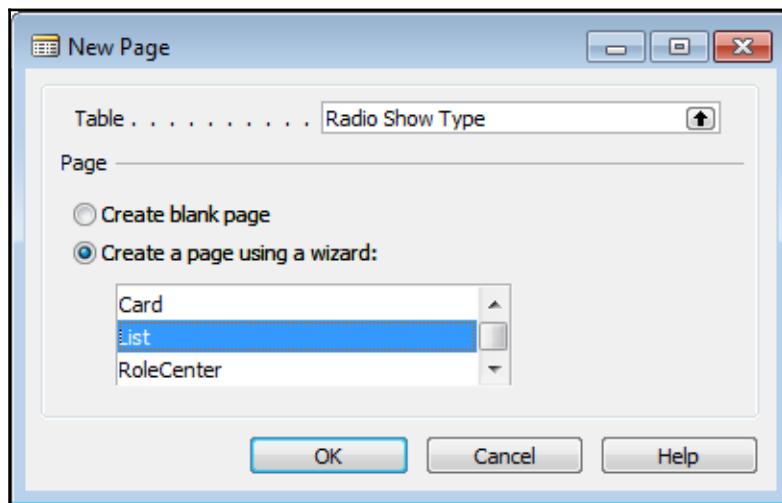
Next, we need to fill the **OptionString** and **Caption** properties for the Option field **Frequency**. Highlight the **Frequency** field, then click on the Properties icon or press *Shift + F4*. Enter values for the **OptionString** property, as shown in the next screenshot; don't forget to allow the leading space followed by a comma to get a space/blank as the first option. Be sure to copy and paste the same information to the **OptionCaption** property. The **OptionCaptionML** property will fill in automatically with a copy of that information (since we do not have a second language installed). Notice that the properties that have been changed from the default are displayed in bold. This new feature makes it much easier for developers to see what properties have been modified:



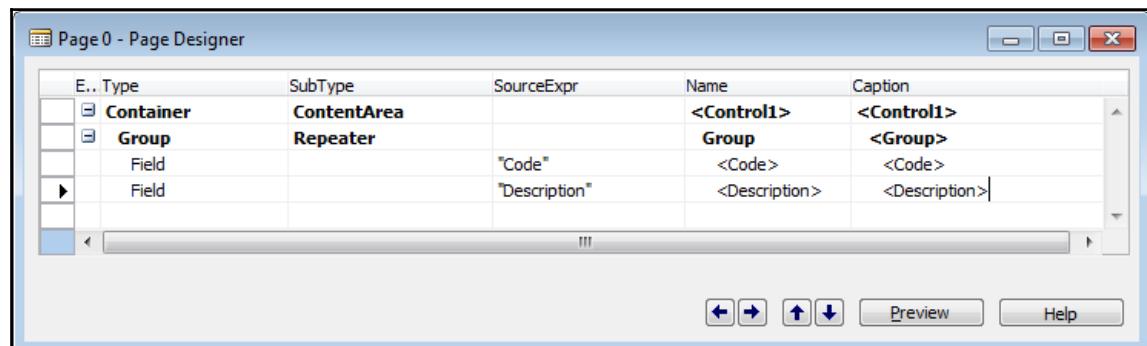
Next, we want to define the reference table we are going to tie to the **Type** field. The table will contain a list of the available Radio Show Types such as Music, Talk, Sports, and so on. We will keep this table very simple, with **Code** as the unique key field and **Description** as the text field. Both fields will be of the default length as shown in the following layout. Create the new table and save it as Table 50001 with the name of Radio Show Type:

Field No.	Field Name	Description	Data Type	Length
10	Code	Primary Key of data type Code	Code	10
20	Description	A text field	Text	30

Before we can use this table as a reference from the Radio Station table, we need to create a list page that will be used both for data entry and data selection for the table. We will use Page Designer and the Page Wizard to create a list page. We should be able to do this pretty quickly. Click on **Pages**, click on the **New** button, enter **50001** in the **Table** field (the table field will redisplay the table name), then choose the Wizard to create a page type of **List**:



Populate the page with both (all) the fields from the `Radio Show Type` table. Our designed page should look like the following screenshot:



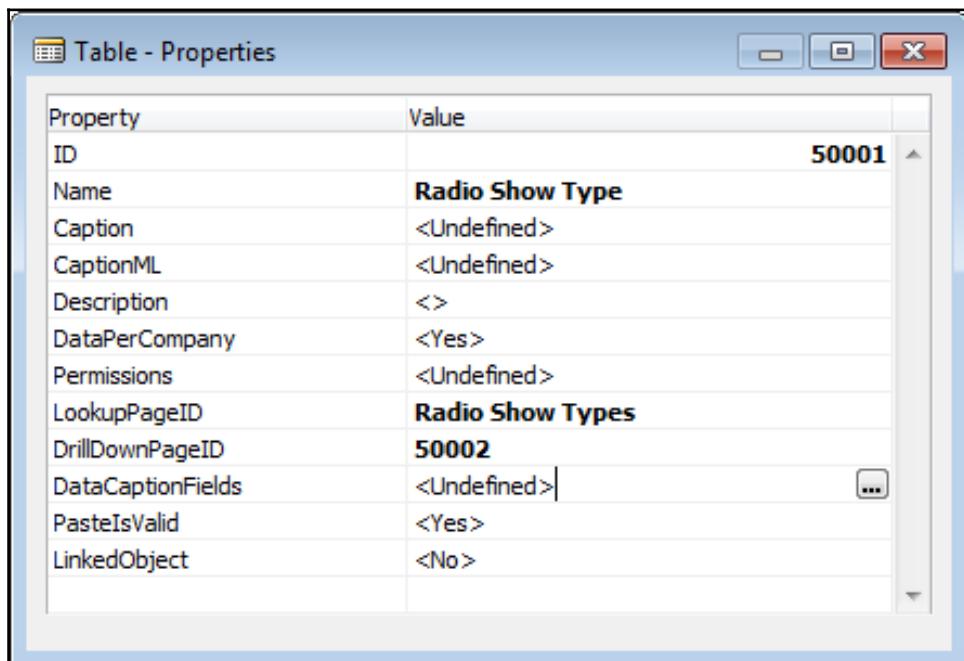
Save the page as number 50002, exit the Page Designer, and name the page Radio Show Types. Test the page by highlighting it in the **Object Designer** and then clicking on the **Run** button. The new page will be displayed. While the page is open, enter some data (by clicking on **New**) like the examples shown in the following screenshot:

The screenshot shows a SharePoint list view titled "Radio Show Types". The list contains four items:

Code	Description
CALL-IN	Talk and Listener In...
MUSIC	Music and Misc
NEWS	In-depth Stories
TALK	Mostly Talk

The "CALL-IN" item is currently selected, as indicated by the highlighted row. The list has a header bar with buttons for "New", "Edit List", "Delete", "View", "Show Attached", and "Page".

Now we'll return to the Radio Show Type table and set the table's properties for **LookupPageID** and **DrillDownPageID** to point to the new page we have just created. As a reminder, we will use Design to open the table definition, then focus on the empty line below the **Description** field, and either click on the properties icon or press *Shift + F4*. In the values for each of the two PageID properties, we can either enter the page name (Radio Show Types) or the page number (50002). Either entry will work, but as you can see in the following screenshot, the appearance depends on what you enter:



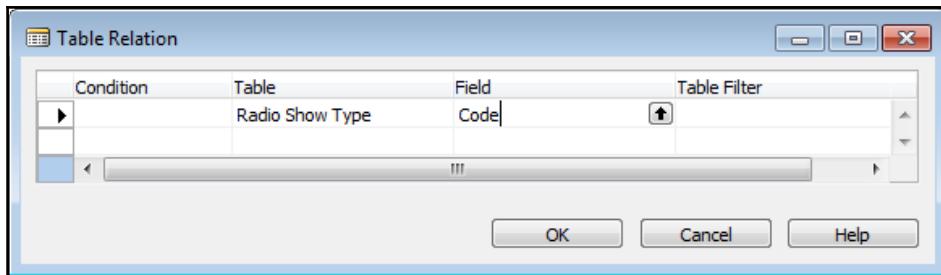
After the table has been saved, the next time we view those two PageID properties, they look like this:

The screenshot shows a Windows-style dialog box titled "Table - Properties". It contains a table with two columns: "Property" and "Value". The properties listed are: ID, Name, Caption, CaptionML, Description, DataPerCompany, Permissions, LookupPageID, DrillDownPageID, DataCaptionFields, PasteIsValid, LinkedObject, and TableType. The values for these properties are: 50001, Radio Show Type, <Undefined>, <Undefined>, <>, <Yes>, <Undefined>, Radio Show Types, Radio Show Types, <Undefined>, <Yes>, <No>, and <Normal>. The "Name" and "LookupPageID" properties are highlighted in bold blue text.

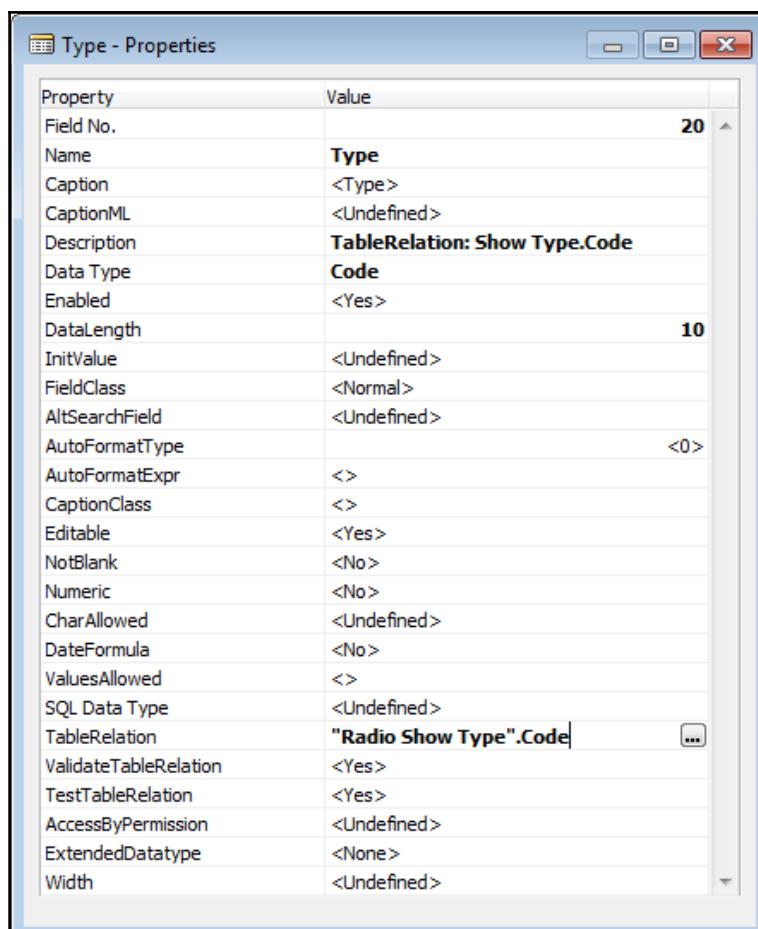
Property	Value
ID	50001
Name	<b>Radio Show Type</b>
Caption	<Undefined>
CaptionML	<Undefined>
Description	<>
DataPerCompany	<Yes>
Permissions	<Undefined>
LookupPageID	<b>Radio Show Types</b>
DrillDownPageID	<b>Radio Show Types</b>
DataCaptionFields	<Undefined>
PasteIsValid	<Yes>
LinkedObject	<No>
TableType	<Normal>

## Assigning a table relation property

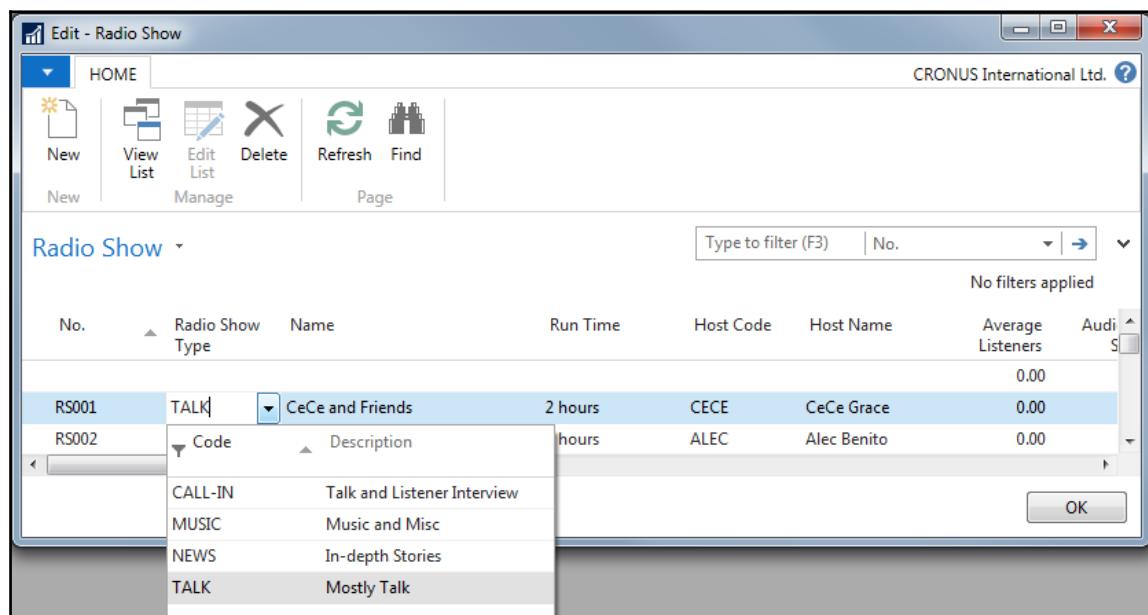
Finally, we open the **Radio Show** table again by highlighting the table line and clicking on the **Design** button. This time highlight the **Type** field and access its **Properties** screen. Highlight the **TableRelation** property and click on the ellipsis button (the three dots). We see the **Table Relation** screen with four columns as shown in the following image. The middle two columns are headed **Table** and **Field**. In the top line in the **Table** column, enter 50001 (the table number) or **Radio Show Type** (the table name). In the same line, the **Field** column, click on the up arrow button and choose **Code**:



We exit the **Table Relation** screen (by clicking the **OK** button) and return to the **Type - Properties** page that looks like following image. Save and exit the modified table:



To check that the **TableRelation** is working properly, we could run the **Radio Show** table (that is, highlight the table name and click on the **Run** button). We could also run the **Radio Show List** page and have almost exactly the same view of the data. This is because the Run of a table creates a temporary list page that includes all the fields in the table, thus it contains the same data fields as the page we created using the page wizard. In either one, we should highlight the **Radio Show Type** field and click on the drop-down arrow to view the list of available entries. The following screenshot is of our Radio Show List page; you should also try it using the **Run** function on the **Radio Show** table:

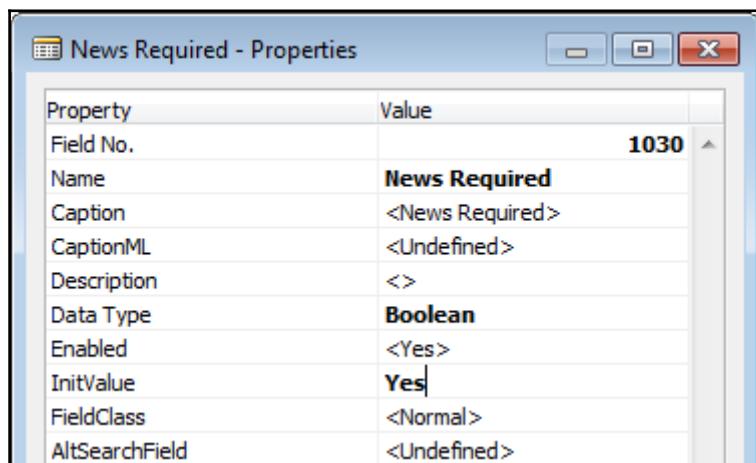


If all has gone according to plan, the **Radio Show Type** field will display a drop-down arrow (the downward pointing arrowhead button). Whether we click on that button or press **F4**, we will invoke the drop-down list for the **Radio Show Type** table as shown in the preceding screenshot.

## Assigning an **InitValue** property

Another property we can define for several of the Radio Show fields is **InitValue**. WDTU has a standard policy that news, sports, and weather be broadcast for a few minutes each at the beginning of every hour in the day. We want the Boolean (Yes/No) fields, **News Required**, **Sports Required**, and **Weather Required** to default to **Yes**. We also want the default time value of the **News Duration**, **Sports Duration**, and **Weather Duration** to be two minutes, two minutes, and one minute respectively. That way the first five minutes of every hour can be spent on keeping the listeners informed of the latest happenings.

Setting the default for a field to a specific value simply requires setting the **InitValue** property to the desired value. In the case of the required Boolean fields, that value is set to **Yes**. Using the Table Designer, we must design the Radio Station table, access the Properties screen for the News Required field. Repeat this for the Sports Required and Weather Required fields. After we have filled in the values for the three fields, exit the Properties screen, exit the Table Designer, and save the changes:



In the case of the **Duration** fields, we will set the **InitValue** to two minutes each for News Duration and Sports Duration, and one minute for Weather Duration. Duration fields require both the time span numeric and the time unit of measure (seconds, minutes, hours, and so forth). Our entries will look like 2 minutes (or 2 minute, both are acceptable).

## Adding a few activity-tracking tables

Our WDTU organization is a profitable and productive radio station. We track historical information about our advertisers, royalties owed, and listenership. We track the music that is played, the rates we charge for advertisements based on the time of day, and we provide a public service by broadcasting a variety of government and other public service announcements.

We aren't going to cover all these features and functions in the following detailed exercises. However, it's always good to have a complete view of the system on which we are working, even if we 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.

Any system development should start with a design document that completely spells out the goals and the functional design details. Neither system design nor project management will be covered in this book, but when we begin working on production projects, proper attention to both of these areas will be critical to success. Use of a proven project management methodology can make a project much more likely to be on time and within budget.

Based on the requirements our analysts have given us, we need to expand our application design. We started with a `Radio Show` table, one reference table (**Radio Show Type**), and pages for each of them. We earlier entered some test data and added a few additional fields to the `Radio` table (which we will not add to our pages here).

Now we will add a supplemental table, document (header and line) tables, plus a ledger (activity history) table relating to Playlist activities. Following this, we will also create some pages for our new data structures.

Our WDTU application will now include the following tables:

- **Radio Show:** A master list of all programs broadcast by our station.
- **Radio Show Type:** A reference list of possible types of radio shows.
- **Playlist Header:** A single instance of a `Radio Show` with child data in the form of playlist lines.
- **Playlist Line:** Each line represents one of a list of items and/or durations per `Radio Show`.
- **Playlist Item Rate:** A list of rates for items played during a show as determined by our advertising sales staff or the royalty organization we use.

- **Radio Show Ledger:** A detailed history of all the time spent and items played during the show with any related royalties owed or advertisement revenue expected.
- **Listenership Ledger:** A detailed history of estimated listenership provided by the ratings organization to which we subscribe.
- **Publisher:** A reference list of the publishers of content that we use. This will include music distributors, news wires, sports, and weather sources, as well as WDTU (we use material that we publish).

Remember, one purpose of this example system is for you to follow along in a hands-on basis in your own system. You may 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 support the volunteers provide.

For the best learning experience, you should be creating each of these objects in your development system to learn by experimenting. During 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 with the minimum cost.

## New tables for our WDTU project

First, we create a `Playlist Header` table (table number 50002), which will contain one record for each scheduled Radio Show:

The screenshot shows the Microsoft Access Table Designer window titled "Table 50002 Playlist Header - Table Designer". The table has the following structure:

E..	Field No.	Field Name	Data Type	Length	Description
▶	10	No.	Code	20	
▶	20	Radio Show No.	Code	20	
▶	30	Description	Text	30	
▶	40	Broadcast Date	Date		
▶	50	Duration	Duration		
▶	60	Start Time	Time		
▶	70	End Time	Time		

## Tables

---

Then we will create the associated Playlist Line table (table number 50003). This table will contain child records to the Playlist Header table. Each Playlist Line record represents one scheduled piece of music or advertisement or public service announcement or embedded show within the scheduled Radio Show defined in the Playlist Header table. The **Description** for each of the **Option** fields shows the information that needs to be entered into the **OptionString**, **OptionCaption** and **OptionCaptionML** properties for those fields:

E..	Field No.	Field Name	Data Type	Length	Description
▶	10	Document No.	Code	20	
✓	20	Line No.	Integer		
✓	30	Type	Option		,Resource,Show,Item
✓	40	No.	Code	20	
✓	50	Data Format	Option		,Vinyl,CD,MP3,PSA,Advertisement
✓	60	Publisher	Code	10	
✓	70	Description	Text	30	
✓	80	Duration	Duration		
✓	90	Start Time	Time		
✓	100	End Time	Time		

Now we'll create our Playlist Item Rate table. These rates include both what we charge for advertising time and what we must pay in royalties for material we broadcast:

E..	Field No.	Field Name	Data Type	Length	Description
▶	10	Source Type	Option		Vendor,Customer
✓	20	Source No.	Code	20	
✓	30	Item No.	Code	20	
✓	40	Start Date	Date		
✓	50	End Date	Date		
✓	60	Rate Amount	Decimal		
✓	70	Publisher Code	Code	20	

## Tables

---

An entry table contains the detailed history of processed activity records. In this case, the data is a detailed history of all the Playlist Line records for previously broadcast shows:

The screenshot shows the 'Table 50005 Radio Show Entry - Table Designer' window. It displays a table with 12 columns: E.., Field No., Field Name, Data Type, Length, and Description. The 'Length' column is only visible for the second field. The 'Description' column is empty for most fields.

E..	Field No.	Field Name	Data Type	Length	Description
▶	10	Entry No.	Integer		
✓	20	Radio Show No.	Code	20	
✓	30	Type	Option		
✓	40	No.	Code	20	
✓	50	DataFormat	Option		
✓	60	Description	Text	30	
✓	70	Date	Date		
✓	80	Time	Time		
✓	90	Duration	Duration		
✓	100	Fee Amount	Decimal		
✓	110	ASCAP ID	Integer		
✓	120	Publisher Code	Code	20	

Help

Now we'll create one more entry table to retain data we receive from the listenership rating service:

The screenshot shows the 'Table 50006 Listenership Entry - Table Designer' window. It displays a table with 9 columns: E.., Field No., Field Name, Data Type, Length, and Description. The 'Length' column is only visible for the fifth field. The 'Description' column is empty for most fields.

E..	Field No.	Field Name	Data Type	Length	Description
▶	10	Entry No.	Integer		
✓	20	Ratings Source Entry No	Integer		
✓	30	Date	Date		
✓	40	Start Time	Time		
✓	50	End Time	Time		
✓	60	Radio Show No.	Code	20	
✓	70	Listener Count	Decimal		
✓	80	Audience Share	Decimal		
✓	90	Age Demographic	Option		

Help

Finally, the last new table definition for now, our Publisher table:

The screenshot shows the Microsoft Access Table Designer window titled "Table 50007 Publisher - Table Designer". The table has two fields defined:

E..	Field No.	Field Name	Data Type	Length	Description
▶	10	No.	Code	20	
▼	20	Name	Text	30	

A "Help" button is located in the bottom right corner of the window.

## New list pages for our WDTU project

Each of the new tables we have created should be supported with an appropriately named list page. As part of our WDTU project work, we should create the following pages:

- 50003 - Playlist Document
- 50005 - Playlist Item Rates
- 50006 - Radio Show Entries
- 50007 - Listenership Entries
- 50008 - Publishers

## Keys, SumIndexFields, and TableRelations in our examples

So far, we have created basic table definitions and associated pages for the WDTU project. The next step is to flesh out those definitions with additional keys, SIFT field definitions, table relations, and so on. The purpose of these are to make our data easier and faster to access, to take advantage of the special features of NAV to create data totals, and to facilitate relationships between various data elements.

## Secondary keys and SumIndexFields

The `Playlist Line` table default primary key was the one field **Document No.**. In order for the primary key to be unique for each record, another field is needed. For a line table, the additional field is the **Line No.** field, which is incremented through C/AL code for each record. So, we'll change the key for table 50003 accordingly:

Table 50003 Playlist Line - Keys	
E.. Key	SumIndexFields
▶ ✓ Document No.,Line No.	

We know a lot of reporting will be done based on the data in the Radio Show Ledger. We also know we want to do reporting on data by `Radio Show` and by the type of entry (individual song, specific ad, and so on). So, we will add secondary keys for each of those, including a **Date** field so we can rapidly filter the data by **Date**. The reporting that is financial in nature will need totals of the **Fee Amount** field, so we'll put that in the **SumIndexFields** column for our new keys:

Table 50005 Radio Show Entry - Keys	
E.. Key	SumIndexFields
▶ ✓ Entry No.	
✓ Radio Show No.,Date	Fee Amount
✓ Type,No.,Date	Fee Amount

We know that to do the necessary listenership analysis, the listenership ledger needs an additional key combined with **SumIndexFields** for totaling listener statistics:

Table 50006 Listenership Entry - Keys	
E.. Key	SumIndexFields
▶ ✓ Entry No.	
✓ Radio Show No.,Date,Start Time,Age Dem...	Listener Count,Audience Share

To utilize the **SumIndexFields** we have just defined, we will need to define corresponding FlowFields in other tables. We will leave that part of the development effort for the next chapter, where we are going to discuss Fields, FlowFields, and FlowFilters in detail.

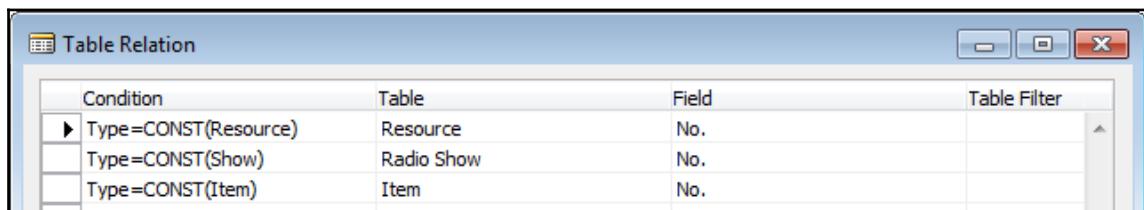
## Table Relations

For the tables where we defined fields intended to refer to data in other tables for lookups and validation, we must define the relationships in the referring tables. Sometimes those relationships are complicated, dependent on other values within the record.

In Table 50003, Playlist Line, we have the field **No..** If the **Type** field contains **Resource**, then the **No.** field should contain a **Resource No..** If the **Type** field contains **Show**, then the **No.** field should contain a **Radio Show No..** And, if the **Type** field contains **Item**, the **No.** field should contain an **Item No..** The pseudo-code (approximate syntax) for that logic can be written as follows:

```
IF Type = 'Resource' THEN No. := Resource.No.
ELSE IF Type = 'Show' THEN No. := Radio Show.No.
ELSE IF Type = 'Item' THEN No. := Item.No.
```

Fortunately, a tool built into the C/SIDE editor makes it easy for us to build that complex logic in the **TableRelation** property. When we click on the **TableRelation** property, then click on the ellipsis button (three dots), we get a **Table Relation** screen we can use to construct the necessary logic structure:



When we exit the **Table Relation** screen by clicking on the **OK** button, the **TableRelation** line looks like the following:

SQL Data Type	<Undefined>
TableRelation	IF (Type=CONST(Resource)) Resource.No. ELSE IF (Type=CONST(Show)) "Radio Show".No. ELSE IF (Type=CONST(Item)) Item.No.
ValidateTableRelation	<Yes>
TestTableRelation	<Yes>

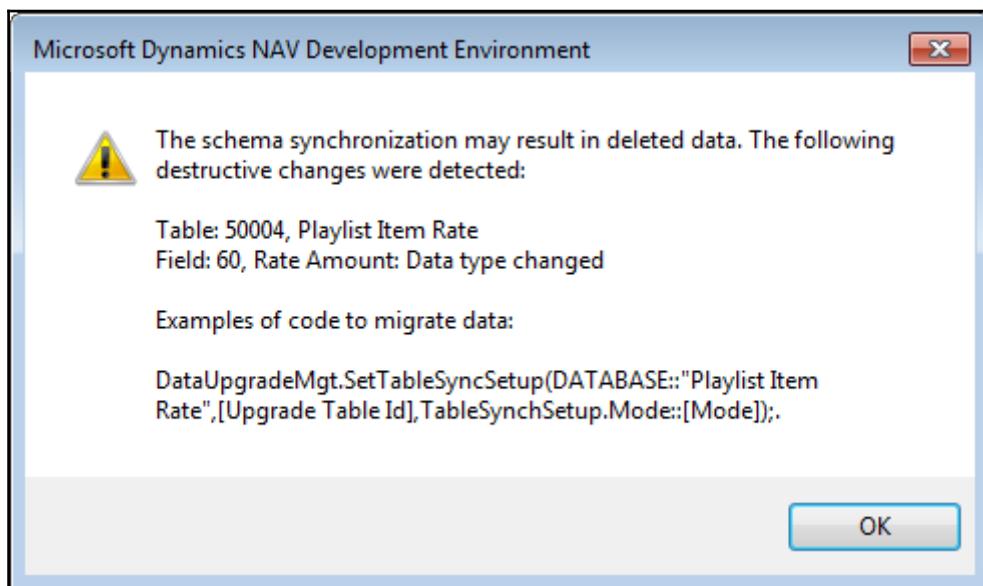
Table 50004, Playlist Item Rate, has a similar **TableRelation** requirement for the **Source No..** field in that table. In this case, if **Source Type** = **Vendor** then the **Source No..** field will refer to the **Vendor No..** field or if the **Source Type** = **Customer** then the **Source No..** field will refer to the **Customer No..** field:

Table Relation			
Condition	Table	Field	Table Filter
► Source Type=CONST(Vendor)	Vendor	No.	
Source Type=CONST(Customer)	Customer	No.	

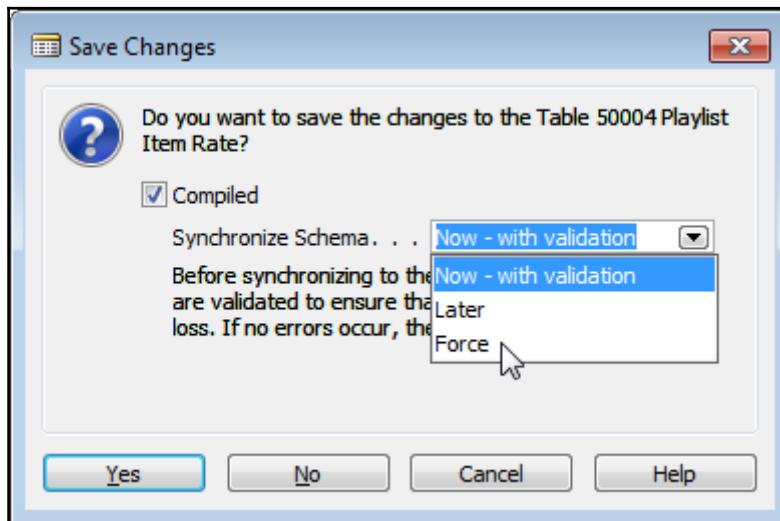
When we exit the **Table Relation** screen this time (by clicking on the **OK** button), the **TableRelation** line looks like the following:

SQL Data Type	<Undefined>
TableRelation	<b>IF (Source Type=CONST(Vendor)) Vendor.No. ELSE IF (Source Type=CONST(Customer)) Customer.No.</b>
ValidateTableRelation	<Yes>

If in the process of making these changes (or some future changes), we realize that we need to change the data type of a field and try to do so, we may get this error message:



The intent of this message is to keep us from unintentionally deleting or corrupting data through a change in the table definition. After checking that we really aren't going to make that mistake, perhaps making a backup of the data about to be affected, we can override the system's error checking and force the change to be done by setting the **Synchronize Schema** ... field to **Force** in the **Save Changes** screen, as shown in the following screenshot:



## Modifying an original NAV table

One of the big advantages of the NAV system development environment is the fact that we are allowed to enhance the tables that are part of the original standard product. Many package software products do not provide this flexibility. Nevertheless, with privilege comes responsibility. When we modify a standard NAV table, we must do so carefully.

In our system, we are going to use the standard `Item` table - Table 27, to store data about recordings such as music and advertisements and PSAs which we have available for broadcast. One of the new fields will be an Option field. Another will refer to the `Publisher` table we created earlier. When the modifications to the `Item` table design are completed, they will look like the following screenshot:

The screenshot shows the Microsoft Dynamics NAV Table Designer window titled "Table 27 Item - Table Designer". The table has columns: E.., Field No., Field Name, Data Type, Length, and Description. The rows list various fields with their properties:

E..	Field No.	Field Name	Data Type	Length	Description
✓	7386	Next Counting End Date	Date		
✓	7700	Identifier Code	Code	20	
✓	50000	Publisher	Code	10	
✓	50010	ASCAP ID	Integer		
✓	50020	Duration	Duration		
✓	50030	DataFormat	Option		
▶ ✓	50040	MP3 Location	Text	250	
✓	99000750	Routing No.	Code	20	
✓	99000751	Production BOM No.	Code	20	

Note that we were careful not to touch any of the standard fields that were already defined in the `Item` table. Plus, we numbered all of our new fields in the range of 50000 to 99999, making them easy to identify as belonging to a Partner modification.

## Version List documentation

In Chapter 1, *Introduction to NAV 2017*, we mentioned the importance of good documentation, one component being the assignment of version numbers to modifications and enhancements. Frequently, modifications are identified with a letter/number combination code, the letters indicating who did the modification (such as the NAV Partner initials - or, in this case, the book authors' combined initials) and a sequential number for the specific modification. Our Partner initials are *CDM*, so all our modifications will have a version number of *CDMxx*. We will use the chapter number of this book for the sequential number, such as:

- *CDM01 - Chapter 01*
- *CDM02 - Chapter 02*
- *CDM01,02 - Chapters 01 and 02*

When applied to the table objects we have created so far, our **Version List** entries look like the following:

The screenshot shows the 'Object Designer' window with the 'Table' type selected in the left sidebar. The main area displays a grid of table objects with columns for Type, ID, Name, Modified, Version List, Date, Time, Compiled, and Locked. The 'Version List' column shows the history of changes for each table. For example, '27 Item' has a history entry of 'NAWW110.0,CDM02' at '01/09/17 7:45:36...' and another entry of 'CDM01,02' at '02/08/17 11:00:5...'. Other tables listed include 'Radio Show', 'Radio Show Type', 'Playlist Header', 'Playlist Line', 'Playlist Item Rate', 'Radio Show Entry', 'Listenership Entry', and 'Publisher'.

Type	ID	Name	Modified	Version List	Date	Time	Compiled	Locked
Table	27	Item	✓	NAWW110.0,CDM02	01/09/17	7:45:36...	✓	
Table	50000	Radio Show	✓	CDM01,02	02/08/17	11:00:5...	✓	
Table	50001	Radio Show Type	✓	CDM02	12/13/14	6:40:52...	✓	
Table	50002	Playlist Header	✓	CDM02	02/18/17	1:42:25...	✓	
Table	50003	Playlist Line	✓	CDM02	01/11/17	8:56:44...	✓	
Table	50004	Playlist Item Rate	✓	CDM02	10/22/16	9:17:33...	✓	
Table	50005	Radio Show Entry	✓	CDM02	10/22/16	9:10:58...	✓	
Table	50006	Listenership Entry	✓	CDM02	10/22/16	8:59:54...	✓	
Table	50007	Publisher	✓	CDM02	10/22/16	9:01:06...	✓	

When working on a customer's system, a more general purpose versioning structure should be used in the same general format as the one used by Microsoft for the product. Such a structure would be in the format *CDM5.00.01* (CD company, Fifth Book Edition, minor version 00 (no Service Pack), build 01. The next release of objects would then be *CDM5.00.02*.

## Types of table

For this discussion, we will divide table types into three categories. As developers, we can change the definition and the contents of the first category (the Fully Modifiable tables). We cannot change the definition of the base fields of the second category, but we can change the contents (the Content Modifiable tables) and add new fields. The third category can be accessed for information, but neither the definition nor the data within is modifiable (the Read-Only tables).

## Fully Modifiable tables

The following tables are included in the Fully Modifiable tables category.

## Master Data

The Master Data table type contains primary data (such as customers, vendors, items, employees, and so on). In any enhancement project, these are the tables that should be designed first because everything else will be based on these tables. When working on a modification, necessary changes to Master Data tables should be defined first. Master Data tables always use card pages as their primary user input method. The Customer table is a Master Data table. A customer record is shown in the following screenshot:

The screenshot shows the Microsoft Dynamics NAV Customer Card page for customer 01121212, Spotsmeyer's Furnishings. The card is divided into several sections:

- General:** Displays basic information like No., Name, Balance (LCY), and Balance Due (LCY).
- Address & Contact:** Shows address details (Address, Post Code, City, Country/Region Code) and contact information (Primary Contact Code, Contact Name, Phone No., Email, Fax No., Home Page).
- Invoicing:** Displays VAT Registration No. and Customer Posting Group.
- Sell-to Customer Sales History:** A grid showing sales activity across various categories:
 

Ongoing Sales Quotes	Ongoing Sales Blanket Orders	Ongoing Sales Orders
0	0	0
Ongoing Sales Invoices	Ongoing Sales Return Orders	Ongoing Sales Credit Memos
0	0	0
Posted Sales Shipments	Posted Sales Invoices	Posted Sales Return Receipts
0	0	0
Posted Sales Credit Memos		
- Customer Statistics:** Summary of sales metrics:
 

Balance (LCY)	Outstanding Orders (LCY)	Shipped Not Invd. (LCY)	Outstanding Invoices (LCY)
0,00	0,00	0,00	0,00

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

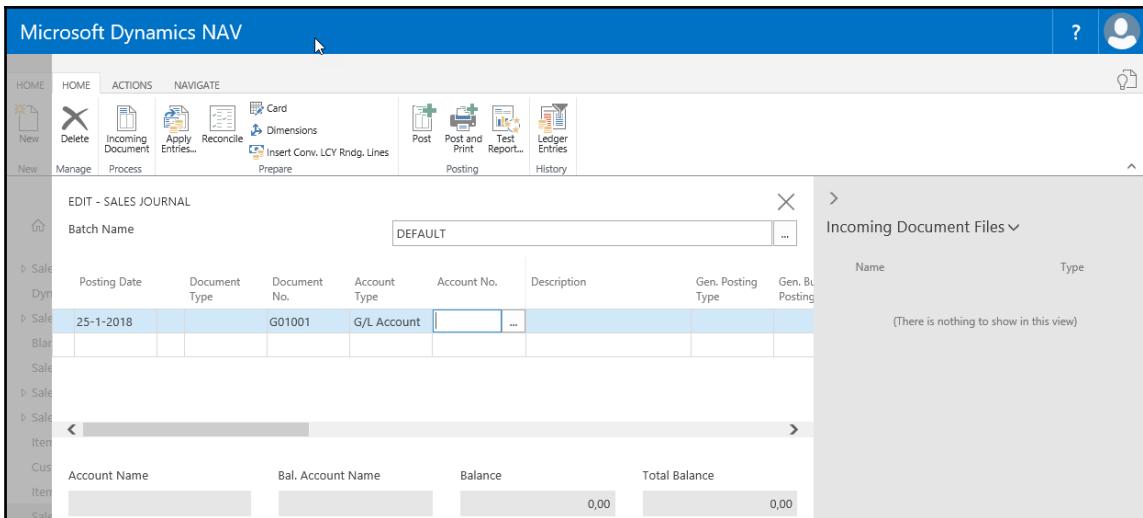
## Journal

The Journal table type contains unposted activity detail, data that other systems refer to as transactions. Journals are where the most repetitive data entry occurs in NAV. In the standard system, all Journal tables are matched with corresponding Template tables (one 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 Worksheet pages as their primary user input method. The following screenshots shows two Journal Entry screens. They both use the General Journal table, but each have quite a different appearance, and are based on different pages and different templates:

The screenshot shows the Microsoft Dynamics NAV interface for editing a General Journal. The top navigation bar includes 'HOME', 'ACTIONS', and 'NAVIGATE' tabs. Under 'ACTIONS', there are buttons for Delete, Reconcile, Import Payroll, Apply Entries..., Preview Posting, Insert Conv. LCY Rndg. Lines, Card, Dimensions, Post, Post and Print, Test Report..., Get Standard Journals..., Save as Standard Journal..., Ledger Entries, and History. The main area is titled 'EDIT - GENERAL JOURNAL' and shows a table of journal entries. The columns include Posting Date, Document Type, Document No., Account Type, Account No., Description, Gen. Posting Type, Gen. Bu. Posting, Dimension Code, Dimension Value Code, and Dimension Value. The first entry is dated 26-1-2018, type G/L Account, no. G00001, account 1220, description 'Packing Machine 2018 Purchase', type Purchase, dimension DEPARTMENT PROD, dimension value production. The second entry is dated 26-1-2018, type G/L Account, no. G00001, account 8210, description 'Boxes for Packing Machine 201 Purchase', type Purchase, dimension DEPARTMENT PROD, dimension value production. The third entry is dated 26-1-2018, type G/L Account, no. G00001, account 8210, description 'Glue for Packing Machine 201 Purchase', type Purchase, dimension DEPARTMENT PROD, dimension value production. The fourth entry is dated 26-1-2018, type G/L Account, no. G00002, account 5710, description 'Invoice no. 156786 for Gasolin', type Purchase, dimension DEPARTMENT PROD, dimension value production. The bottom of the screen shows a summary table with columns for Account Name, Bal. Account Name, Balance, and Total Balance, showing values 110,97 and 0,00 respectively. A note '(There is nothing to show in this view)' is visible. On the right, there are sections for 'Dimensions' and 'Incoming Document Files'.

Comparing the preceding and following screenshots, the differences include not only which fields are visible, but also what logic applies to data entry defaults and validations:



## 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 each be tailored for different purposes. Control information contained in a Template includes the following:

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

For 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 such as **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:

EDIT - GENERAL JOURNAL TEMPLATES + new										Copy VAT Setup to Jnl Lines	
	Name	Description	Type	Rec...	Bal. Account Type	Bal. Account No.	No. Series	Posting No. Series	Source Code	Reason Code	Force Doc. Balance
	ASSETS	... Fixed Asset G/L Journal	Assets	<input type="checkbox"/>	G/L Account		FA-GJNL		FAGLJNL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	CASHRCPT	... Cash receipts	Cash Receipts	<input type="checkbox"/>	G/L Account		GJNL-RCPT		CASHRECINL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	GENERAL	... GENERAL	General	<input type="checkbox"/>	G/L Account		GJNL-GEN		GENJNL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	INTERCOMP	... Intercompany	Intercompany	<input type="checkbox"/>	G/L Account		IC_GJNL		INTERCOMP		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	JOB	... Job G/L Journal	Jobs	<input type="checkbox"/>	G/L Account		GJNL-JOB		JOBGLJNL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	PAYMENT	... Payments	Payments	<input type="checkbox"/>	G/L Account		GJNL-PMT		PAYMENTJNL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	PURCHASE	... Purchases	Purchases	<input type="checkbox"/>	G/L Account		GJNL-PURCH		PURCHJNL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	RECURRING	... Recurring General Journal	General	<input checked="" type="checkbox"/>	G/L Account			GJNL-REC	GENJNL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
	SALES	... Sales	Sales	<input type="checkbox"/>	G/L Account		GJNL-SALES		SALESJNL		<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
				<input type="checkbox"/>							<input type="checkbox"/> <input type="checkbox"/>

In addition to the Templates, there are Batch tables, which allow us to set up any number of batches of data under each journal template. The Batch, Template, Journal Line structure provides a great deal of flexibility in data organization and definition of required fields while utilizing a common underlying table definition (the General Journal).

## Entry tables

The Entry table type contains Posted activity detail, the data other systems call history. NAV data flows from a Journal through a Posting routine into an Entry. A significant advantage of NAV Entry design is the fact that NAV allows retaining all transaction detail indefinitely. While there are routines supporting compression of the Entry data, if at all feasible, we should retain the full historical detail of all activity. This allows users to have total flexibility for historical comparative or trend data analysis.

Entry data is considered accounting data in NAV. We are not allowed to directly enter the data into an Entry or change existing data in an Entry, but must post to an Entry. Posting is done by creating Journal Lines, validating the data as necessary, then posting those journal lines into the appropriate ledgers. Although we can physically force data into an Entry with our developer tools, we should not do so.

When used with accounting an Entry is called Ledger Entry.



When Ledger Entry data is accounting data, we are not permitted to delete this data from an Entry table. Corrections are done by posting adjustments or reversing entries. We can compress or summarize some Entry data (very carefully), which eliminates detail, but we should not change anything that would affect accounting totals for money or quantities.

User views of Entry data are generally through use of list pages. 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 here are representative. The users can utilize page-customization tools (which we will discuss in Chapter 4, *Pages - the Interactive Interfaces*) in order to create personalized page displays in a wide variety of ways. First, we have the Customer Ledger Entries list:

The screenshot shows the Microsoft Dynamics NAV interface with the title bar "Microsoft Dynamics NAV". The top navigation bar includes links for HOME, ACTIONS, and various document management options like Edit List, Show Posted Document, Navigate, Apply Entries, Unapply Entries..., Reverse Transaction..., Incoming Document, Dimensions, Applied Entries, Detailed Ledger Entries, Notes, Links, Open in Excel, and Page. The main content area is titled "EDIT - CUSTOMER LEDGER ENTRIES - 20000 - SELANGORIAN LTD." It displays a table of ledger entries with columns for Posting Date, Document Type, Document No., Customer No., Message to Recipient, and Description. The table shows several entries, with the first one highlighted in blue. A sidebar on the right provides "Customer Ledger Entry Details" including Document (Invoice 103002), Due Date (5-2-2018), Pmt. Discount Date (22-1-2018), and counts for Rem. Entries (0), Applied Entries (0), and Detailed Ledger Entries (1). Below this is a section for "Incoming Document Files" with a note "(There is nothing to show in this view)".

Second is the Item Ledger Entries list:

The screenshot shows the Microsoft Dynamics NAV interface with the title bar "Microsoft Dynamics NAV". The ribbon menu is open at the top, showing "HOME" as the active tab. Below the ribbon, there are several icons for navigating and managing data. The main area displays a grid titled "VIEW - ITEM LEDGER ENTRIES - ITEM 1896-S ATHENS DESK". The grid has columns for Posting Date, Entry Type, Document Type, Document No., Item No., Description, Location Code, Quantity, Invoiced Quantity, Remaining Quantity, and Sales A. The data in the grid is as follows:

Posting Date	Entry Type	Document Type	Document No.	Item No.	Description	Location Code	Quantity	Invoiced Quantity	Remaining Quantity	Sales A
25-1-2018	Transfer	Transfer Shipment	108005	1896-S		OUT. LOG.	25	25	25	
25-1-2018	Transfer	Transfer Shipment	108005	1896-S		RED	-25	-25	0	
22-1-2018	Sale	Sales Shipment	102032	1896-S		RED	-1	0	0	
21-1-2018	Sale	Sales Shipment	102031	1896-S		RED	-6	0	0	
19-1-2018	Sale	Sales Returns	107004	1896-S		RED	1	1	1	
16-1-2018	Sale	Sales Returns	107002	1896-S		GREEN	1	1	1	
13-1-2018	Sale	Sales Shipment	102020	1896-S		RED	-1	-1	0	
10-1-2018	Sale	Sales Shipment	102015	1896-S		GREEN	-1	-1	0	
31-12-2017	Positive Adj...	START	1896-S			GREEN	49	49	48	

The Customer Ledger Entries page displays critical information such as **Posting Date** (the effective accounting date), **Document Type** (the type of transaction), **Customer No.**, and the **Original and Remaining Amount** of the transaction. The record also contains the **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.

The Item Ledger Entries page displays similar information pertinent to inventory transactions. As previously described, **Posting Date**, **Entry Type**, and **Item No.**, as well as the assigned **Location Code** for the item, control the meaning of each transaction. Item Ledger Entries are expressed both in **Quantity** and **Amount (Value)**. The Open entries here are tied to the **Remaining Quantity**, such as material that has been received but is still available in stock. In other words, the Open entries represent current inventory. Both the Customer Ledger Entry and Item Ledger Entry tables have underlying tables that provide additional details for entries affecting values.

## Subsidiary (Supplementary) tables

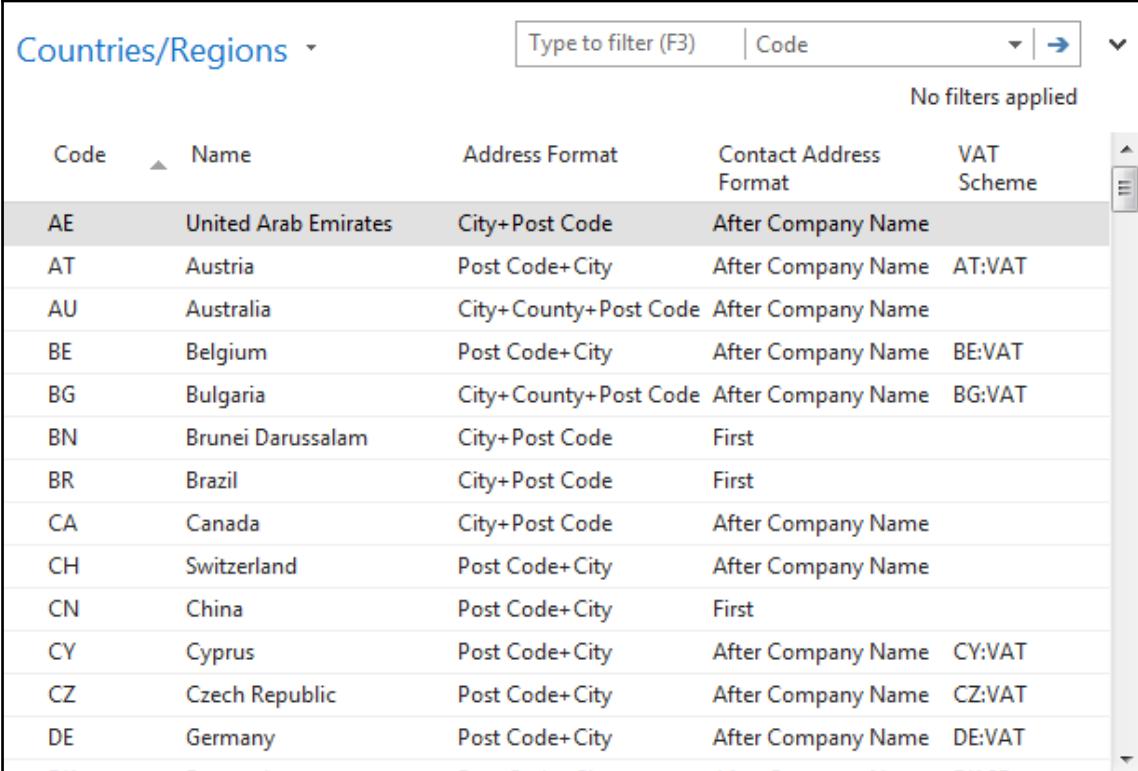
The Subsidiary (also called Supplemental) table type contains lists of codes, descriptions or other validation data. Subsidiary table examples are postal zone codes, country codes, currency codes, currency exchange rates, and so on. Subsidiary tables are often accessed by means of one of the Setup menu options because they must be set up prior to being used for reference purposes by other tables. In our WDTU example, tables 50001 Radio Show Type and 50007 Publisher are Subsidiary tables.

The following screenshots show some sample Subsidiary tables for Location, Country/Region, and Payment Terms. Each table contains data elements that are appropriate for its use as a Subsidiary 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 over time as appropriate:

The screenshot shows a software interface titled "View - Location List". The top navigation bar includes "HOME", "ACTIONS", "REPORT", and a user name "CRONUS Internati...". Below the navigation bar are several action buttons: "New", "Transfer Order", "Edit", "View", "Delete", "Manage", "Location", "Report", "Show Attached", and "Page". A search bar at the top right contains the placeholder "Type to filter (F3)" and a dropdown menu showing "Location List". Below the search bar, a message says "No filters applied". The main area displays a table with two columns: "Code" and "Name". The data rows are:

Code	Name
BLUE	Blue Warehouse
GREEN	Green Warehouse
OUT. LOG.	Outsourced Logistics
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 locations for Awaiting Inspection, Passed Inspection, and Failed Inspection:



A screenshot of a Microsoft Dynamics 365 application window titled "Countries/Regions". The window includes a search bar ("Type to filter (F3) | Code") and a button bar with a refresh icon and a dropdown arrow. Below the title, it says "No filters applied". The main area is a grid table with columns: "Code", "Name", "Address Format", "Contact Address Format", and "VAT Scheme". The rows list various countries with their codes, names, address formats, contact address formats, and VAT schemes. The "Address Format" column uses color coding: grey for AE, blue for AT, green for AU, yellow for BE, red for BG, purple for BN, orange for BR, pink for CA, light blue for CH, dark blue for CN, light green for CY, dark green for CZ, and light orange for DE.

Code	Name	Address Format	Contact Address Format	VAT Scheme
AE	United Arab Emirates	City+Post Code	After Company Name	
AT	Austria	Post Code+City	After Company Name	AT:VAT
AU	Australia	City+County+Post Code	After Company Name	
BE	Belgium	Post Code+City	After Company Name	BE:VAT
BG	Bulgaria	City+County+Post Code	After Company Name	BG:VAT
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	
CN	China	Post Code+City	First	
CY	Cyprus	Post Code+City	After Company Name	CY:VAT
CZ	Czech Republic	Post Code+City	After Company Name	CZ:VAT
DE	Germany	Post Code+City	After Company Name	DE:VAT

The Countries/Regions list in the preceding screenshot is used as validation data, defining the acceptable country codes. It also provides control information for the mailing **Address Format** (general organization address) and the **Contact Address Format** (for an individual contact's address).

## Tables

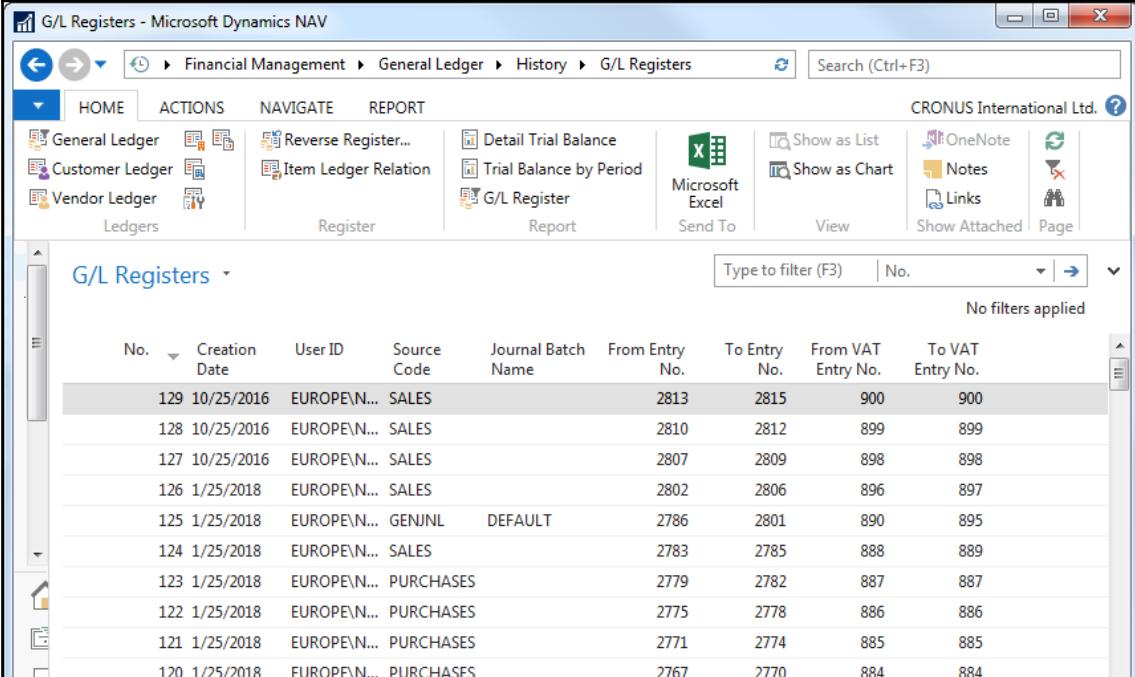
---

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 due in 1 month with a discount of 2% applied to 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:

Code	Due Date Calculation	Discount Date Calc...	Discount %	Calc. Pmt...	Description
14 DAYS	14D		0	<input 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 trail of the physical timing and sequence of postings. This, combined with the full details retained in the Ledger Entry, makes NAV a very auditable system, because we can see exactly what activity was done and when it was done:



The screenshot shows the Microsoft Dynamics NAV interface for the G/L Registers page. The top navigation bar includes links for Financial Management, General Ledger, History, and G/L Registers. Below the navigation bar is a toolbar with various actions like Reverse Register, Item Ledger Relation, Detail Trial Balance, Trial Balance by Period, G/L Register, Microsoft Excel, and OneNote integration. The main area displays a table of G/L Registers with the following data:

No.	Creation Date	User ID	Source Code	Journal Batch Name	From Entry No.	To Entry No.	From VAT Entry No.	To VAT Entry No.
129	10/25/2016	EUROPE\N...	SALES		2813	2815	900	900
128	10/25/2016	EUROPE\N...	SALES		2810	2812	899	899
127	10/25/2016	EUROPE\N...	SALES		2807	2809	898	898
126	1/25/2018	EUROPE\N...	SALES		2802	2806	896	897
125	1/25/2018	EUROPE\N...	GENJNL	DEFAULT	2786	2801	890	895
124	1/25/2018	EUROPE\N...	SALES		2783	2785	888	889
123	1/25/2018	EUROPE\N...	PURCHASES		2779	2782	887	887
122	1/25/2018	EUROPE\N...	PURCHASES		2775	2778	886	886
121	1/25/2018	EUROPE\N...	PURCHASES		2771	2774	885	885
120	1/25/2018	EUROPE\N...	PURCHASES		2767	2770	884	884

The user views the Register through a tabular page, as shown in the previous screenshot. We 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. Another NAV feature, the **Navigate** function, which we will discuss in detail in Chapter 4, *Pages - the Interactive Interfaces*, 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.

## Posted Document

The Posted Document type contains the history version 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 have stored in paper files. A Posted Document looks very similar to the original source document. For example, a Posted Sales 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 the user's life easier by providing related information without requiring a separate lookup action.

First, we see the Sales Order document ready to be posted:

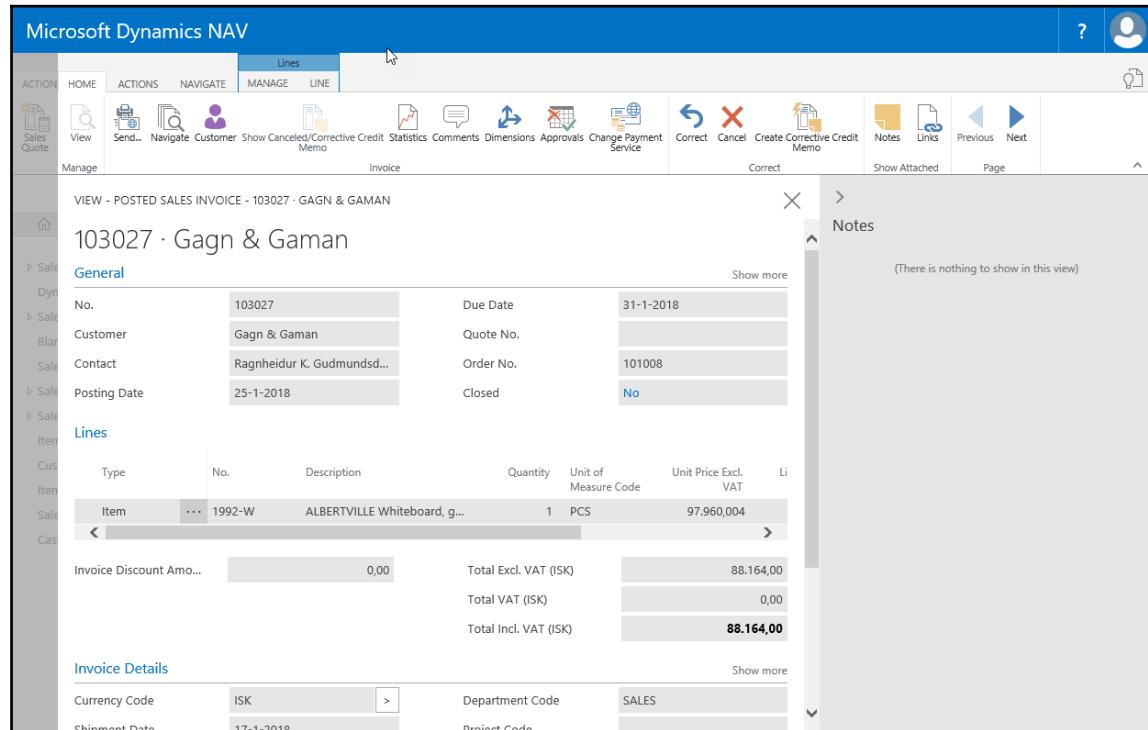
The screenshot shows the Microsoft Dynamics NAV interface for a Sales Order. The main area displays a Sales Order for customer 'Selangorian Ltd.' with three items listed in the 'Lines' section. The summary table at the bottom shows Subtotal Excl. VAT (GBP) as 1.109,07, Total Excl. VAT (GBP) as 1.109,07, Total VAT (GBP) as 277,27, and Total Incl. VAT (GBP) as 1.386,34. To the right of the main screen, a sidebar provides 'Sell-to Customer Sales History' statistics and 'Customer Details' for the selected customer.

	0	0	4
Ongoing Sales Quotes	0	Ongoing Sales Blanket Orders	Ongoing Sales Orders
Ongoing Sales Invoices	0	Ongoing Sales Return Orders	Ongoing Sales Credit Memos
Posted Sales Shipments	4	Posted Sales Invoices	Posted Sales Return Receipts
Posted Sales Credit Memos	2		

Customer Details		
Customer No.	20000	
Phone No.		
Email	selangorian.ltd@cronuscorp.net	
Fax No.		

Tables

The following screenshot is that of the partial shipment Sales Invoice document after the invoice has been posted for the shipped goods:



# Singleton

The Singleton table type contains system or functional application control information, often referred to as Setup. 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. Since a Setup table has only one record, it can have a primary key field that has no value assigned (this is how all the standard NAV Setup tables are designed). The Singleton table design pattern can be found at

Pattern can be found at  
<http://blogs.msdn.com/b/nav/archive/2013/07/19/nav-design-pattern-of-the-week-single-record-setup-table.aspx>.

## Tables

---

The **Inventory Setup** page is as follows:

The screenshot shows the Microsoft Dynamics NAV interface for 'Edit - Inventory Setup'. The top navigation bar includes 'ACTION', 'HOME', 'NAVIGATE', and various icons for Sales Quote, Inventory Periods, Units of Measure, Item Discount Groups, Inventory Posting Setup, Inventory Posting Groups, Item Journal Templates, Notes, and Links. The main content area is titled 'Inventory Setup' and contains several configuration sections:

- General**: Includes fields for Automatic Cost Posting (checkbox), Expected Cost Posting to G/L (checkbox), Automatic Cost Adjustment (dropdown: Never, Item, Day), Copy Comments Order to Shpt. (checkbox), Copy Comments Order to Rcpt. (checkbox), Outbound Whse. Handling Time (text box), Inbound Whse. Handling Time (text box), and Prevent Negative Inventory (checkbox).
- Location**: Includes a 'Location Mandatory' checkbox.
- Dimensions**: Includes a 'Item Group Dimension Code' field.
- Numbering**: Shows mappings between item numbers and posted numbers:

Item Nos.	ITEM1	Posted Invnt. Put-away Nos.	I-PUT+
Non-stock Item Nos.	NS-ITEM	Inventory Pick Nos.	I-PICK
Transfer Order Nos.	T-ORD	Posted Invnt. Pick Nos.	I-PICK+
Posted Transfer Shpt. Nos.	T-SHPT	Inventory Movement Nos.	I-MOVEMENT

## Temporary

The Temporary table is used within objects to hold temporary data. A Temporary table does not exist outside the instance of the object where it is defined using a permanent table as the source of the table definition. The Temporary table has exactly the same data structure as the permanent table after which it is modeled.

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).

Temporary tables are not generally accessible to users except on a display-only basis. They can directly be the target of reports, pages, and XML ports. In general, 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.



There is a Temporary table technique used by advanced developers to define a new temporary table format without consuming a (paid for) licensed table slot. Define the new table in an unlicensed number range. If the current production license allows for tables 50000 through 50099, assign the new layout to 50500 (for example). That layout can then be used to define a temporary table in an object. The layout cannot be used to actually store data in the database, only to provide a convenient data format design for some special intermediate process.

## Content Modifiable tables

The Content Modifiable table category includes only one table type, the System table type.

### System table

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; we cannot create System tables. However, with full developer license rights, we can modify System tables to extend their usage. With full system permissions, we 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 three relate to system security functions:

- **User:** The table of identified users and their security information
- **Permission Set:** The table containing a list of all the permission sets in the database
- **Permission:** The table defining what individual permission sets are allowed to do, based on object permission assignments
- **Access Control:** The table of the security roles that are assigned to each Windows Login

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.
- **Chart:** This defines all of the chart parts that have been set up for use in constructing pages.
- **Web Service:** This lists the pages, queries, and codeunits that have been published as web services.
- **Profile:** This contains a list of all the active profiles and their associated **Role Center** pages. A profile is a collection of NAV users who are assigned to the same **Role Center**.
- **User Personalization:** In spite of its name, this table does not contain information about user personalization that has occurred. Instead, this table contains the link between the user ID and the profile ID, the language, company, and debugger controls. (A personalization is a change in the layout of a page by a user, such as adding or removing fields, page parts, restructuring menus, resizing columns, and so on. This information is in the User Metadata table.)

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

- 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 category:

## Virtual

The Virtual table type is computed at runtime by the system. A Virtual table contains data and is accessed like other tables, but we cannot modify either the definition or the contents of a Virtual table. We can think of the Virtual tables as system data presented in the form of a table so it is readily available to C/AL code. 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 Session tables) provide information about the internal structure and operating activities of our database. A good way to learn more about any of these tables is to create a list or card page bound to the table of interest. Include all the fields in the page layout, **Save** the page and **Run** it. We can then view the field definition and data contents of the target virtual table.

Some virtual tables (such as Date and Integer) provide tools that can be used in our 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 we explore standard NAV reports, we will frequently see the Integer table being used to supply a sequential count in order to facilitate a reporting sequence (often in a limited numeric range such as 1 or 1 to 10).

We cannot see these tables presented in the list of table objects, but can only access them as targets for pages, reports, or variables in C/AL code. Knowledge of the existence, contents, and usage of these Virtual tables are not useful to an end user. However, as developers, we will regularly use some of the Virtual tables. There is educational value in studying the structure and contents of these tables, as well as having the ability to create valuable tools with knowledge of and by accessing of one or more Virtual tables.

## Review questions

1. Which of the following is a correct description of a table in NAV 2017? Choose two:
  - a) A NAV table is the definition of data structure
  - b) A NAV table includes a built-in data entry page
  - c) A NAV table can contain C/AL code but that should be avoided
  - d) A NAV table should implement many of the business rules of a system

2. All primary keys should contain only one data field. True or false?
3. It is possible to link a NAV table to a table outside of the NAV database using what property? Choose one:
  - a) DatabaseLink
  - b) ObjectPointer
  - c) LinkedObject
  - d) C# Codelet
4. System tables cannot be modified. True or false?
5. Which of the following are table triggers? Choose two:
  - a) OnInsert
  - b) OnChange
  - c) OnNewKey
  - d) OnRename
6. Keys can be enabled or disabled in executable code. True or false?
7. Because Setup tables only contain one record, they do not need to have a primary key. True or false?
8. Table numbers intended to be used for customized table objects should only range between 5000 to 9999. True or false?
9. Which of the following tables can be modified by partner developers? Choose three:
  - a) Customer
  - b) Date
  - c) User
  - d) Item Ledger Entry
10. The DropDownList display on a field lookup in the RTC can be changed by modifying the table's **Field Groups**. True or false?
11. Temporary table data can be saved in a special database storage area. True or false?

12. The following Virtual tables are commonly used in NAV development projects. Choose two:
  - a) Date
  - b) GPS Location
  - c) Integer
  - d) Object Metadata
13. SumIndexFields can be used to calculate totals. True or false?
14. Table permissions (for access to another table's data) include which of the following permissions. Choose three:
  - a) read
  - b) sort
  - c) delete
  - d) modify
15. The **TableRelation** property allows a field in one table to reference data in another table. True or false?
16. Tables can be created or deleted dynamically. True or false?
17. Only tables have triggers and only fields have properties. True or false?
18. Ledger Entry data in NAV can be freely updated through either posting routines or direct data entry. True or false?
19. SQL Server for NAV supports SIFT by which mechanism? Choose one:
  - a) SQL SIFT indexes
  - b) SQL dynamic indexes
  - c) SQL indexed views
  - d) SIFT not supported in SQL
20. Reference tables and Virtual tables are simply two different names for the same type of tables. True or false?

## Summary

In this chapter, we focused on the foundation 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 WDTU application. We briefly discussed **Field Groups** and how they are used.

We identified the essential table structure elements including Properties, Object Numbers, Triggers, Keys, and SumIndexFields. Finally, we reviewed the several categories of tables found in NAV 2017.

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 on what can be done with Triggers. Then we will explore how other object types use tables, working towards developing a fully featured NAV development toolkit.

# 3

# Data Types and Fields

*"You can't build a great building on a weak foundation. You must have a solid foundation if you're going to have a strong superstructure."*

- Gordon B. Hinckley

*"Perfection is achieved, not when there is nothing more to add, but rather when there is nothing more to take away."*

- Antoine de Saint-Exupery

The design of an application should begin at the simplest level, with the design of the data elements. The type of data our development tool supports has a significant effect on our design. Because NAV is designed for financially-oriented business applications, NAV data types are financially and business oriented.

In this chapter, we will cover many of the data types we use within NAV. For each data type, we will cover some of the more frequently modified field properties and how particular properties, such as Field Class, are used to support application functionality. Field Class is a fundamental property which defines whether the contents of the field are data to be processed or control information to be interpreted. In particular, we will be covering:

- Basic definitions
- Fields
- Data types
- The FieldClass properties
- Filtering

## Basic definitions used in NAV

First, let's review some basic NAV terminology:

- **Data type:** Defines the kind of data that can be held in a field, whether it be numeric (such as an integer or decimal), text, a table RecordID, time, date, Boolean, and so forth. The data type defines what constraints can be placed on the contents of a field, determines the functions in which the data element can be used (not all data types are supported by all functions), and defines what the results of certain functions will be.
- **Fundamental data type:** A simple, single component structure consisting of a single value at any point in time, for example, a number, a string, or a Boolean value.
- **Complex data type:** A structure made up of, or relating to, simple data types, for example, records, program objects such as Pages or Reports, **Binary Large Objects (BLOBs)**, DateFormulas, external files, and indirect reference variables.
- **Data element:** An instance of a data type, which may be a constant or a variable.
- **Constant:** A data element explicitly defined in the code by a literal value. Constants are not modifiable during execution, only by a developer using C/SIDE. All the simple data types can be represented by constants. Examples are MAIN (Code or Text), 12.34 (Decimal), and +01-312-444-5555 (Text).
- **Variable:** A data element that can have a value assigned to it dynamically during execution. 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 properties required for its particular data type. A field is defined by the values of 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, some that are data dependent and some that are not. Check out the remaining properties by using **Developer and IT Pro Help** from within the table designer.

We can access the properties of a field while viewing the table in the **Design** mode, highlighting the field, and then clicking on the **Properties** icon, clicking on **View | Properties**, or pressing *Shift + F4*. All 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 NAV 2017's Help functions liberally for additional information and examples.

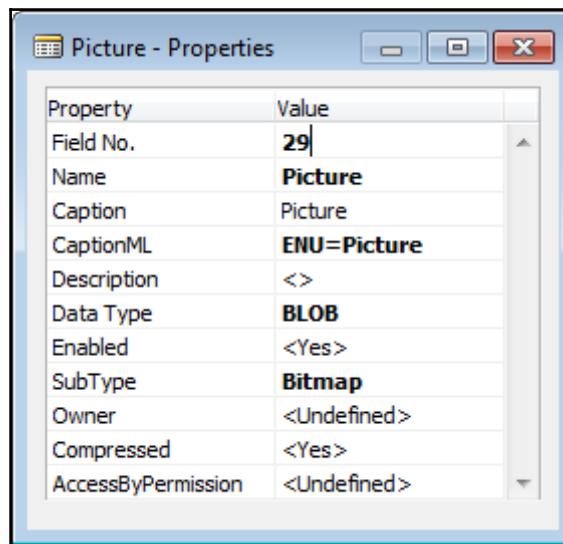
The property value which is enclosed in `< >` (less than and greater than brackets) is the default value for that property. When we set a property to any other value, `< >` should not be present unless they are supposed to be a part of the property value (for example, as part of a Text string value). When a property has been changed from its default value, the NAV 2017 C/AL Editor displays the new property value in bold font.

All 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 by a developer at any time and NAV will automatically ripple that change throughout the system. If no **Caption** value has been defined, the name is used as the default caption when data from this field is displayed. Changing names that are used as literals 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.
- **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 a specific installation of the software.

- **CaptionML**: This defines the multi-language caption for the table. It also identifies the language in use, for example ENU for US English (as seen in the following screenshot).
- **Description**: This is an optional use property, for our internal documentation.
- **Data Type**: This defines what type of data format applies to this field (for example, Integer, Date, Code, Text, Decimal, Option, Boolean).
- **Enabled**: This determines whether or not the field is activated for user generated events. The property defaults to <Yes> and is rarely changed.
- **AccessByPermission**: Determines the permission mask required for a user to access this field in Pages or in the **user interface (UI)**.

The following screenshot shows the properties for the **Picture** field of **Data Type BLOB** in the Company Information table (this field is often used to store a company logo):



The set of properties shown for a BLOB data type field is the simplest set of field properties. After the properties that are shared by all the data types, the BLOB-specific properties are shown:

- **SubType**: 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:** Defines the NAV Server user who owns the object in the BLOB field.
- **Compressed:** This defines whether the data stored in the BLOB field is stored in a compressed format. If we want to access the BLOB data with an external tool (from outside NAV), this property must be set to <No>.

The properties of Code and Text data type fields are quite similar to one another. This is logical since both represent types of textual data. The following screenshots are from the Customer table:

No. - Properties	
Property	Value
Field No.	1
Name	No.
Caption	No.
CaptionML	ENU=No.
Description	<>
Data Type	Code
Enabled	<Yes>
DataLength	20
InitValue	<Undefined>
FieldClass	<Normal>
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>
AccessByPermission	<Undefined>
ExtendedDatatype	<None>
Width	<Undefined>

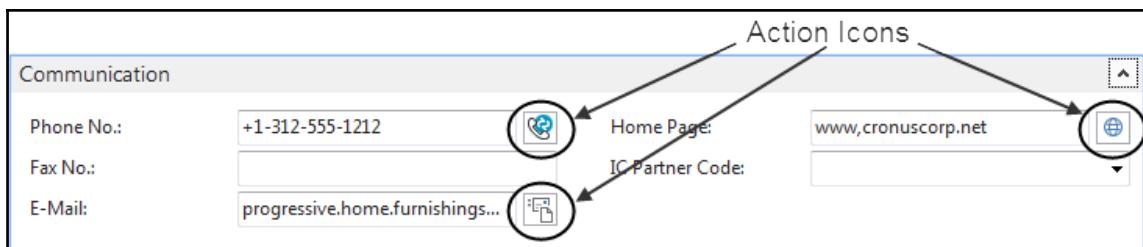
  

Name - Properties	
Property	Value
Field No.	2
Name	Name
Caption	Name
CaptionML	ENU=Name
Description	<>
Data Type	Text
Enabled	<Yes>
DataLength	50
InitValue	<Undefined>
FieldClass	<Normal>
AutoFormatType	<0>
AutoFormatExpr	<>
CaptionClass	<>
Editable	<Yes>
NotBlank	<No>
Numeric	<No>
CharAllowed	<Undefined>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

The following are some common properties between the `Code` and `Text` data types:

- **DataLength:** This specifies the maximum number of characters the field can contain. This is 250 characters in a table, but if no maximum is specified for a `Text` field, there is no length limitation for a variable stored only in memory (working storage). `Code` fields in memory cannot exceed 1,024 characters in length.
- **InitValue:** This is the value that the system will supply as a default when the field is initialized.
- **CaptionClass:** This can be set up by the developer to allow users to dynamically change the caption for a textbox or a checkbox. **CaptionClass** defaults to empty. For more information, refer to **Developer and IT Pro Help**. It is used in base NAV in the **Dimensions** fields.
- **Editable:** This is set to `<No>` when we don't want to allow a field to be edited, for example, if it is a computed or assigned value field that the user should not change. **Editable** defaults to `<Yes>`.
- **NotBlank, Numeric, CharAllowed, DateFormula, and ValuesAllowed:** Each of these allows us to place constraints on the data that can be entered into this field by a user. They do not affect data updates driven by application C/AL code.
- **SQL Data Type:** This applies to the `Code` fields only. **SQL Data Type** defines what data type will be allowed in this particular `Code` field and how it will be mapped to a SQL Server data type. This controls sorting and display. Options are `Varchar`, `Integer`, `BigInteger` or `Variant`. `Varchar` is the default and causes all the data to be treated as text. `Integer` and `BigInteger` allow only numeric data to be entered. A `Variant` can contain any of a wide range of NAV data types. In general, once set, this property should not be changed. These settings should not affect data handling done in SQL Server external to NAV, but the conservative approach is not to make changes here.
- **TableRelation:** This is used to specify a relationship to data in the specified target table. The target table field must be in the primary key. The relationship can be conditional and/or filtered. The relationship can be used for validation, lookups, and data-change propagation.
- **ValidateTableRelation:** If a **TableRelation** field is specified, set this to `<Yes>` in order to validate the relation when data is entered or changed (in other words, confirm that the entered data exists in the target table). If a **TableRelation** field is defined and this property is set to `<No>`, the automatic table referential integrity will not be maintained. Caution: application code can be written that will bypass this validation.

- **TestTableRelation**: A property left over from earlier versions which no longer has use or value.
- **ExtendedDataType**: This property allows the optional designation of an extended data type which automatically receives special formatting and validation. Type options include an e-mail address, a URL, a phone number, report filter, progress bar ratio, or a masked entry (as dots). An action icon may also be displayed as shown in the following screenshot, where there are three fields with ExtendedDataType defined:



Let's take a look at the properties of two more data types, `Decimal` and `Integer`, especially those properties related to numeric content:

- **DecimalPlaces**: This sets the minimum and maximum number of decimal places (`min:max`) for storage and display in a `Decimal` data item. Default is 2 (2:2), the minimum is 0, and the maximum is 255.
- **BlankNumbers**, **BlankZero**, and **SignDisplacement**: These can be used to control the formatting and display of the data field on a page. **BlankNumbers** and **BlankZero** means that all fields of the chosen values are to be displayed as blank. **SignDisplacement** allows data positioning to be shifted for negative values.
- **MinValue** and **MaxValue**: When set, these constrain the range of data values allowed for user entry. The range available depends on the field data type.

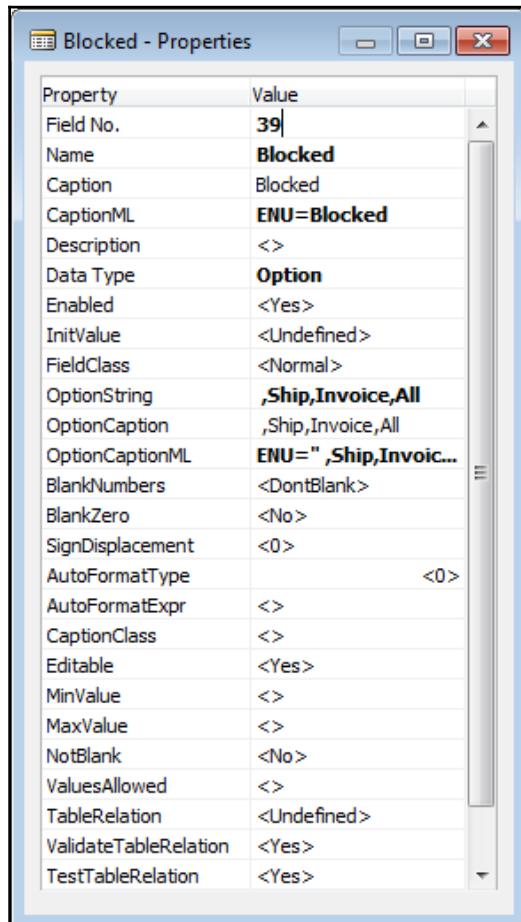
- **AutoIncrement:** This allows for the definition of one Integer field in a table to automatically increment for each record entered. When used, which is not often, it is almost always to support the automatic updating of a field used as the last field in a primary key, enabling the creation of a unique key. Use of this feature does not ensure a contiguous number sequence. When the property is set to <Yes>, the automatic functionality should not be overridden in code:

Credit Limit (LCY) - Proper...		Statistics Group - Properties	
Property	Value	Property	Value
Field No.	<b>20</b>	Field No.	<b>26</b>
Name	<b>Credit Limit (LCY)</b>	Name	<b>Statistics Group</b>
Caption	Credit Limit (LCY)	Caption	Statistics Group
CaptionML	<b>ENU=Credit Limit (...</b>	CaptionML	<b>ENU=Statistics Gr...</b>
Description	<>	Description	<>
Data Type	<b>Decimal</b>	Data Type	<b>Integer</b>
Enabled	<Yes>	Enabled	<Yes>
InitValue	<Undefined>	InitValue	<Undefined>
FieldClass	<Normal>	FieldClass	<Normal>
DecimalPlaces	<Undefined>	BlankNumbers	<DontBlank>
BlankNumbers	<DontBlank>	BlankZero	<No>
BlankZero	<No>	SignDisplacement	<0>
SignDisplacement	<0>	AutoFormatType	<b>&lt;0&gt;</b>
AutoFormatType	<b>1</b>	AutoFormatExpr	<>
AutoFormatExpr	<>	CaptionClass	<>
CaptionClass	<>	Editable	<Yes>
Editable	<Yes>	MinValue	<>
MinValue	<>	MaxValue	<>
MaxValue	<>	NotBlank	<No>
NotBlank	<No>	ValuesAllowed	<>
ValuesAllowed	<>	AutoIncrement	<No>
TableRelation	<Undefined>	TableRelation	<Undefined>
ValidateTableRelation	<Yes>	ValidateTableRelation	<Yes>
TestTableRelation	<Yes>	TestTableRelation	<Yes>
AccessByPermission	<Undefined>	AccessByPermission	<Undefined>
ExtendedDatatype	<None>	ExtendedDatatype	<None>

The properties for an `Option` data type are similar to those of other numeric data types. This is reasonable because an `Option` is stored as an integer, but there are also `Option` specific properties:

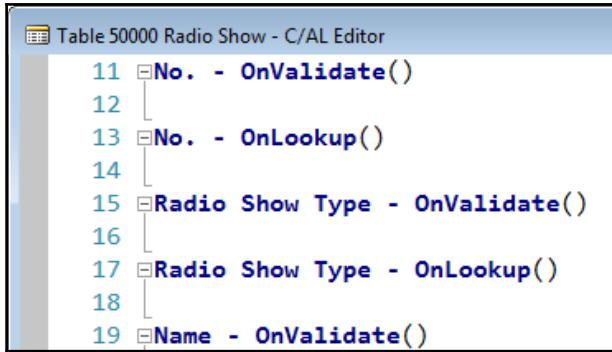
- **OptionString**: Details the text interpretations for each of the stored integer values contained in an `Option` field
- **OptionCaption** and **OptionCaptionML**: These serve the same captioning and multi-language purposes as caption properties for other data types

Internally, options are stored as integers tied to each option's position in the `OptionString` property, starting with position 0, 1, 2, and so on. `OptionString` and `OptionCaption` are shown in the following screenshot:



## Field triggers

To see field triggers, let us look at our **Table 50000 Radio Show**. Open the table in the **Design** mode. Highlight the **No.** field, press *F9*, and you will see the following:



Each 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. The intended use is to validate that the entry conforms to the design parameters for the field. It can also be executed under program control through the use of the `VALIDATE` function (which we will discuss later).
- `OnLookup()`: Lookup behavior can be triggered by pressing *F4* or *Shift + F4* from an ellipsis button, or by clicking on the lookup arrow in a field as shown in the following screenshot:



- If the field's **TableRelation** property refers to a table, then the default behavior is to display a drop-down list to allow the selection of a table entry to be stored in this field. The list will be based on the **Field Groups** defined for the table. We may choose to override that behavior by coding different behavior for a special case. We must be careful, because any entry whatsoever in the body of an `OnLookup()` trigger, even a comment line, will eliminate the default behavior of this trigger.

## Field events

Instead of placing your business code in the **OnValidate** trigger of a field, you can also subscribe to system generated events. For each field we can subscribe to the **OnBeforeValidate** event and the **OnAfterValidate** event. These events are triggered, as their name suggests, before and after the code we place in the **OnValidate** trigger.

Subscribing to these events can be useful to avoid making modifications to the objects shipped by Microsoft so that future updates are easier to implement and merging code is prevented.



More information about events in Dynamics NAV can be found on MSDN at [https://msdn.microsoft.com/en-us/library/mt299505\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/mt299505(v=nav.90).aspx).

## Data structure examples

Some 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 14 Location** and **Table 91 User Setup** both have good examples of the **OnValidate** trigger C/AL code, as do all of the primary master tables (Customer, Vendor, Item, Job, and so on).

## Field numbering

The number of each field within its parent table object is the unique identifier that NAV uses internally to identify that field. We can easily change a field number when we are initially defining a table layout, but after other objects, such as Pages, Reports, or Codeunits, reference the fields in a table, it becomes difficult to change the number of referenced fields. Deleting a field and reusing its field number for a different purpose is not a good idea and can easily lead to programming confusion.



We cannot safely change the definition of, re-number, or delete a field that has data present in the database. The same can be said for reducing the defined size of a field to less than the largest size of data already present in that field. However, if we force the change, the force function will override the system's built-in safeguards. This action can truncate or delete data.

When we add new fields to standard NAV product tables (those shipped with the product), the new field numbers must be in the 50,000 to 99,999 number range, unless we have been explicitly licensed for another number range. Field numbers for fields in new tables that we create may be anything from 1 to 999,999,999.

When a field representing the same data element appears in related tables (for example, **Table 37 Sales Line** and **Table 113 Sales Invoice Line**), the same field number should be assigned to that data element for each of the tables. Not only is this consistent approach easier for reference and maintenance, but it also supports the TRANSFERFIELDS function. TRANSFERFIELDS permits the copying of data from one table's record instance to another table's record instance by doing record to record mapping based on the field numbers.

If we plan ahead and number the fields logically and consistently from the beginning of our design work and provide an entry in the `Description` column for each field, we will create code that's easier to maintain. It's a good idea to leave frequent gaps in field number sequences within a table. This allows the easier insertion of new fields numerically adjacent to related, previously defined fields. In turn, that makes it easier for the next developer to understand the modification's data structure.

See *Object Numbering Conventions* in the *Developer and IT Pro Help* section in the Development Environment for additional information.

## Field and variable naming

In general, the rules for naming fields (data elements in a table) and variables (data elements within the working storage of an object) are the same, and we will discuss them on that basis. The *Developer and IT Pro Help* section in *Naming Conventions* describes many recommended best practices for naming within NAV. Much additional information can also be found in the recently released *C/AL Coding Guidelines* at <https://community.dynamics.com/nav/w/designpatterns/156.cal-coding-guidelines>, including a *How do I* video.

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.

Uniqueness includes not duplicating reserved words or system variables. Refer to the *C/AL Reserved Words* list in the *Developer and IT Pro Help* section.



Avoid using any word as a variable name that appears as an uppercase or capitalized word in either the *Developer and IT Pro Help* section or any of the published NAV technical documentation. For example, we shouldn't use the words *page* or *image* as variable names.

Variable names in NAV are not case sensitive. There is a 128-character length limit on variable names (but still a 30-character length limit on field names in tables). Variable names can contain all ASCII characters except for control characters (ASCII values 0 to 31 and 255) and the double quote (ASCII value 34), as well as some Unicode characters used in languages other than English. Characters outside the standard ASCII set (0-127) may display differently on different systems.



Note that the compiler won't tell us that an asterisk (\*, ASCII value 42) or question mark (? , ASCII value 63) cannot be used in a variable name. However, because both the asterisk and the question mark can be used as wildcards in many expressions, especially filtering, neither one should be used in a variable name.

The first character of a variable name must be a letter A to Z (uppercase or lowercase), or an underscore (\_ , ASCII value 95) - unless the variable name is enclosed in double quotes when it is referenced in code (and such names should be avoided). Alphabets other than the 26-character English alphabet may interpret the ASCII values to characters other than A to Z, and may include more than 26 characters. A variable name's first character can be followed by any combination of the legal characters.

If we use any characters other than the A-Z alphabet, numerals, and underscores, we must surround our variable name with double quotes each time we use it in C/AL code (for example, `Cust List`, which contains an embedded space, or `No`, which contains a period).

When we create a variable with a complex data type such as **Record**, **Report**, **Codeunit**, **Page**, **XMLport**, **Query**, or **Testpage**, and do not supply a name, the variable name will be automatically generated according to *C/AL Coding Guidelines* by the Development Environment.

See *Naming Conventions* in the *Developer and IT Pro Help* section for additional C/AL variable naming guidance.

# Data types

We are going to segregate the data types into several groups. We will first look at fundamental data types, then complex data types.

## Fundamental data types

Fundamental data types are the basic components from which the complex data types are formed. They are grouped into numeric, string, and date/time data types.

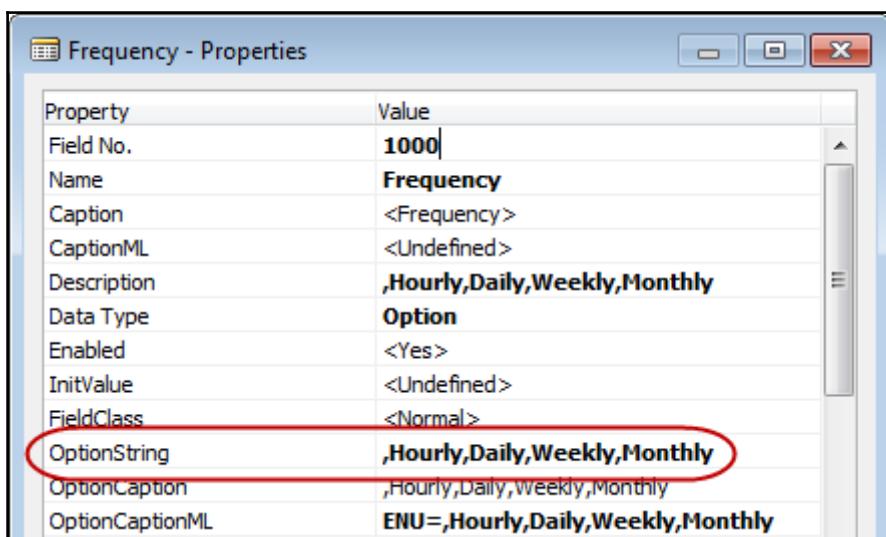
### Numeric data

Just like other systems, NAV supports several numeric data types. The specifications for each NAV data type are defined for NAV independent of the supporting SQL Server database rules. But some data types are stored and handled somewhat differently from a SQL Server point of view than the way they appear to us as NAV developers and users. For more details on the SQL Server-specific representations of various data elements, refer to the *Developer and IT Pro Help* section. Our discussion will focus on the NAV representation and handling for each data type.

The various numeric data types are as follows:

- **Integer:** An integer number ranging from -2,147,483,646 to +2,147,483,647.
- **Decimal:** A decimal number in the range of +/- 999,999,999,999,999,999. 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 +2,147,483,647. An option is normally represented in the body of our C/AL code as an option string. We can compare an option to an integer in C/AL rather than using the option string, but that's not 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. Since the maximum length of this string is 250 characters, the practical maximum number of choices for a single option is less than 125. The currently selected choice within the set of options is stored in the option field 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 if none has been chosen. An example of this (**blank,Hourly,Daily,Weekly,Monthly**) follows:



- Boolean: A Boolean variable is stored as 1 or 0, in C/AL code is 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** (language dependent), **P** or blank, **True**, or **False**.
- BigInteger: 8-byte Integer as opposed to the 4 bytes of Integer. BigIntegers are for very big numbers (from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807).

- Char: A numeric code between 0 and 65,535 (hexadecimal FFFF) representing a single 16-bit Unicode character. 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 individual text character values. Char variables cannot be defined as permanent variables in a table, but only as working storage variables within C/AL objects.
- Byte: A single 8-bit ASCII character with a value of 0 to 255. Byte variables can operate either as text or as numbers. Numeric operations can be done on Byte variables. Byte variables can also be defined with individual text character values. Byte variables cannot be defined as permanent variables in a table, but only as working storage variables within C/AL objects.
- Action: A variable returned from a PAGE RUNMODAL function or RUNMODAL (Page) function that specifies what action a user performs on a page. Possible values are OK, Cancel, LookupOK, LookupCancel, Yes, No, RunObject and RunSystem.
- ExecutionMode: Specifies the mode in which a session is running. The possible values are Debug or Standard.

## 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 field can be from 1 to 250 characters long. In working storage within an object, a Text variable can be any length if there is no length defined. If a maximum length is defined, it must not exceed 1024. NAV 2017 does not require a length to be specified, but if we define a maximum length, it will be enforced. When calculating the length of a record for design purposes (relative to the maximum record length of 8000 bytes), the full defined field length should be counted.
- Code: Help says the length constraints for Code variables are the same as those for text variables, but the C/AL Editor enforces limits of 1 to 250 characters in length. All the letters are automatically converted to uppercase when data is entered into a Code variable; any leading or trailing spaces are removed.

## Date/Time data

Date/Time data display is region specific, in other words, the data is displayed according to local standards for date and time display. 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 to December 31, 9999. A 0D (numeral zero, letter D) represents an undefined date (stored as a SQL Server **DateTime** field) interpreted as January 1, 1753. According to the *Developer and IT Pro Help* section, NAV 2017 supports a date of 1/1/0000 (presumably as a special case for backward compatibility but not supported by 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, 011917D or 01192017D both represent January 19, 2017. 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 20 if the two digit year is from 00 to 29, or 19 if the year is from 30 through 99.

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 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 choice, 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, 2017 would display as 12/31/17 (depending on the date format masking), and the closing date entry would display as C12/31/17. All the C12/31/17 ledger entries would appear after all normal 12/31/17 ledger entries. The following screenshot shows two 2016 closing date entries mixed with normal entries from January through April 2017. (This data is from CRONUS demo. The 2016 Closing entries have an **Opening Entry** description showing that these were the first entries for the demo data in the respective accounts. This is not a normal set of production data.):

General Ledger Entries										Type to filter (F3)	Posting Date	▼   ↗	▼
Posti... Date	Document Type	Document No.	G/L Acco...	Description	Gen. Postin...	Gen. Bus. Posting ...	Gen. Prod. Posting ...	Amount	Bal	Ac			
12/31/2017		00-12A	1140	Depreciation 2017				-60,814.84	G/L				
C12/31/2016		START	1210	Opening Entry				582,872.18	G/L				
1/1/2017	Invoice	108018	1220	Order 106018	Purchase	DOMESTIC	MISC	6,600.00	G/L				
2/1/2017	Invoice	108019	1220	Order 106019	Purchase	DOMESTIC	MISC	4,512.00	G/L				
2/1/2017	Invoice	108020	1220	Order 106022	Purchase	DOMESTIC	MISC	7,140.00	G/L				
3/1/2017	Invoice	108021	1220	Order 106020	Purchase	DOMESTIC	MISC	3,024.00	G/L				
4/1/2017	Invoice	108022	1220	Order 106021	Purchase	DOMESTIC	MISC	3,840.00	G/L				
C12/31/2016		START	1240	Opening Entry				-362,263.84	G/L				

- **Time:** This contains an integer number, which is interpreted on a 24-hour clock, in milliseconds plus 1, from 00:00:00 to 23:59:59:999. A 0T (numeral zero, letter T) represents an undefined time and is stored as 1/1/1753 00:00:00.000.
- **DateTime:** This represents a combined Date and Time, stored in **Coordinated Universal Time (UTC)** and always displays the local time (that is, the local time on our system). **DateTime** fields do not support NAV Closing dates. **DateTime** is helpful for an application that needs to 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, but dates earlier than January 1, 1754 cannot be entered (don't test with dates late in 9999 as an intended advance to the year 10000 won't work). Assigning a date of 0DT yields an undefined or blank **DateTime**.
- **Duration:** This represents the positive or negative difference between two **DateTime** values, in milliseconds, stored as a BigInteger. Durations are automatically output in the text format *DDD days HH hours MM minutes SS seconds*.

## Complex data types

Each complex data type 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:** Refers to any standard Windows file outside the NAV database. There is a reasonably complete set of functions to allow a range of actions, including creating, deleting, opening, closing, reading, writing, and copying data files. For example, we could create our own NAV routines in C/AL to import or export data from or to a file that has been created by some other application.



With the three tier architecture of NAV 2017, business logic runs on the server, not the client. We need to keep this in mind whenever we are referring to local external files, because they will be on the server by default. The use of Universal Naming Convention (UNC) paths can make this easier to manage.

- **Record:** Refers to a single data row within a NAV table consisting of individual fields. Quite often, multiple variable instances of a Record (table) are defined in working storage to support a validation process, allowing access to different records within the table at one time in the same function.

## Objects

**Page**, **Report**, **Codeunit**, **Query**, and **XMLPort** each represent an object data type. Object data types are used when there is a need for reference to an object or a function in another object. Examples are as follows:

- Invoking **Report** or **XMLPort** from a **Page** or a **Report** object
- Calling a function for data validation or processing is coded as a function in a table or a Codeunit

## Automation

The following are Automation data types (these are not supported by the NAV Web client). OCX and Automation data types are supported in NAV 2017 for backward compatibility only:

- **OCX:** Allows the definition of a variable that represents and allows access to an ActiveX or OCX custom control. Such a control is typically an external application object which we can invoke from our NAV object.
- **Automation:** Allows the definition of a variable that we may access similarly to an OCX. The application must act as an Automation Server and must be registered with the NAV client or server calling it. For example, we can interface from NAV into the various Microsoft Office products (Word, Excel, and so on) by defining them in Automation variables.
- **DotNet:** Allows the definition of a variable for .NET Framework interface types within an assembly. This supports the access of .NET Framework type members, including methods, properties and constructors from C/AL. These can be members of the global assembly cache or custom assemblies.

## Input/Output

The following are the **Input/Output** data types:

- **Dialog:** Supports the definition of a simple UI window without the use of a **Page** object. Typically, the **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** type data item.
- **InStream and OutStream:** These allow reading from and writing to external files, BLOBS, and objects of the Automation and .Net data types.

## DateFormula

**DateFormula** provides for the definition and storage of a simple, but clever, set of constructs to support the calculation of runtime-sensitive dates. A **DateFormula** data type is stored in a non-language dependent format, thus supporting multi-language functionality. A **DateFormula** data type is a combination of:

- Numeric multipliers (for example 1, 2, 3, 4, and so on)
- Alpha time units (all must be uppercase)

- $D$  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
- $Y$  for year:
  - $CM$  for current month,  $CY$  for current year,  $CW$  for current week
- Math symbols interpretation:
  - $+$  (plus) as in  $CM + 10D$  means the current month end plus 10 days (in other words, the 10th of next month)
  - $-$  (minus) as in  $(-WD3)$  means the date of the previous Wednesday (the third day of the past week)
- Positional notation ( $D15$  means the 15th day of the month and  $15D$  means 15 days)

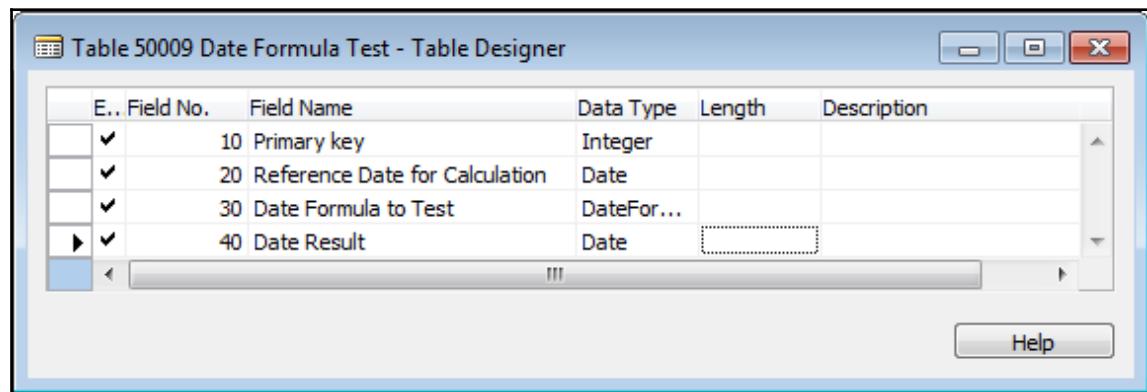
Payment terms for invoices support the full use of **DateFormula**. All **DateFormula** results are expressed as a date based on a reference date. The default reference date is the system date, not the work date.

Here are some sample **DateFormulas** and their interpretations (displayed dates are based on the US calendar) with a reference date of July 10, 2017, a Friday:

- $CM$ , the last day of the current month, 07/31/15
- $CM + 10D$ , the 10th of next month, 08/10/15
- $WD6$ , the next sixth day of the week, 07/11/15
- $WD5$ , the next fifth day of the week, 07/17/15
- $CM - M + D$ , the end of the current month minus one month plus one day, 07/01/15
- $CM - 5M$ , the end of the current month minus five months, 02/28/15

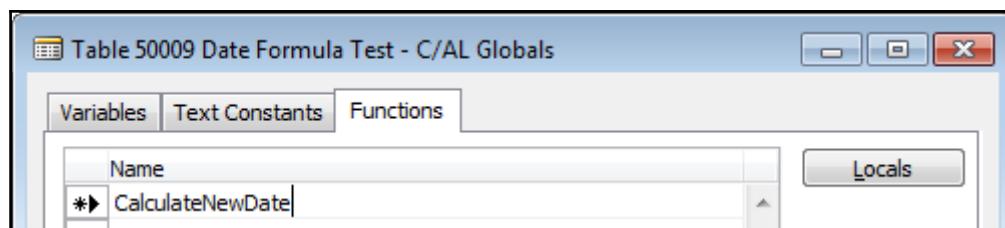
Let us take the opportunity to use the **DateFormula** data type to learn a few NAV development basics. We will do so by experimenting with some hands-on evaluations of several **DateFormula** values. We will create a table to calculate dates using **DateFormula** and **Reference Date**.

Go to **Tools | Object Designer | Tables**. Click on the **New** button and define the fields shown in the following screenshot. Save it as Table 50009, named Date Formula Test. After we've finished this test, we will save the table for some later testing:

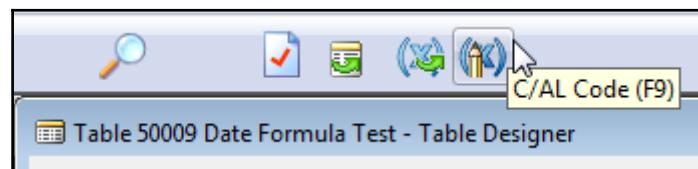


Now we will add some simple C/AL code to our table so that, when we enter or change either the **Reference Date** or the **DateFormula** data type, we can calculate a new result date.

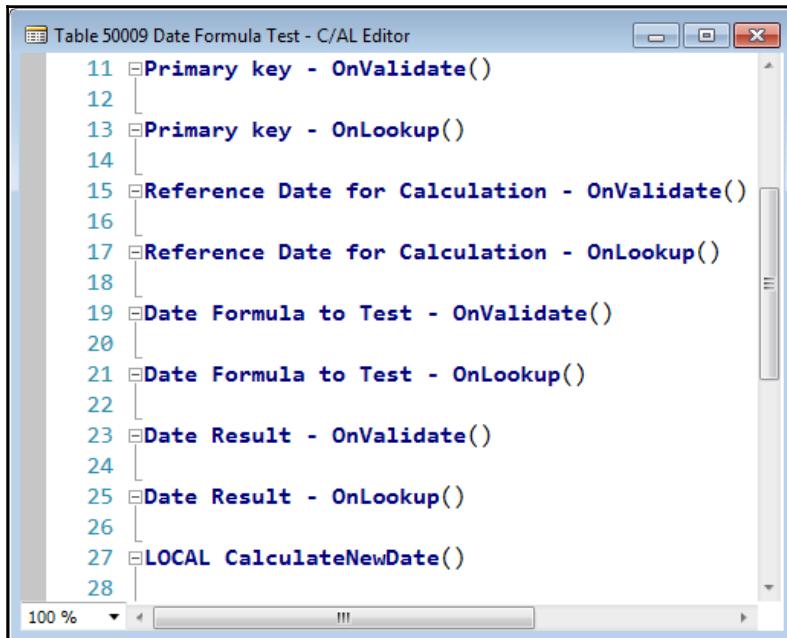
First, access the new table via the **Design** button, then go to the global variables definition form through the **View** menu option, the sub-option **C/AL Globals**, and finally, choose the **Functions** tab. Type in our new function name as `CalculateNewDate` on the first blank line, as shown in 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 us to the following screen, where we see all the field triggers, plus the trigger for the new function that we've just defined. The table triggers are not visible, unless we scroll up to show them. Notice that our new function was defined as a LOCAL function. This means that it cannot be accessed from another object unless we change it to a Global function:



The screenshot shows the C/AL Editor window titled "Table 50009 Date Formula Test - C/AL Editor". The code list contains the following entries:

```
11 Primary key - OnValidate()
12 
13 Primary key - OnLookup()
14 
15 Reference Date for Calculation - OnValidate()
16 
17 Reference Date for Calculation - OnLookup()
18 
19 Date Formula to Test - OnValidate()
20 
21 Date Formula to Test - OnLookup()
22 
23 Date Result - OnValidate()
24 
25 Date Result - OnLookup()
26 
27 LOCAL CalculateNewDate()
28
```

The code is written in a structured format with line numbers on the left and descriptive text on the right.

Because our goal now is to focus on experimenting with the **DateFormula** data type, we will not go into detail explaining the logic we are creating. The logic we're going to code follows:



When an entry is made (new or changed) in either the **Reference Date** field or in the **Date Formula to Test** field, invoke the **CalculateNewDate** function to calculate a new **Result Date** value based on the entered data.

First, we will create the logic within our new function, **CalculateNewDate()**, to evaluate and store a **Date Result** based on the **DateFormula** and **Reference Date** data type that we enter into the table.

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

The screenshot shows the Microsoft Dynamics NAV C/AL Editor window titled "Table 50009 Date Formula Test - C/AL Editor". The code is listed below:

```
11 Primary key - OnValidate()
12
13 Primary key - OnLookup()
14
15 Reference Date for Calculation - OnValidate()
16 CalculateNewDate;
17
18 Reference Date for Calculation - OnLookup()
19
20 Date Formula to Test - OnValidate()
21 CalculateNewDate;
22
23 Date Formula to Test - OnLookup()
24
25 Date Result - OnValidate()
26
27 Date Result - OnLookup()
28
29 LOCAL CalculateNewDate()
30 "Date Result" := CALCDATE("Date Formula to Test","Reference Date for Calculation");
31
```

If you get an error message of any type when you close and save the table, you probably have not copied the C/AL code exactly as it is shown in the screenshot (also shown in the following block of code for ease of copying):

```
CalculateNewDate;
"Date Result" := CALCDATE("Date Formula to Test","Reference Date
for Calculation");
```

This code will cause the `CalculateNewDate()` function to be called via the `OnValidate` trigger when an entry is made in either the **Reference Date for Calculation** or the **Date Formula to Test** fields. The function will place the result in the **Date Result** field. The use of an integer value in the redundantly named **Primary Key** field allows us to enter any number of records into the table (by manually numbering them 1, 2, 3, and so forth).

Let's experiment with several different date and date formula combinations. We will access the table via the **Run** button. This will cause NAV to generate a default format page and run it in the Role Tailored Client.

Enter a primary key value of 1 (one). In **Reference Date for Calculation**, enter either an upper or lower case T for Today, 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 (the numeral 1 followed by uppercase or lowercase letter D, C/SIDE will take care of making it upper case) in the **Date Formula to Test** field. We will see that the **Date Result** field contents are changed to be one day beyond the date in the **Reference Date for Calculation** field.

Now for another test entry, start with a 2 in the **Primary Key** field. Again, enter the letter T (for Today) in the **Reference Date for Calculation** field, and enter the letter W (for Week) in the **Date Formula to Test** field. We will get an error message telling us that our formulas should include a number. Make the system happy and enter 1W. We'll now see a date in the **Date Result** field that is one week beyond our system date.

Set the system's Work Date to a date in the middle of a month (remember, we discussed setting the Work Date in Chapter 1, *Introduction to NAV 2017*). 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. Our result date will be the last day of our 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 our result date will be the first day of our Work Date month. Notice that the **DateFormula** logic handles month end dates correctly, even including leap years. Try starting with a date in the middle of February 2016 to confirm that:

Primary key	Reference Date for ...	Date Formula to Test	Date Result
1	2/1/2017	1D	2/2/2017
2	2/1/2017	1W	2/8/2017
3	3/10/2017	CM	3/31/2017
4	3/10/2017	-CM	3/1/2017
5	2/15/2016	CM	2/29/2016

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? We get an error message stating that **You cannot base a date calculation on an undefined date**. In other words, NAV cannot make the requested calculation without a **Reference Date** data type. Before we put this function into production, we want our code to check for a **Reference Date** data type before calculating. We could default an empty date to the System Date or the Work Date and avoid this particular error.

The preceding and following screenshots show different sample calculations. Build on these sample calculations and then experiment more on your own:

5	2/15/2016	CM	2/29/2016
6	2/15/2017	CM	2/28/2017
7	4/6/2017	1M	5/6/2017
8	4/6/2017	-1M	3/6/2017
9	9/3/2017	-1W-1D	8/26/2017
10	8/26/2017	1W+1D	9/3/2017
42	3/10/2017	-1W	3/3/2017
55	1/15/2017	-CM-1D	12/31/2016
56	1/15/2017	-1Y-CM	1/1/2016

We can create a variety of different algebraic date formulae and get some very interesting and useful results. One NAV user business has due dates for all invoices of 10th of the next month. The invoices are dated on the dates they are actually printed, at various times throughout the month. But by using **DateFormula** of `CM + 10D`, each invoice due date is always automatically calculated to be the 10th of the next month.

Don't forget to test with `WD` (weekday), `Q` (quarter), and `Y` (year) as well as `D` (day), `W` (week), and `M` (month). For our code to be language independent, we should enter the date formulae with `< >` delimiters around them (example: `<1D+1W>`). NAV will translate the formula into the correct language codes using the installed language layer.

Although our focus for the work we just completed was the **Date Formula** data type, we've accomplished a lot more than simply learning about that one data type:

- We created a new table just for the purpose of experimenting with a C/AL feature that we might use. This is a technique that comes in handy when we are learning a new feature, trying to decide how it works or how we might use it.

- We put some critical `OnValidate` 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 `LOCAL` function. That function is then called from all the places to which it applies.
- We did our entire test with a table object and a default tabular page that is automatically generated when we run a table. We didn't have to create a supporting structure to do our testing. Of course, when we are designing a change to a complicated existing structure, we will have a more complicated testing scenario. One of our goals will always be to simplify our testing scenarios, both to minimize the setup effort and to keep our test narrowly focused on the specific issue.
- Finally, and most specifically, 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 data types

The following data types are used for advanced functionality in NAV, sometimes supporting an interface with an external object:

- **RecordID**: It contains the object number and primary key of a table.
- **RecordRef**: It identifies a record/row in a table. `RecordRef` can be used to obtain information about the table, the record, the fields in the record, and the currently active filters on the table.
- **FieldRef**: It identifies a field in a table, thus it allows access to the contents of that field.
- **KeyRef**: It identifies a key in a table and the fields in that key.



Since the specific record, field, and key references are assigned at runtime, `RecordRef`, `FieldRef`, and `KeyRef` are used to support logic which can run on tables that are not specified at design time. This means that one routine built on these data types can be created to perform a common function for a variety of different tables and table formats.

- **Variant**: It defines variables typically used for interfacing with Automation and DotNet objects. Variant variables can contain data of various C/AL data types for passing to an Automation or DotNet object as well as external Automation data types that cannot be mapped to C/AL data types.
- **TableFilter**: It is used for variables which only can be used for setting security filters from the `Permissions` table.

- **Transaction Type:** It has optional values of **UpdateNoLocks**, **Update**, **Snapshot**, **Browse**, and **Report** which define SQL Server behavior for a NAV Report or **XMLport** transaction from the beginning of the transaction.
- **BLOB:** It can contain either specially formatted text, a graphic in the form of a bitmap, or other developer-defined binary data up to 2 GB in size. **Binary Large Objects (BLOBs)** can only be included in tables; they cannot be used to define working storage variables. See *Developer and IT Pro Help* for additional information.
- **BigText:** It can contain large chunks of text up to 2 GB in size. The **BigText** variables can only be defined in the working storage within an object, but not included in tables. The **BigText** variables cannot be directly displayed or seen in the debugger. There is a group of special functions that can be used to handle **BigText** data. See *Developer and IT Pro Help* for additional information.



To handle text strings in a single data element greater than 250 characters in length, use a combination of BLOB and BigText variables.

- **GUID:** It is used to assign a unique identifying number to any database object. **Globally Unique Identifier (GUID)** is 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.
- **TestPage:** It is used to store a test page which is a logical representation of a page that does not display a UI. Test pages are used when doing NAV application testing using the automated testing facility that is part of NAV.

## Data type usage

About 40% of the data types can be used to define the data either stored in tables or in working storage data definitions (that is, in a Global or Local data definition within an object). Two data types, BLOB and TableFilter, can only be used to define table stored data, but not working storage data. About 60% of the data types can only be used for working storage data definitions.

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

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

## FieldClass property options

Almost all data fields have a `FieldClass` property. `FieldClass` has as much effect on the content and usage of a data field as does the data type, in some instances more. In the next chapter, we'll cover most of the field properties, but we'll discuss the `FieldClass` property options now.

### FieldClass - Normal

When the `FieldClass` property is `Normal`, the field will contain the type of application data that's typically stored in a table, the contents we would expect based on the data type and various properties.

### FieldClass - FlowField

`FlowField` must be dynamically calculated. `FlowFields` are virtual fields stored as metadata; they do not contain data in the conventional sense. A `FlowField` contains the definition of how to calculate (at runtime) the data that the field represents and a place to store the result of that calculation. Generally, the `Editable` property for a `FlowField` is set to `No`.

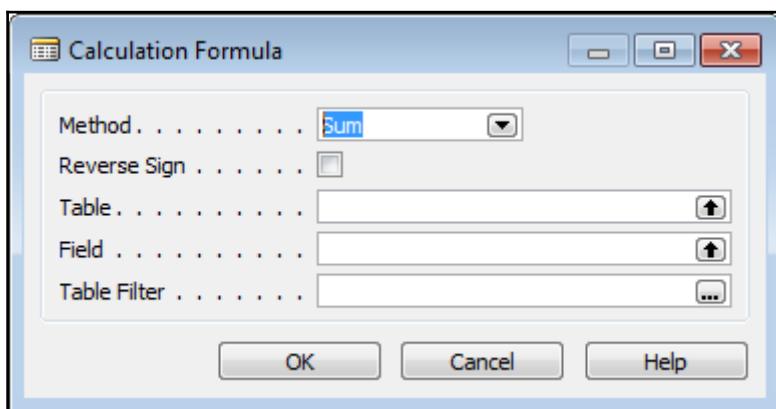
Depending on the `CalcFormula` method, this could be a value, a reference lookup, or a Boolean value. When the `CalcFormula` method is `Sum`, the `FieldClass` property connects a data field to a previously defined `SumIndexField` in the table defined in `CalcFormula`. `FlowField` processing speed will be significantly affected by the key configuration of the table being processed. While we must be careful not to define extra keys, having the right keys defined will have a major effect on system performance, and thus on user satisfaction.

A `FlowField` value is always 0, blank or false, unless it has been calculated. If a `FlowField` is displayed directly on a page, it is calculated automatically when the page is rendered. `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 discussions in *Chapter 5, Queries and Reports* on Reports and *Chapter 8, Advanced NAV Development Tools* on XMLports). In all other cases, a `FlowField` must be forced to calculate using the `C/AL RecordName.CALCFIELDS(FlowField1, [FlowField2], ...)` function or by use of the function `SETAUTOCALCFIELDS`. This is also true if the underlying data is changed after the initial display of a page (that is, the `FlowField` must be recalculated to take a data change into account).

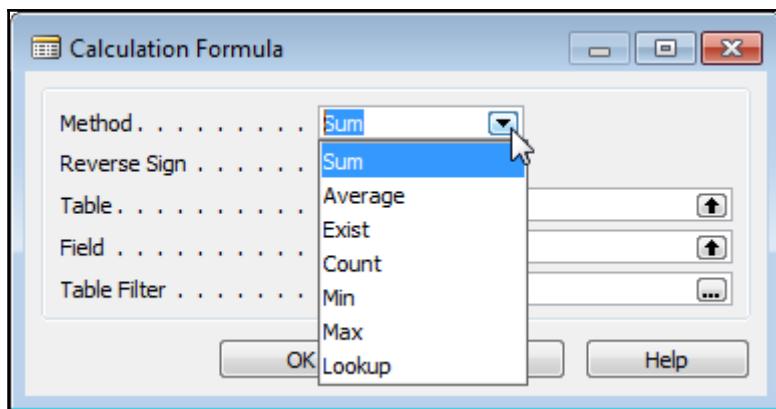


Because a FlowField does not contain actual data, it cannot be used as a field in a key; in other words, we cannot include a FlowField as part of a key. Also, we cannot define a FlowField that is based on another FlowField except in special circumstances.

When a field has its `FieldClass` property set to `FlowField`, another directly associated property becomes available: `CalcFormula` (conversely, the `AutoIncrement`, and `TestTableRelation` properties disappear from view when `FieldClass` is set to `FlowField`). `CalcFormula` is the place where we can define the formula for calculating the FlowField. On the `CalcFormula` property line, there is an ellipsis button. Clicking on that button will bring up the following screen:



Click on the drop-down button to show the seven FlowField methods:



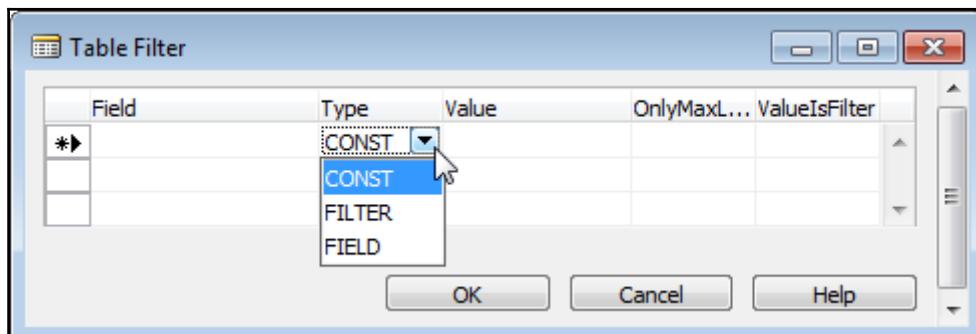
The seven FlowFields are described in the following table:

FlowField method	Field data type	Calculated Value as it applies to the specified set of data within a specific column (field) in a table
<b>Sum</b>	Decimal	The sum total
<b>Average</b>	Decimal	The average value (the sum divided by the row count)
<b>Exist</b>	Boolean	Yes or No / True or False - does an entry exist?
<b>Count</b>	Integer	The number of entries that exist
<b>Min</b>	Any	The smallest value of any entry
<b>Max</b>	Any	The largest value of any entry
<b>Lookup</b>	Any	The value of the specified entry

The **Reverse Sign** control allows us to change the displayed sign of the result for FlowField types **Sum** and **Average** only; the underlying data is not changed. If a **Reverse Sign** is used with the FlowField type **Exist**, it changes the effective function to *does not Exist*.

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

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



When we click on the **Field** column, we 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 define on this line, which must be consistent with the **Type** choices described in the following table:

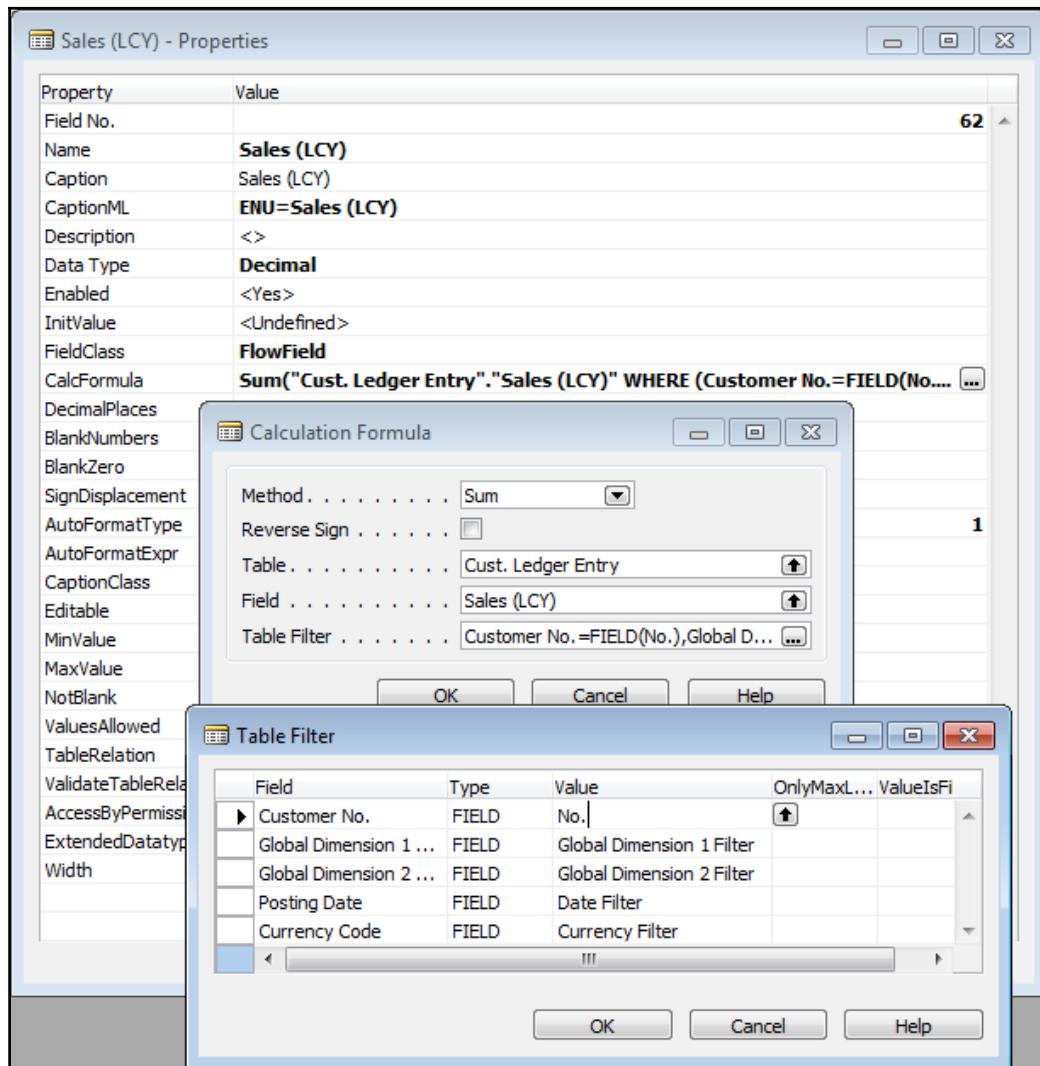
Filter type	Value	Filtering action	OnlyMax - Limit	Values - Filter
Const	A constant which will be defined in the <b>Value</b> field	Uses the constant to filter for equally valued entries		
Filter	A filter which will be spelled out as a literal in the <b>Value</b> field	Applies the filter expression from the <b>Value</b> 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 <b>OnlyMaxLimit</b> parameter is <b>True</b> , then the FlowFilter range will be applied on the basis of only having a MaxLimit, that is, having no bottom limit. This is useful for the date filters for Balance Sheet data. (For an example, see <i>Field 31 - Balance at Date in Table 15 - G/L Account</i> )	True	False
		Causes the contents of the specified field to be interpreted as a filter (For an example, see <i>Field 31 - Balance at Date in Table 15 - G/L Account</i> for an example)	True or False	True

## FieldClass - FlowFilter

FlowFilters control the calculation of FlowFields in the table (when the FlowFilters are included in the **CalcFormula**). FlowFilters do not contain permanent data, but instead 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 us to provide flexible data selection functions to the users. The user does not need to have a full understanding of the data structure to apply filtering in intuitive ways to both the primary data table and also to subordinate data. Based on our C/AL code design, FlowFilters can be used to apply filtering on multiple tables subordinate to a parent table. Of course, it is our responsibility as developers to make good use of this tool. As with many C/AL capabilities, a good way to learn more is by studying standard code as designed by the Microsoft developers of NAV, then experimenting.

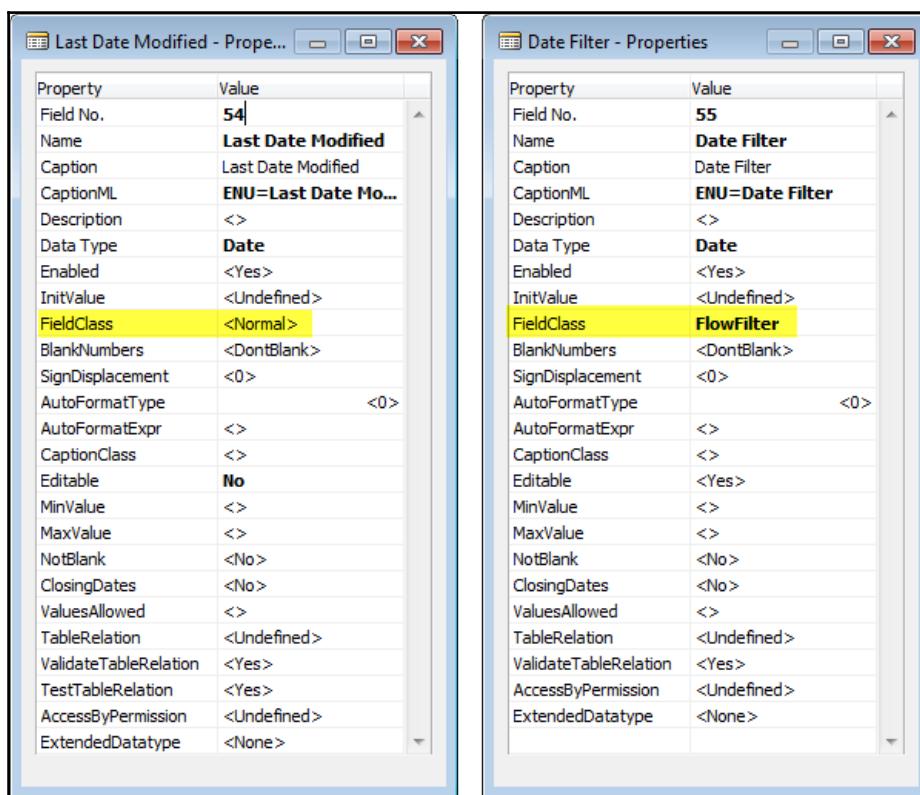
A number of good examples on the use of FlowFilters can be found in the Customer (Table 18) and Item (Table 27) tables. In the Customer table, some of the FlowFields using FlowFilters are Balance, Balance (LCY), Net Change, Net Change (LCY), Sales (LCY), and Profit (LCY) where LCY stands for **Local Currency**. 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.), as well as other fields.

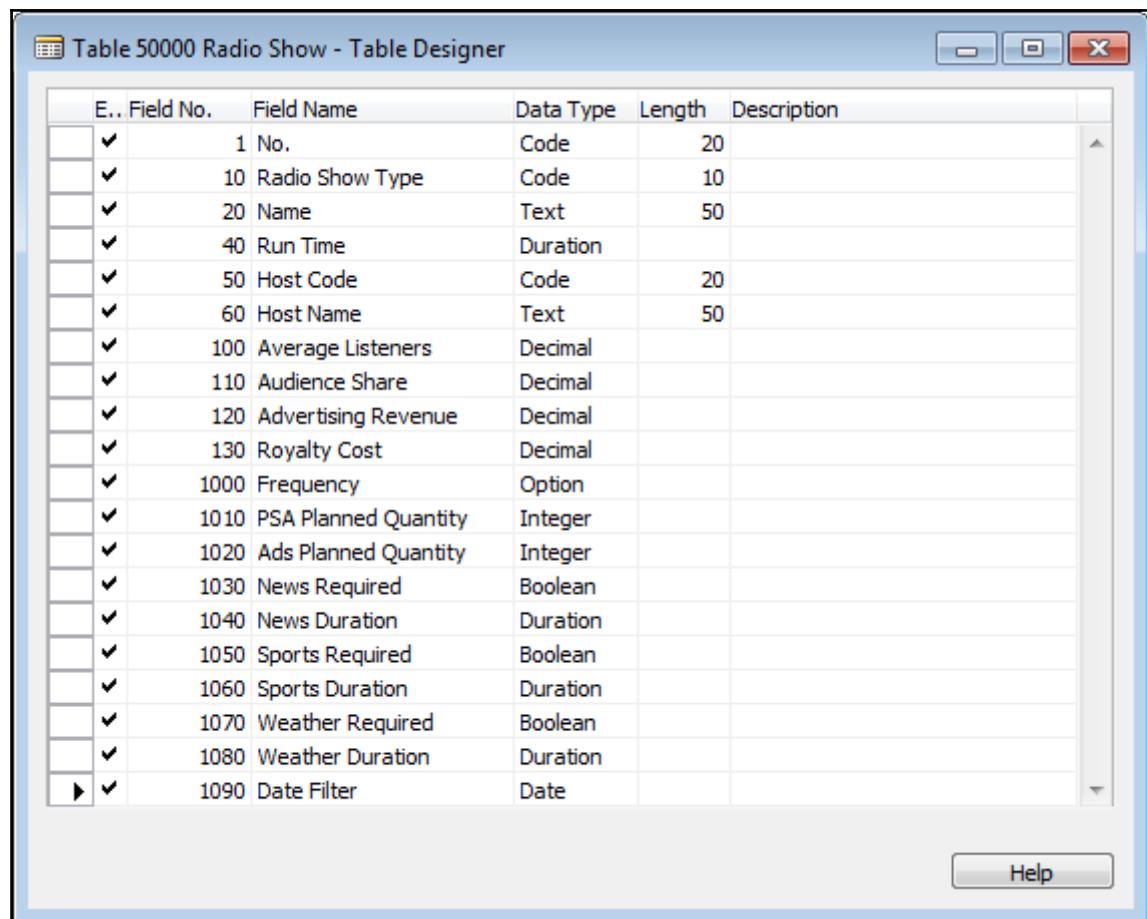
Throughout the standard code, there are FlowFilters in most of the master table definitions. There are the Date Filters and Global Dimension Filters (global dimensions are user-defined codes that facilitate the segregation of accounting data by groupings such as divisions, departments, projects, customer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

The following pair of screenshots show two fields from the Customer table, both with a data type of Date. On the left side of the screenshot is the **Last Date Modified** field (**FieldClass** of `<Normal>`); on the right side of the screenshot is the **Date Filter** field (**FieldClass** of `FlowFilter`). It's easy to see that the properties of the two fields are very similar except for those properties that differ because one is a Normal field and the other is a FlowFilter field:



## FlowFields and a FlowFilter for our application

In our application, we have decided to have several FlowFields and a FlowFilter in **Table 50000 Radio Show**. The reason for these fields is to provide instant analysis for individual shows based on the detailed data stored in subordinate tables. In *Chapter 2, Tables*, we showed Table 50000 with fields 100 through 130 and 1,090 but didn't provide any information on how those fields should be constructed. Let's go through that construction process now. Here's how the fields 100 through 130 and 1,090 should look when we open Table 50000 in the Table Designer. If you didn't add these fields as described in the *Chapter 2, Tables*, section *Creating and Modifying Tables*, do that now:

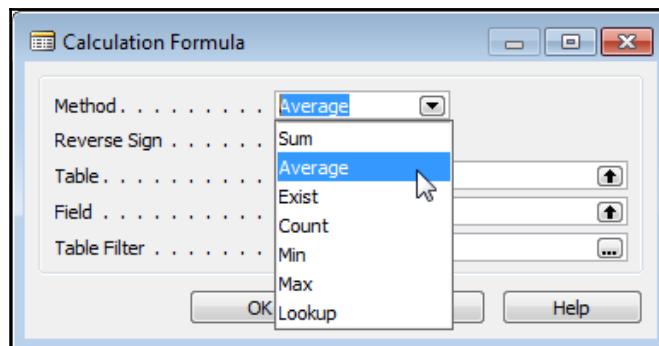


The following five fields will be used for statistical analysis for each Radio Show:

- **Field 100 (Average Listeners):** This is the average number of listeners as reported by the ratings agency
- **Field 110 (Audience Share):** This is the percentage of one station's total estimated listening audience per time slot
- **Field 120 (Advertising Revenue):** This is the sum total of the advertising revenue generated by show
- **Field 130 (Royalty Cost):** This is the sum total of the royalties incurred by the show by playing copyrighted material
- **Field 1090 (Date Filter):** This is a filter to restrict the data calculated for the preceding four fields

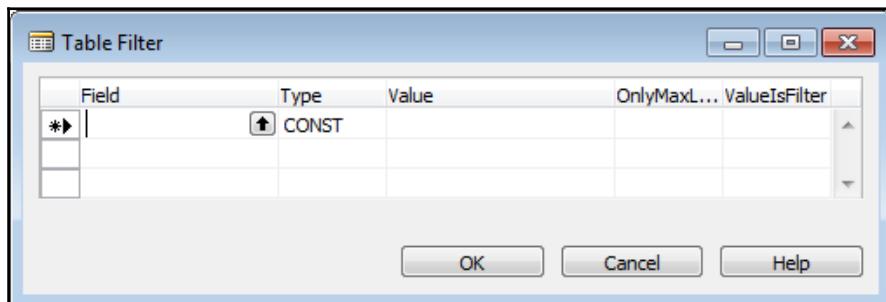
To begin with, we will set the calculation properties for the first FlowField, **Average Listeners**:

1. If Table 50000 isn't already open in the Table Designer, then open it through **Tools | Object Designer**, select the **Table** button on the left as the object type. Find table **50000, Radio Show**, select it and click **Design**.
2. Scroll down to field **100**, select it and click the properties icon at the top of the screen or click **Shift + F4**. Highlight the **FieldClass** property, click on the drop-down arrow and select **FlowField**. A new property will appear called **CalcFormula**, directly underneath the **FieldClass** property. An Assist Edit ellipsis button will appear. Click on it and the **Calculation Formula** form will appear:

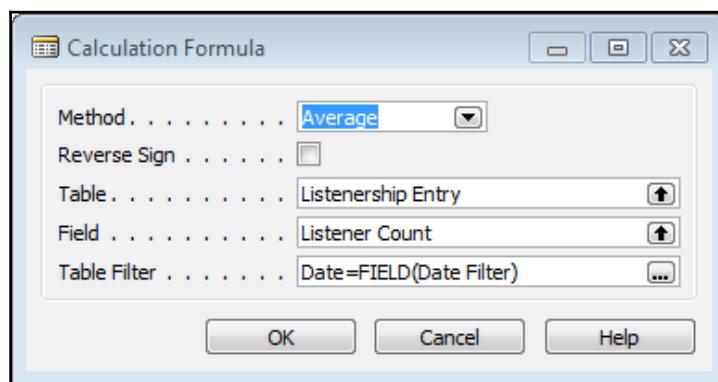


3. Select **Average** from the **Method** dropdown, leave the **Reverse Sign** field unchecked, and type **Listenership Entry or 50006** into the **Table** field. We can either type **Listener Count** or click the lookup arrow button to select the **Listener Count** field from the table.

4. Lastly, we want to define a filter to allow the **Radio Show** statistics to be reviewed based on a user definable date range. Click the Assist Edit ellipsis button on the **Table Filter** field. The following **Table Filter** screen will appear:



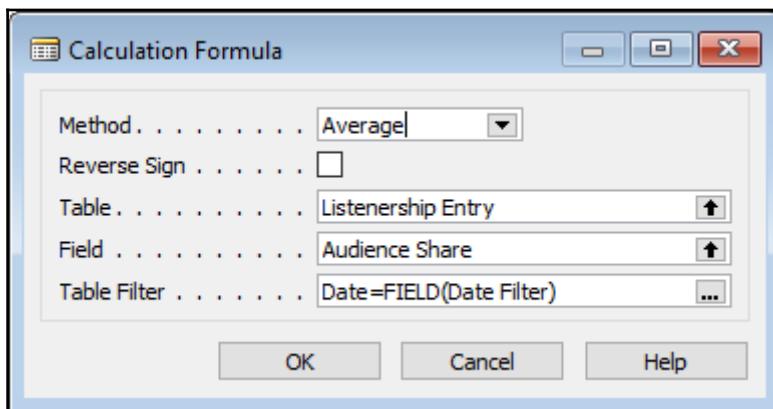
5. Click on the Lookup arrow in the **Field** column and select **Date** from the **Listenership Entry - Field List**.
6. In the **Type** column, click on the drop-down arrow. We see three choices for defining what type of filter to apply: **CONST**, **FILTER**, **FIELD**. In this case, we want to apply a field filter, so choose **FIELD**.
7. The last part of the **Table Filter** definition is the **Value** column. Click on the lookup arrow in the **Value** column and choose **Date Filter** from the **Radio Show - Field List**. This will cause the **Date Filter** field value in the **Radio Show** record to be applied to the values in the **Date** field in **Listenership Entry**, controlling what data to use for the FlowField **Average** calculation.
8. Select **OK** and our **Calculation Formula** screen should look like this:



9. Click on **OK** and the **CalcFormula** property will fill in with the following text:

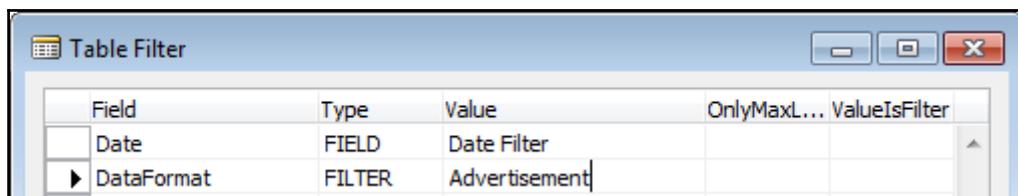
**Average("Listenership Entry"."Listener Count" WHERE (Date=FIELD(Date Filter)))**

10. Since this is a text field, we could enter the syntax manually, but it's much easier and less error prone to use the **Calculation Formula** screen.
11. Set the **Editable** property to **No**.
12. For **Field 110, Audience Share**, repeat the procedure we just went through, but for **Field** select **Audience Share** from the **Listenership Entry** Field List. Our result should look like the following:

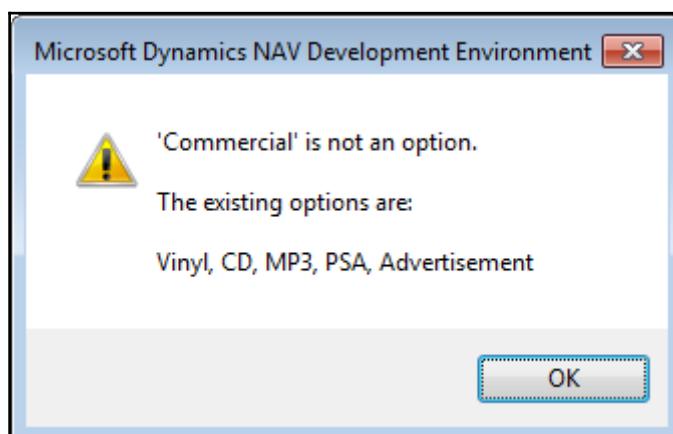


13. For **Field 120, Advertising Revenue** and **Field 130, Royalty Cost**, the FlowField calculation is a sum with multiple fields having filters applied. For each field, the first step will be to set the **FieldClass** property to **FlowField**, then click on the **Assist Edit** button in the **CalcFormula** property to call up the **Calculation Formula** screen.
14. For **Advertising Revenue**, make the **Method** option as **Sum**, for **Table** enter **Radio Show Ledger** or the table number, **50005**, and set **Field** to **Fee Amount**.

15. Click on the Assist Edit button for the table filter. Fill in the first row with the **Field** value as Date, **Type** as FIELD, and **Value** as Date Filter. Fill in the second row with **Field** set to DataFormat, **Type** to FILTER, and Advertisement in the **Value** column (since we are filtering to a single value, we could also have used CONST for the **Type** value). The FlowField will now add up all the Fee Amount values that have a **Format** option selected as Advertisement and which fall within the range of the date filter applied from the Radio Show table:



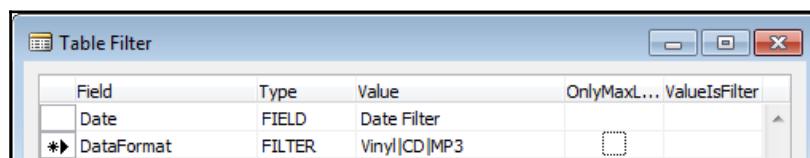
Advertisement is an available value of the field **DataFormat** (Data Type Option). If in the **Radio Show Entry** we had typed a value that was not an Option value such as Commercial, an error message would have displayed showing us what the available Option choices are:



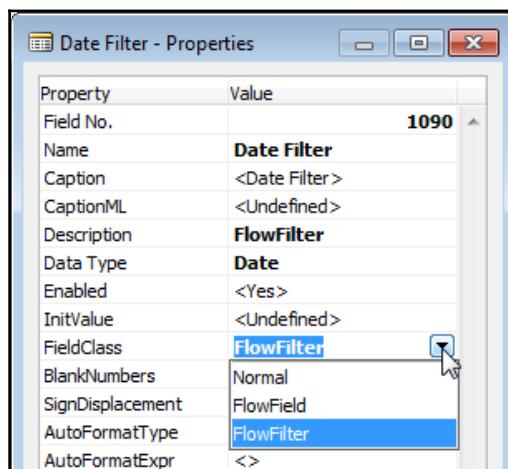
We can use this feature as a development aid when we don't remember what the option values are. We can enter a known incorrect value (such as xxx), press F11 to compile and find out all the correct option values.

16. Click **OK** on the **Table Filter** form, and **OK** again on the **Calculation Formula** form.

17. Start Royalty Cost the same way (**Method** is Sum) all the way through the **Table** (table 50005) and **Field** choices in the **Calculation Formula** form. Click on the **Assist Edit** button for the table filter. Just as before, fill in the first row **Field** with a value of Date, the **Type** with FIELD, and the **Value** with Date Filter.
18. Fill in the second row set **Field** to Format and **Type** to FILTER. In the **Value** column, enter Vinyl | CD | MP3. This means we will filter for all records where the field **Format** contains a value equal to Vinyl OR CD OR MP3 (the Pipe symbol is translated to the Boolean OR). As a result, this FlowField will sum up all the **Fee Amount** values that have a **Format** option selected as Vinyl, CD, or MP3, and a date satisfying the **Date Filter** specified in the Radio Show table:



19. The last field we will define in this exercise is the **Date Filter** field. We have already been referencing this Radio Show table field as a source of a user-defined date selection to help analyze the data from the listenership, payable, and revenue data, but we have not yet defined the field. This one is much easier than the FlowFields because no calculation formula is required.
20. Select the properties for the **Date Filter** field and set the **FieldClass** property to FlowFilter as shown here:



21. Close the **Date Filter - Properties** window and exit **Table Designer**, compiling the `Radio Show` table as we do so. If through this exercise, we had not previously exited and compiled our table modifications, we will get an error message beginning with **The schema synchronization may result in deleted data. The following destructive changes were detected:**. This is followed by a list of all the fields in which we made changes that could affect previously stored data. In this case, that is a list of all the fields which were changed from **Normal** to either **FlowField** or **FlowFilter**. This is because a **Normal** field can store normal data and the other two field types do not. Since we have no data in any of the changed fields, we should choose the **Synchronize Schema** option of **Force** to override the error message and complete the save and compile step. Ideally we should also update the **Version List** field of the table object to indicate we've made additional changes to this table.

## Filtering

Filtering is one of the most powerful tools within NAV. Filtering is the application of defined limits on the data that is to be considered in a process. When we apply a filter to a **Normal** data field, we will only view or process records where the filtered data field satisfies the limits defined by the filter. When we apply a filter to a **FlowField**, the calculated value for that field will only consider data satisfying the limits defined by the filter.

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 information on only a selected group of customers, such as those with an unpaid balance. 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 periods are 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 common to use a combination of the different filtering types. For example, the report just mentioned that lists only customers with an open balance (through a developer-defined filter) could also allow the user to define additional filter criteria. If the user wants to see only Euro currency customers, they would filter on the **Customer Currency Code** field.

Filters are an integral part of the implementation of both FlowFields and FlowFilters. These flexible, 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 competitive systems, standard user inquiries and processes are quite specific. The NAV C/AL toolset allows us to have relatively generic user inquiries and processes, and then allow the user to apply filtering to generate results that fit their specific needs.

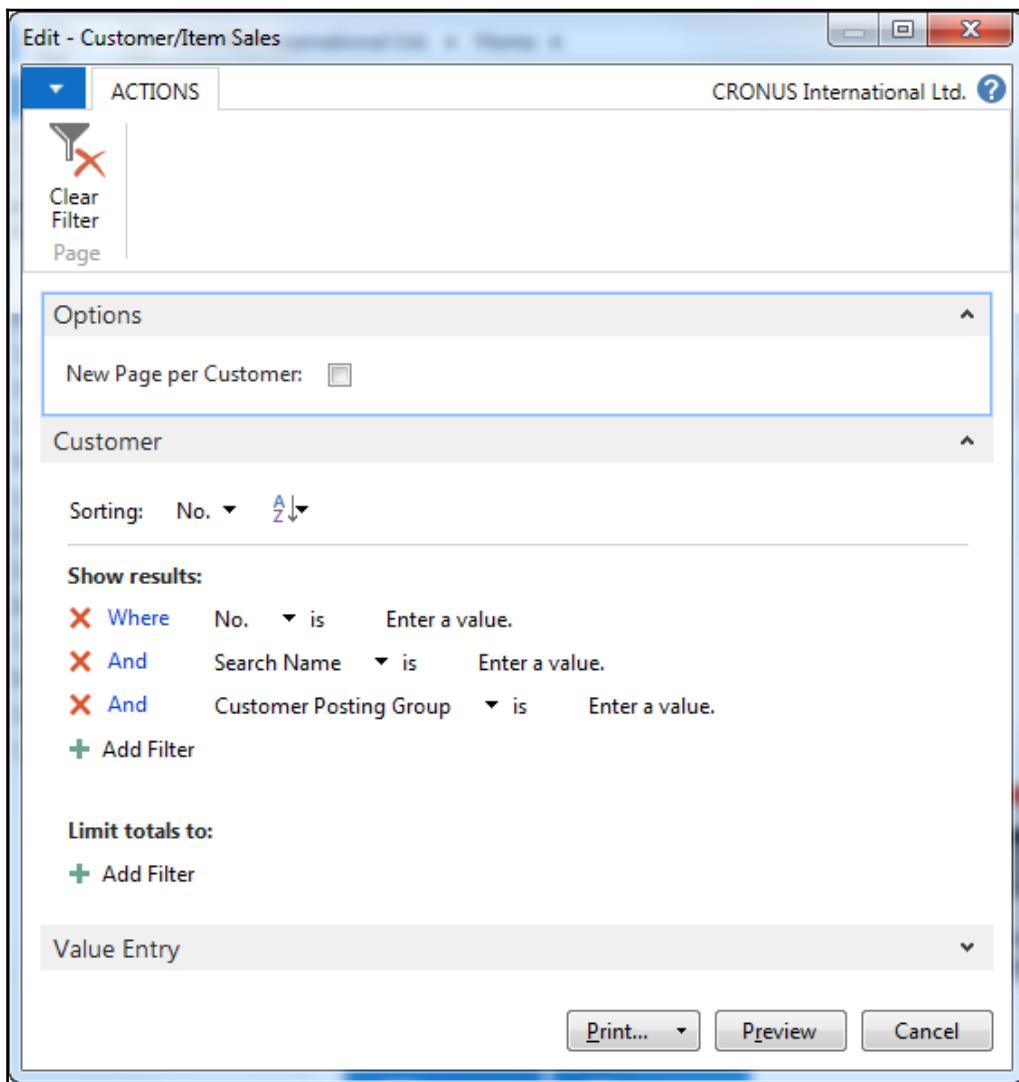
The user sees FlowFilters filtering referred to as **Limit Totals** onscreen. Application of filters and ranges may give varying results depending on Windows settings or the SQL Server collation setup. A good set of examples of filtering options and syntax can be found in *Developer and IT Pro Help* in the section titled *Entering Criteria in Filters*.

## Experimenting with filters

Now it's time for some experimenting with filters. We want to accomplish a couple of things through our experimentation. First, to get more comfortable with how filters are entered; and second, 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. However, the amount of data in the basic Cronus database is small, so any speed differences will be difficult to see in these tests.

We could experiment on any report that allows filtering. A good report for this experimentation is the **Customer/Item** list. This reports which customer purchased which items. The **Customer/Item** List can be accessed on the Role Tailored Client Departments menu via **Sales & Marketing | Sales | Reports | Customer | Customer/Item Sales**.

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



There are also two data fields listed for the entry of filters on the **Value Entry** table as shown in the following screenshot (which has the **Value Entry** FastTab expanded by clicking on it so we can see its predefined filter entry options):



For both the **Customer** and **Value Entry**, these are the fields that the developer of this report determined should be emphasized. If we run the report without entering any filter constraints at all, using the standard Cronus data, the first page of the report will resemble the following:

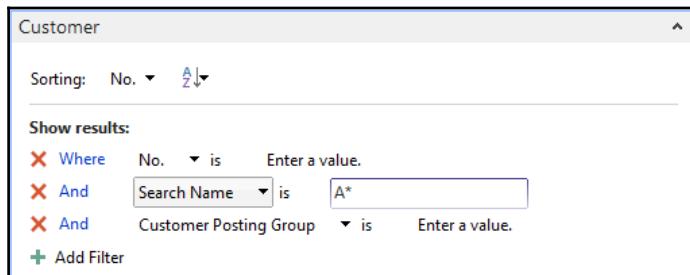
The screenshot shows a 'Customer/Item Sales' report in 'Print Preview' mode. The report header includes:

- Print Preview button
- CRONUS International Ltd. logo
- Date: 2/24/2017 3:03 PM
- Page: 1
- User: CHAMPIDAS

The report title is 'Customer/Item Sales'. It displays a list of sales items with the following columns:

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	498.41	0.00	109.21	21.90
1988-W	CALGARY Whiteboard,	1	PCS	877.32	97.48	168.72	19.20
1972-S	MUNICH Swivel Chair,	1	PCS	123.30	0.00	27.20	22.10
	Progressive Home Furnishings			1,499.03	97.48	305.13	20.40
10000	The Cannon Group PLC Phone No.						
1968-S	MEXICO Swivel Chair, b	3	PCS	351.40	18.50	63.10	18.00
1998-S	ATLANTA Whiteboard, b	7	PCS	6,029.56	317.34	1,079.16	17.90
1964-W	INNSBRUCK Storage U	10	PCS	2,920.00	0.00	1,208.00	41.30
70011	Glass Door	5	PCS	361.50	0.00	177.00	49.00
	The Cannon Group PLC			9,662.46	335.84	2,525.26	26.10
20000	Selangorian Ltd. Phone No.						

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



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:

The screenshot shows a 'Customer/Item Sales' report. The title bar includes 'Print Preview'. The report header says 'Customer/Item Sales', 'Period: CRONUS International Ltd.', '31-10-2016 20:27', 'Page 1', and 'DESKTOP-AHMVITO\MARKB'. It notes 'All amounts are in LCY'. The filter applied is 'Customer: Search Name: A\*'. The data table has columns: Item No., Description, Invoiced Quantity, Unit of Measure, Amount, Discount Amount, Profit, and Profit %. The table lists items for 'Antarcticopy' and 'Autohaus Mielberg KG'.

Item No.	Description	Invoiced Quantity	Unit of Measure	Amount	Discount Amount	Profit	Profit %
3265655	Antarcticopy Phone No.						
1968-S	MEXICO Swivel Chair, b	4 PCS		493,20	0,00	108,80	22,10
1960-S	ROME Guest Chair, gre	7 PCS		875,70	0,00	193,20	22,10
1976-W	INNSBRUCK Storage U	5 PCS		1.152,45	128,05	399,45	34,70
70011	Glass Door	1 PCS		61,45	10,84	24,55	40,00
	Antarcticopy			2.582,80	138,89	726,00	28,10
4963363	Autohaus Mielberg KG Phone No.						
1976-W	INNSBRUCK Storage U	17,57126 PCS		4.050,00	450,00	1.403,77	34,70
1896-S	ATHENS Desk	0 PCS		0,00	0,00	0,00	0,00
1906-S	ATHENS Mobile Pedest	1 PCS		281,40	0,00	61,90	22,00
	Autohaus Mielberg KG			4.331,40	450,00	1.465,67	33,80
Total				6.914,20	588,89	2.191,67	31,70

If we want to expand the customer fields to which we can apply filters, we can access the full list of other fields in the customer table. We can either click on the drop-down symbol next to a filter field that is not already in use or click on the **Add Filter** button to add a new filter field with a drop-down list access. If the number of fields available for filtering is longer than the initial list display allows, the bottom entry in the list is **Additional Columns**. If we click on that, we might end up with a display like the following. Notice that the lists are in alphabetical order, based on the field names. If the list of available fields is too long to display in the second column, that column can be scrolled up and down:

Bill-to No. Of Archived Doc.	No. of Credit Memos
Bill-To No. of Blanket Orders	No. of Invoices
Bill-To No. of Credit Memos	No. of Orders
Bill-To No. of Invoices	No. of Pstd. Credit Memos
Bill-To No. of Orders	No. of Pstd. Invoices
Bill-To No. of Pstd. Cr. Memos	No. of Pstd. Return Receipts
Bill-To No. of Pstd. Invoices	No. of Pstd. Shipments
Bill-To No. of Pstd. Return R.	No. of Quotes
Bill-To No. of Pstd. Shipments	No. of Return Orders
Bill-To No. of Quotes	No. of Ship-to Addresses
Bill-To No. of Return Orders	No. Series
Block Payment Tolerance	Other Amounts
Blocked	Other Amounts (LCY)
Budgeted Amount	Our Account No.
Cash Flow Payment Terms Code	Outstanding Invoices
Chain Name	Outstanding Invoices (LCY)
City	Outstanding Orders
Collection Method	Outstanding Orders (LCY)
Combine Shipments	Outstanding Serv. Orders (LCY)
Comment	Outstanding Serv.Invoices(LCY)
Contact	Partner Type
Contract Gain/Loss Amount	Payment Method Code
Copy Sell-to Addr. to Qte From	Payment Terms Code
Country/Region Code	Payments
County	Payments (LCY)
Cr. Memo Amounts	Phone No.
Cr. Memo Amounts (LCY)	Picture
Credit Amount	Place of Export
Credit Amount (LCY)	Pmt. Disc. Tolerance (LCY)
Credit Limit (LCY)	Pmt. Discounts (LCY)
Currency Code	Pmt. Tolerance (LCY)
Customer Disc. Group	Post Code
Customer Posting Group	Preferred Bank Account Code
Customer Price Group	Prepayment %
Debit Amount	Prices Including VAT
Debit Amount (LCY)	Primary Contact No.
Additional	

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. If we clicked on the lookup arrow in the **Filter** column, a screen would pop up, allowing us to choose from data items in the related table, in this case, **Territories**:

The screenshot shows a SAP Fiori Request Page titled "Customer". The top navigation bar has a back arrow and a search icon. Below the title, there's a sorting dropdown set to "No." with "A-Z" and "Z-A" options. The main area is divided into sections: "Show results:", "Limit totals to:", and "Value Entry". In the "Show results:" section, there are four filter conditions: "Where No. is Enter a value.", "And Search Name is Enter a value.", "And Customer Posting Group is Enter a value.", and "And Territory Code is Enter a value.". The fourth condition is currently selected, and a dropdown menu is open, showing a list of territory codes and their names. The list includes EANG (East Anglia), FOREIGN (Foreign), LND (London), MID (Midlands), N (North), and NE (North East). The "EANG" entry is highlighted. The "Value Entry" section contains a "Show results:" button.

Co...	Name
EANG	East Anglia
FOREIGN	Foreign
LND	London
MID	Midlands
N	North
NE	North East

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 numbers that contain the letter **W**, the report will be similar to the following screenshot:

The screenshot shows a SAP ERP report titled "Customer/Item Sales". The report header includes "Print Preview", "Customer/Item Sales", "Period: CRONUS International Ltd.", "All amounts are in LCY", "Customer: Search Name: A\*", and "Value Entry: Item No.: \*W\*". The report date is 31-10-2016 20:37, page 1, and the system is DESKTOP-AHMVITO\MARKB.

Item No.	Description	Invoiced Quantity	Unit of Measure	Amount	Discount Amount	Profit	Profit %
32656565	Antarcticopy Phone No.						
1976-W	INNSBRUCK Storage U Antarcticopy	5 PCS		1.152,45 1.152,45	128,05 128,05	399,45 399,45	34,70 34,70
49633663	Autohaus Mielberg KG Phone No.						
1976-W	INNSBRUCK Storage U Autohaus Mielberg KG	17,57126 PCS		4.050,00 4.050,00	450,00 450,00	1.403,77 1.403,77	34,70 34,70
Total				5.202,45	578,05	1.803,22	34,70

If we want to see all the items containing either the letter W or the letter S, our filter would be \*W\* | \*S\*. If we made the filter W | S, then we would get only entries that are exactly equal to W or to S because we didn't include any wildcards.

You should go back over the various types of filters we discussed, try each one and try them in combination. Get creative! Try some things that you're not sure will work and see what happens. Explore a variety of reports or list pages in the system by applying filters to see the results of your experiments. A good page on which to apply filters is **Customer List (Sales & Marketing | Sales | Customers)**. This filtering experimentation process is safe (you can't hurt anything or anyone) and a great learning experience.

## Accessing filter controls

NAV 2017 has two very different approaches to setting up filtering; one for the Development Environment and the other for the Role Tailored Client. Because we develop in the former, we will briefly cover filtering there. Because we target our development for use in the Role Tailored Client, we need to be totally comfortable with filtering there, as that is the interface for our users.

## Development Environment filter access

There are four buttons at the top of the screen that relate to filtering, plus one for choosing the active key (that is - current sort sequence). Depending on the system configuration (OS and setup), they will look similar to those in the following screenshot:



From left to right, they are:

- **Field Filter (F7):** Highlight a field, press *F7* (or select **View | Field Filter**), and the data in that field will display ready for us to define a filter on that data field. We can freely edit the filter before clicking on **OK**.
- **Table Filter (Ctrl + F7):** Press the *Ctrl* key and *F7* simultaneously (or select **View | Table Filter**). We will get a screen that allows us to choose fields in the left column and enter related filters in the right column. Each filter is the same as would have been created using the **Field Filter** option. The multiple filters for the individual fields are ANDed together (that is, they all apply simultaneously). If we invoke the **Table Filter** form when any **Field Filter** option is already applied, they will be displayed in the form.
- **Flow Filter (Shift + F7):** Since we cannot view any data containing FlowFields in the Development Environment, using the **Flow Filter** option in the Development Environment is not useful.
- **Show All (Shift + Ctrl + F7):** This will remove all Field filters, but will not remove any Flow filters.
- **Sort (Shift + F8):** This allows us to choose which key is active on a displayed data list.

When we are viewing a set of data (such as a list of objects) and want to check if any filters are in effect, we should check the bottom of the screen for the word **FILTER**.

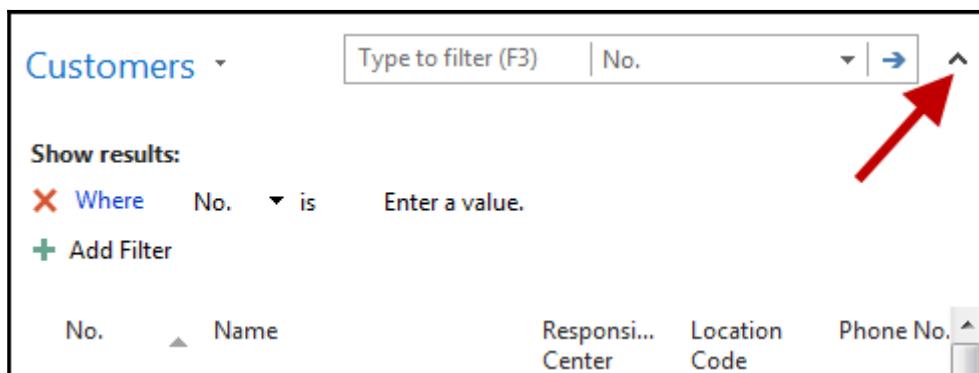
## Role Tailored Client filter access

The method of accessing fields to use in filtering in the **Role Tailored Client (RTC)** is quite different from that in the Development Environment.

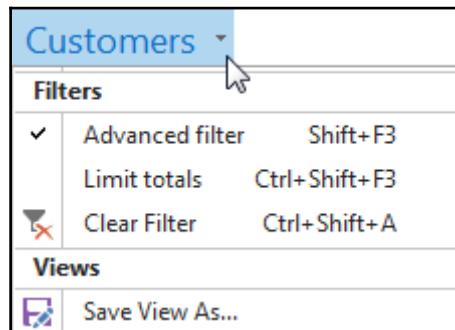
When a page such as the Customer List is opened, the filter section at the top of the page looks like the following image. On the upper-right corner is a place to enter single-field filters. This is the **Type to filter (F3)** (also referred to as a Quick filter), essentially equivalent to the Field Filter in the Development Environment. The fields available for filtering are the same as the visible columns showing in the list:



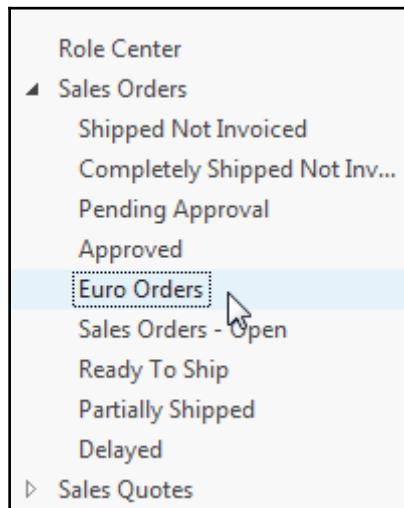
If we click on the chevron circle button in the upper-right corner to expand the **Filter pane**, the result will look similar to the following. This filter display includes an additional filtering capability, **Show results**, allowing entry of filters of the Limit totals to type:



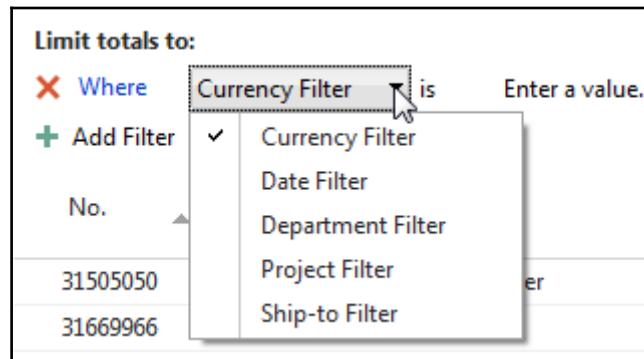
If we go to the **Filter** pane header line where the **Menu** caption of the page is located (**Customers** in this page) and click on the drop-down symbol, we will see a set of selection options (the filter menu) like that in the following screenshot. The **Advanced filter** option provides for the entry of multiple **Field Filter** (essentially the same as the Development Environment Table Filter). The **Limit totals** filter provides for the entry of FlowFilter constraints:



This is one of two places we can clear filters of all types (we can also enter *Ctrl + Shift + A* as indicated in the filter menu). The **Save View As...** option allows the user to save the filtered view, name it and add it to an Activity Group in the **Navigation** pane. The following screenshot shows a series of **Saved Views on Sales Orders** (most of them available out-of-the-box). The **Euro Orders** entry is a **Saved View** created by a user:



If we click on **Limit totals** (or click *Ctrl + Shift + F3*), the **Limit totals to:** portion of the filter pane will display. When we click on the drop-down arrow, we will get a list of all the FlowFields to which we can apply one or more **Limit totals** (FlowFilters):



Depending on the specific page and functional area, Flowfield filtering can be used to segregate data on the dimension fields. For example, in the page shown in the preceding screenshot, we could filter for data regarding a single department or project (both of which are dimension fields), or range of departments, or projects or a range of customer ship-to locations.

## Review questions

1. The maximum length for a C/AL field name or variable name is 250 characters long. True or false?
2. 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

3. How many of the following field data types support the storing of application data such as names and amounts - 1, 2, 3 or 4?
  - a) FlowFilter
  - b) Editable
  - c) Normal
  - d) FlowField
4. The `ExtendedDataType` property supports the designation of all but one of the following data types, displaying an appropriate action icon. Choose the one not supported:
  - a) E-mail address
  - b) Website URL
  - c) GPS location
  - d) Telephone number
  - e) Masked entry
5. One of the following is not a FlowField Method. choose one:
  - a) Median
  - b) Count
  - c) Max
  - d) Exist
  - e) Average
6. It is important to have a consistent, well-planned approach to field numbers, especially if the application will use the `TransferFields` function. True or false?
7. Field filters and limit totals cannot both be used at the same time. True or false?

8. Which property is used to support the multi-language feature of NAV ? Choose one:
  - a) Name
  - b) CaptionML
  - c) Caption
  - d) LanguageRef
9. Which of the following are field triggers? Choose two:
  - a) OnEntry
  - b) OnValidate
  - c) OnDeletion
  - d) OnLookup
10. Which of the following are complex data types? Choose three:
  - a) Records
  - b) Strings of text
  - c) DateFormula
  - d) DateTime data
  - e) Objects
11. 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?
12. Text and code variables can be any length:
  - a) In a memory variable (working storage)? True or false?
  - b) In a table field? True or false?
13. FlowField results are not stored in the NAV table data. True or false?

14. The following two filters are equivalent. True or false?
  - a) (\*W50?|I?5|D\*)
  - b) (I?5) OR (D\*) OR (\*W50?)
15. Limit totals apply to FlowFilters. True or false?
16. All data types can be used to define data both in tables and in working storage. True or false?
17. DateFormula alpha time units include which of the following? Choose two:
  - a) C for century
  - b) W for week
  - c) H for holiday
  - d) CM for current month
18. FlowFilter data is stored in the database. True or false?
19. Option data is stored as alpha data strings. True or false?
20. Which of the following are numeric data types in NAV 2017? Choose two:
  - a) Decimal
  - b) Option
  - c) Hexadecimal
  - d) BLOB
21. Which of the following act as wildcards in NAV 2017? choose two:
  - a) Decimal point ( . )
  - b) Question mark ( ? )
  - c) Asterisk ( \* )
  - d) Hash mark ( # )

## 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. We walked through a number of examples to illustrate most of these elements, though we have postponed exploring triggers until later, when we have more knowledge of C/AL. We covered data type and FieldClass, properties which determine what kind of data can be stored in a field.

We reviewed and experimented with the date calculation tool that gives C/AL an edge in business applications. We discussed filtering, how filtering is considered as we design our database structure, and how the users will access data. Finally, more of our NAV Radio Show application was constructed.

In the next chapter, we will look at the many different types of pages in more detail. We'll put some of that knowledge to use to further expand our example NAV application.

# 4

## Pages - the Interactive Interface

*"Design is not just what it looks like and feels like. Design is how it works."*

- Steve Jobs

*"True interactivity is not about clicking on icons or downloading files, it's about encouraging communication."*

- Ed Scholssberg

Pages are NAV's object type for interactively presenting information. The page rendering routines that paint the page on the target display handle much of the data presentation detail. This allows a variety of clients to be created by Microsoft, such as a Web browser resident client, the Windows RTC client, and a universal client that can be used on all devices, such as phones, tablets, and other platforms (iPad, Android, Windows).

**Independent Software Vendors (ISVs)** have created mobile clients and even clients targeted to devices other than video displays.

One of the benefits of Page technology is the focus on the user experience rather than the underlying data structure. As always, the designer/developer has the responsibility of using the tools to their best effect. Another advantage of NAV 2017 pages is the flexibility they provide the user for personalization, allowing them to tailor what is displayed and how it is organized.

In this chapter, we will explore the various types of pages offered by NAV 2017. We will review many options for format, data access, and tailoring pages. We will also learn about the Page Designer tools and the inner structures of pages. We will cover the following topics:

- Page Design and Structure Overview
- Types of Pages
- Page Designer
- Page Components
- Page Controls
- Page Actions
- WDTU Page Enhancement exercises

## **Page Design and Structure Overview**

Pages serve the purpose of input, output, and control. They are views of data or process information designed for onscreen display only. They are also user data entry vehicles.

Pages are made up of various combinations of Controls, Properties, Actions, Triggers, and C/AL Code, which are briefly explained here:

- **Controls:** These provide the users with ways to view, enter, and edit data, choose options or commands, initiate actions, and view status
- **Properties:** These are attributes or characteristics of an object that define its state, appearance, or value
- **Actions:** These are menu items that may be icons
- **Triggers:** These are predefined functions that are executed when certain actions or events occur

The internal structure of a page maps to a tree structure, some of which is readily visible in the Page Designer display while the rest is in the background.

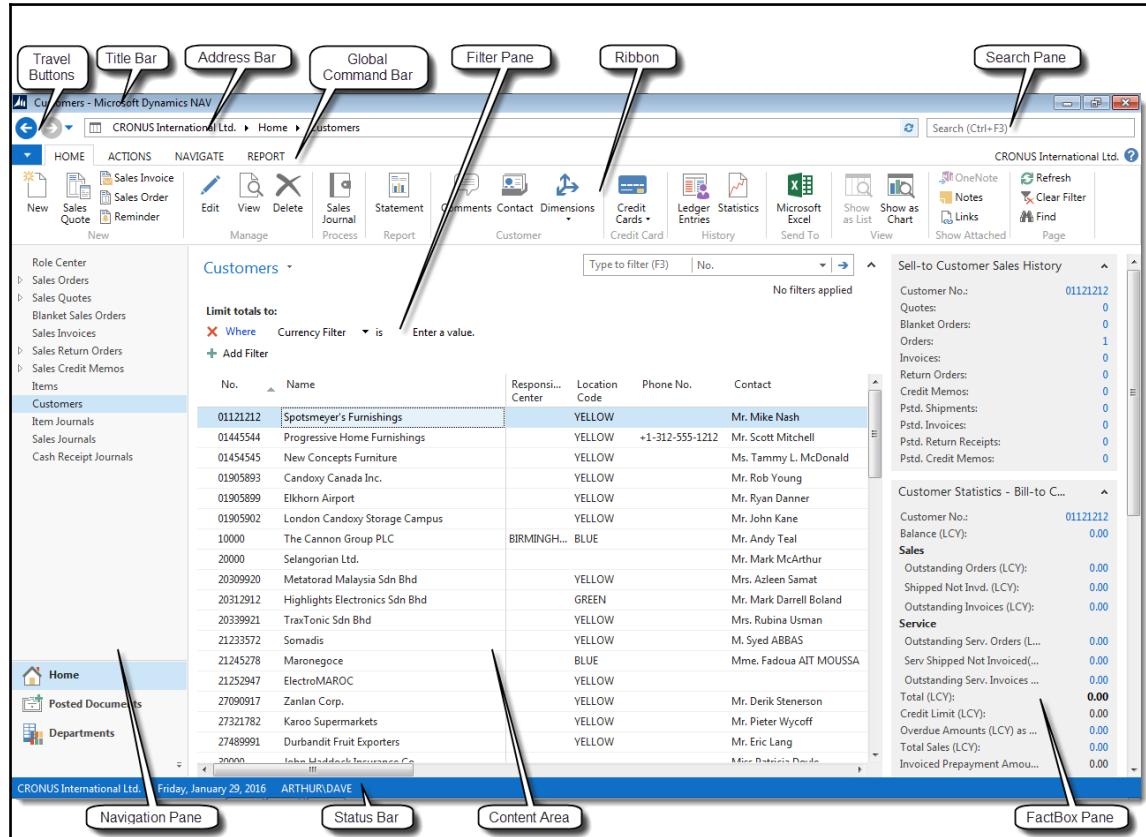
## Page Design guidelines

Good design practice dictates that enhancements integrate seamlessly with the existing software, unless there is an overwhelming justification for being different. When we add new pages or change existing pages, the changes should have the same look and feel as the original pages, unless the new functionality requires significant differences. This consistency not only makes the user's life easier, it makes support, maintenance, and training more efficient.

There will be instances where we will need to create a significantly different page layout in order to address a special requirement. Maybe we will need to use industry-specific symbols, or we will need to create a screen layout for a special display device. Perhaps, we will create a special dashboard display to report the status of work queues. Even when we will be different, we should continue to be guided by the environment and context in which our new work will operate.

# NAV 2017 Page structure

Let's take a look at what makes up a typical page in the NAV 2017 Windows Client. The page in the following screenshot includes a **List** page at its core (the Content Area):



Here is a brief description of all these options:

- The **Travel Buttons** serve the same purpose they serve in the Explorers--to move backward or forward through previously displayed pages.
- The **Title Bar** displays **Page Caption** and product identification.

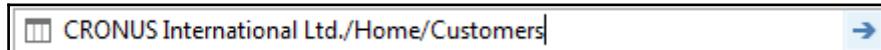
- The **Address Bar** (also referred to as the Address Box) displays the navigation path that led to the current display. It defaults to the following format, which is sometimes referred to as the **breadcrumb path**:



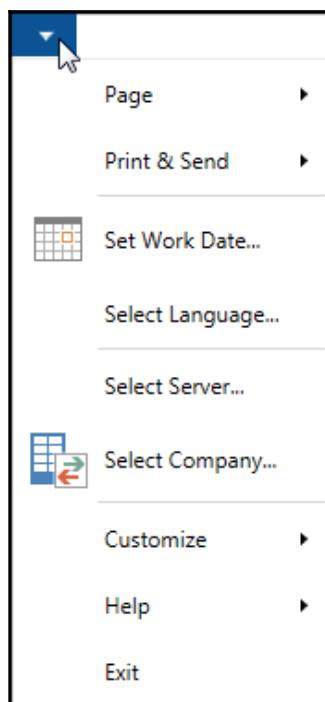
If we click on one of the right-facing arrowheads in the **Address Bar**, child menu options will be displayed in a drop-down list, as shown in the following screenshot. The same list of options subordinate to **Sales Orders** is displayed both in the drop-down menu from the **Address Bar**, and in the detailed list of options in the navigation pane:

A screenshot of the Microsoft Dynamics NAV Sales Orders screen. The top navigation bar shows the address path: CRONUS International Ltd. > Home > Sales Orders. The main area displays the Sales Orders list with columns for No., Sell-to Customer Name, External Document..., and Location Code. On the left, the navigation pane shows a tree view of sales-related entities: Role Center, Sales Orders (selected), Sales Quotes, Blanket Sales Orders, and Sales Invoices. A dropdown menu is open over the address bar, listing various sales order status filters: Shipped Not Invoiced, Completely Shipped Not Invoiced, Pending Approval, Approved, Euro Orders, Sales Orders - Open, Ready To Ship, Partially Shipped, and Delayed. Below the dropdown, there are filter and limit total controls.

If we click in the blank space in the **Address Bar** to the right of the breadcrumbs, the path display will change to a traditional path format, as shown in the following screenshot:



- The **Global Command Bar** provides access to a general set of menu options, which varies slightly based on what is in the content area. The leftmost menu in the **Command Bar** is accessed by clicking on the drop-down arrow at the left end of the **Global Command Bar**. It provides access to some basic application information and administration functions, as shown in the following screenshot:



- The **Filter Pane** is where the user controls the filtering to be applied to the page display.
- The **Ribbon** contains shortcut icons to actions. These same commands will be duplicated in other menu locations, but are in the **Ribbon** for quick and easy access. The **Ribbon** can be collapsed (made not visible) or expanded (made visible) under user control.
- The **Search Field** allows users to find pages, reports, or views based on the object's name (full or partial). Search finds only objects that are accessible from the Navigation Pane.
- The Navigation Pane contains menu options based on the active **Role Center**, which is tied to the users login. It also contains activity buttons at a minimum: the **Home** and **Departments** buttons. The **Departments** button and its menu items are generated based on the contents of the **NAV MenuSuite**.
- The **Status Bar** shows the name of the active Company, the work date, and the current User ID. The **Status Bar** shows the name of the active company, the work date, and the current User ID. If we double click on the company name, we can then change companies. If we double-click on the Work Date, we can then change the Work Date.
- The **Content Area** is the focus of the page. It may be a Role Center, a List page, or a Departments menu list.
- The **FactBox Pane** can appear on the right side of certain page types, such as Card, List, ListPlus, Document, Navigate, or Worksheet. A FactBox can only display a **Card Part**, **List Part**, **System Part**, or a limited set of predefined charts. FactBoxes can provide no-click and one-click access to related information about the data in focus in the Content Area.

## Types of pages

Let's review the types of pages available for use in an application. Then, we will create several examples for our WDTU Radio Station system.

Each time we work on an application design, we will carefully need to consider which page type is best to use for the functionality we are creating. Types of pages available include Role Center, List, Card, List Part, Card Part, List Plus, Document, Worksheet, Navigate Page, Confirmation Dialog, and Standard Dialog. Pages can be created and modified by the developer and can be personalized by the administrator, super user, or user.

## Role Center page

Users are each assigned a Role Center page as their home page in NAV--the page where they land when first logging into NAV 2017. The purpose of a Role Center page is to provide a task-oriented home base that focuses on the tasks that the user typically needs in order to do his/her job on a day-to-day basis. Common tasks for any user should be no more than one or two clicks away.

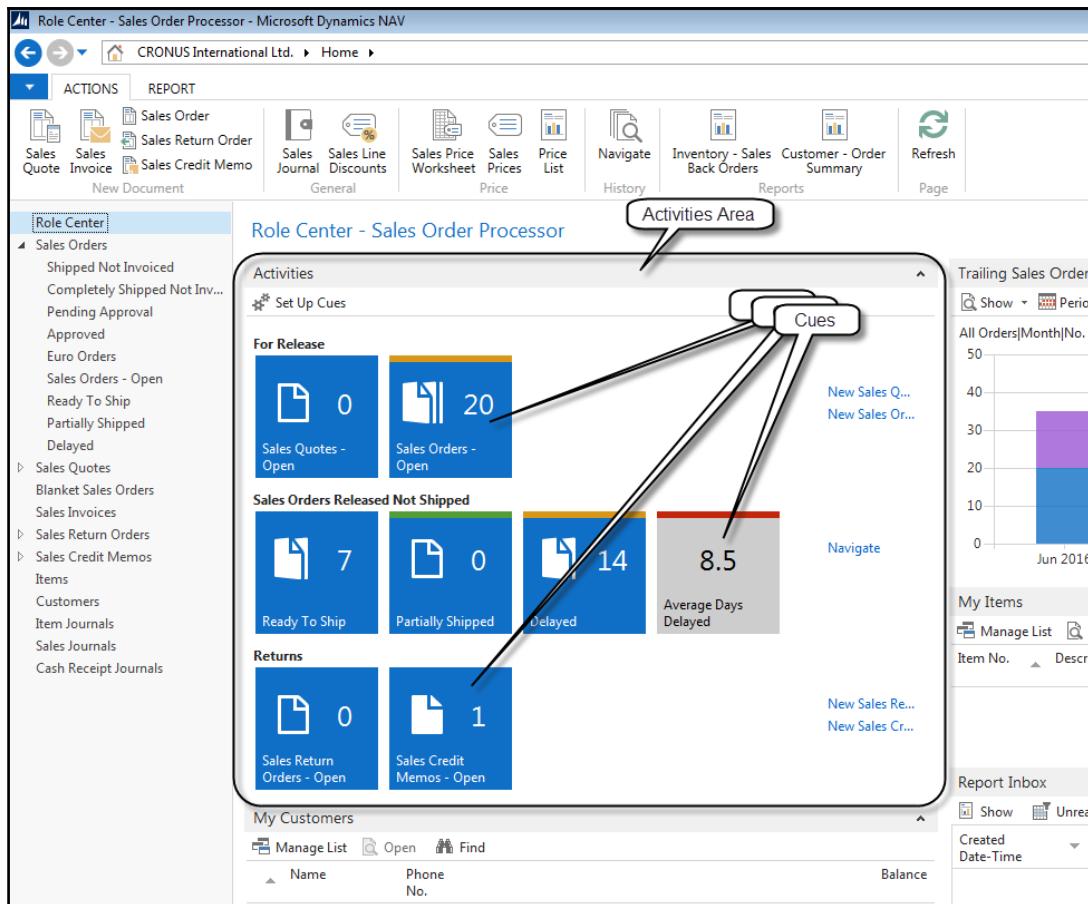
The standard NAV 2017 distribution includes predefined **Role Center** pages, including generic roles such as the Bookkeeper, Sales Manager, and Production Planner. Some of the provided Role Centers are richly featured and have been heavily tailored by Microsoft as illustrations of what is possible. On the other hand, some of the provided Role Centers are only skeletons, acting essentially as place holders.



It is critical to understand that the provided Role Center pages are intended to be templates, not final deliverables.

Central to each Role Center Page is the **Activities Area**. The **Activities Area** provides the user with a visual overview of their primary tasks. Within the **Activities Area** are the **Cues**. Each blue Cue icon (containing a document symbol) represents a filtered list of documents in a particular status, indicating an amount of work to be handled by the user. The grey Cue icons (with no document symbol) display a calculated value.

The following screenshot shows a **Role Center** page for the user role profile of **Sales Order Processor**:



## List page

List pages are the first pages accessed when choosing any menu option to access data. This includes all the entries under the **Home** button on the Navigation Pane. A List page displays a list of records (rows), one line per record, with each displayed data field in a column.

When a List page is initially selected, it is not editable. When we double-click an entry in a List, either an editable Card page or an editable List page entry is displayed. Examples of this latter behavior are the Reference table pages, such as **Post Codes**, **Territories**, and **Languages**. A List page can also be used to show a list of master records to allow the user to visually scan through the list of records, or to easily choose a record on which to focus.

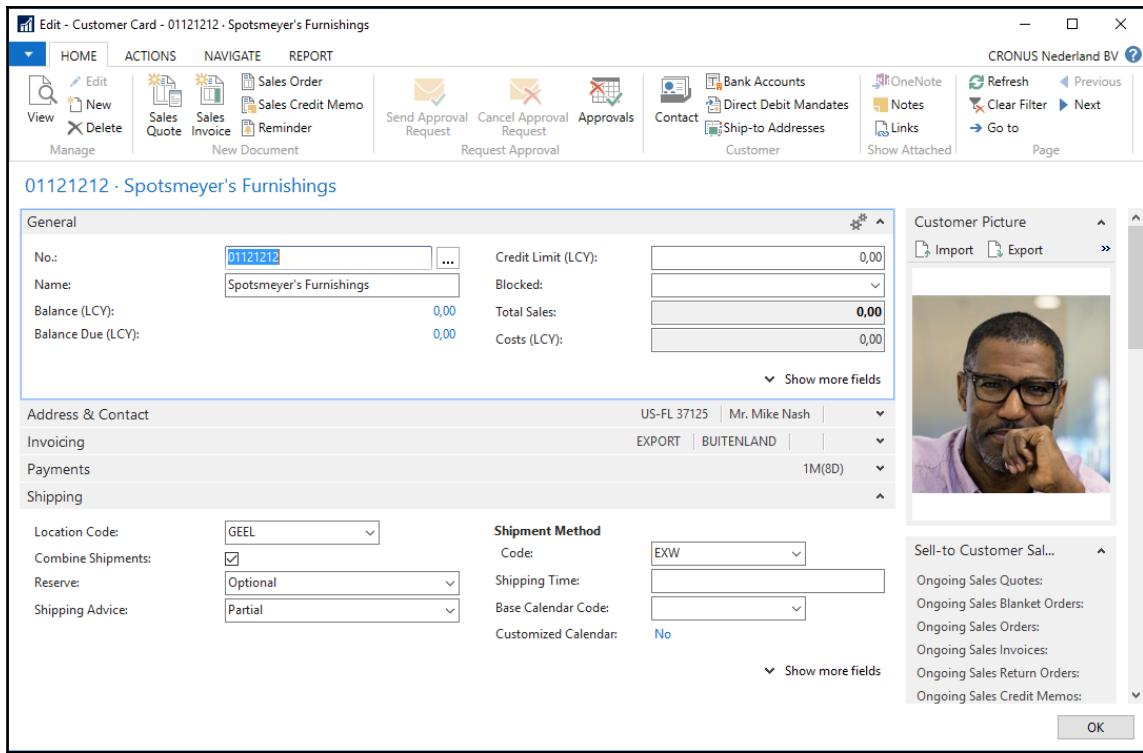
List pages may optionally include **FactBoxes**. Some NAV 2017 List pages, such as **Customer Ledger Entries** (Page 25), allow editing of some fields (for example, **Invoice Due Dates**) and not of other fields.

The following screenshot shows a typical list page--the **Items** List, Page 31:

No.	Description	Type	Inventory	Su... Exist	As... BO...	Production BOM No.	Routing No.	Base Unit of Measure
1000	Fiets	Inventory	32	No	No	1000	1000	STUKS
1001	Tourfiets	Inventory	0	No	No	1000	1000	STUKS
1100	Voorwiel	Inventory	152	No	No	1100	1100	STUKS
1110	Velg	Inventory	400	No	No			STUKS
1120	Spaken	Inventory	10.000	No	No			STUKS
1150	Voornaaf	Inventory	200	No	No	1150	1150	STUKS
1151	Draagas voorwiel	Inventory	200	No	No			STUKS
1155	Socket (voor)	Inventory	200	No	No			STUKS
1160			200	..	..			STUKS

## Card page

Card pages display and allow updating of a single record at a time. Card pages are used for master tables and setup data. Complex cards can contain multiple **FastTabs** and **FactBoxes** as well as display data from subordinate tables. For example, a Card page image for the **Customer Card**, Page 21, follows with the **General** and **Shipping** FastTab expanded and the other FastTabs (**Address & Contact**, **Invoicing**, **Payments**) collapsed:



## Document page

**Document** (task) pages have at least two FastTabs in a header/detail format. The FastTab at the top contains the header fields in a card style format, followed by a FastTab containing multiple records in a list style format (a ListPart page). Some examples are **Sales Orders**, **Sales Invoices**, **Purchase Orders**, **Purchase Invoices**, and **Production Orders**. The Document page type is appropriate whenever we have a parent record tied to subordinate child records in a one-to-many relationship. A Document page may also have FactBoxes. An example, the **Sales Order Document** page - Page 42, follows with two FastTabs expanded and four FastTabs collapsed:

The screenshot shows the SAP Fiori Sales Order interface for document number 101019. The top navigation bar includes 'Edit - Sales Order - 101019 · Candoxy Nederland BV', 'HOME', 'ACTIONS', 'NAVIGATE', and 'CRONUS Nederland BV'. Below the header are several action buttons: View, Release, Reopen, Copy Document..., Order Promising, Statistics, Assembly Orders, Invoices, Archive Document, Shipments, Email Confirmation..., Print Confirmation..., Order Confirmation, Posting, Request Approval, Show Attached, and Page.

The main content area is divided into sections:

- General:** Contains fields for Customer (Candoxy Nederland BV), Contact (Rob Verhoff), Due Date (31-1-2018), Requested Delivery Date, Posting Date (31-1-2018), External Document No., Order Date (22-1-2018), and Status (Released). A 'Show more fields' button is present.
- Lines:** A table showing sales lines with columns: Type, No., Description, Location Code, Quantity, Qty. to Assemble to Order, and Reserved Quantity. The table contains three items:
 

Item	1952-W	OSLO Boekenkast	ROOD	2		
Item	1928-W	ST.MORITZ Ladekast	ROOD	2		
Item	1976-W	INNSBRUCK Kast/w.deur	ROOD	2		

 Subtotal Excl. VAT (EUR): 5.181,16, Inv. Discount Amount Excl. VAT (EUR): 0,00, Invoice Discount %: 0, Total Excl. VAT (EUR): 5.181,16, Total VAT (EUR): 984,42, Total Incl. VAT (EUR): 6.165,58.
- Sell-to Customer Sales:** A list of related documents: Ongoing Sales Quotes, Ongoing Sales Blanket Orders, Ongoing Sales Orders, Ongoing Sales Invoices, Ongoing Sales Return Orders, Ongoing Sales Credit Memos, Posted Sales Shipments, Posted Sales Invoices, Posted Sales Return Receipts, and Posted Sales Credit Memos.
- Customer Details:** Fields include Customer No.: 31987987, Phone No., Email, Fax No., Credit Limit (LCV): 0,00, Available Credit: 0,00, Payment Terms C... LM, and Contact: Rob Verh...
- Sales Line Details:** Fields include Item No.: 1952-W, Required Quantity: 1, and Availability.

An 'OK' button is located at the bottom right of the dialog.

## FastTabs

FastTabs, as shown in the preceding **Customer Card** and **Sales Order** screenshots, are collapsible/expandable replacements for traditional left-to-right forms tabs. FastTabs are often used to segregate data by subject area on a Card page or a Document page. In the preceding screenshot, the **General** and **Lines** FastTabs are expanded and the remaining FastTabs are collapsed. Individually, important fields can be promoted so that they display on the FastTab header when the tab is collapsed, allowing the user to see this data with minimal effort. Examples appear on the Sales Order's collapsed **Invoice Details** and **Prepayment** FastTabs. Promoted field displays disappear from the FastTab header when the FastTab is expanded.

## ListPlus page

A ListPlus page is similar in layout to a Document page as it will have at least one FastTab with fields in a card type format and one FastTab with a list page format. Unlike a Document page, a ListPlus page may have more than one FastTab with card format fields and one or more FastTabs with a list page format, while a Document page can only have a single list style subpage. The card format portion of a ListPlus page often contains control information determining what data is displayed in the associated list, such as in Page 113, **Budget**, which is shown in the following screenshot:

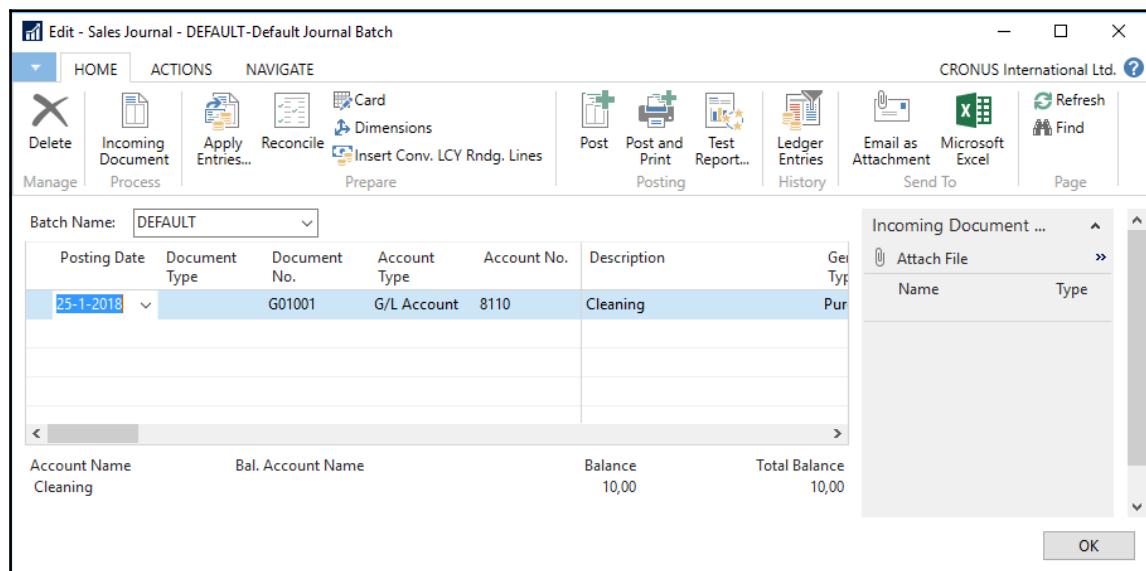
Code	Name	Budgeted Amount	25-01-18	26-01-18	27-01-18
1220	Increases during the Year				
1230	Decreases during the Year				
1240	Accum. Depr., Oper. Equip.				
<b>1290</b>	<b>Operating Equipment, Total</b>				
<b>1300</b>	<b>Vehicles</b>				
1310	Vehicles				
1320	Increases during the Year				

A ListPlus page may also have FactBoxes. Other examples of ListPlus pages are Page 155, **Customer Sales** and Page 157, **Item Availability by Periods**.

## Worksheet (Journal) page

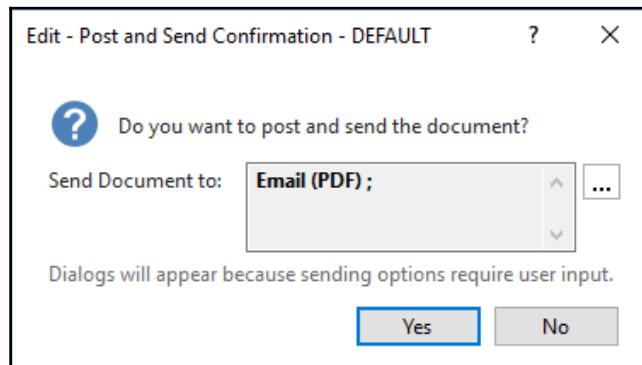
The **Worksheet** pages are widely used in NAV to enter transactions. The Worksheet page format consists of a list page style section showing multiple record lines in the content area, followed by a section containing either additional detail fields for the line in focus or containing totals. All the Journals in NAV use Worksheet pages. Data is usually entered into a Journal/Worksheet by keyboard entry, but in some cases, via a batch process.

The following screenshot shows a Worksheet page, **Sales Journal - Page 253**:



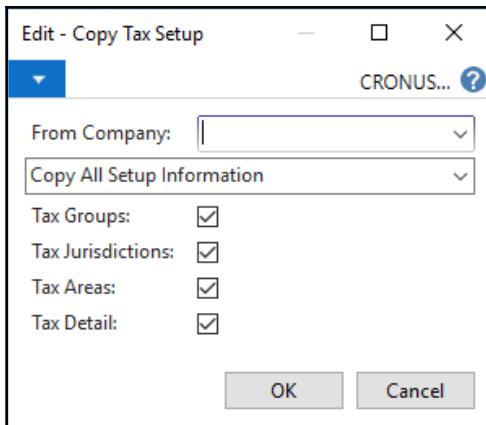
## Confirmation Dialog page

This is a simple display page embedded in a process. It is used to allow a user to control the flow of a process. A sample **Confirmation Dialog** page is shown in the following screenshot:



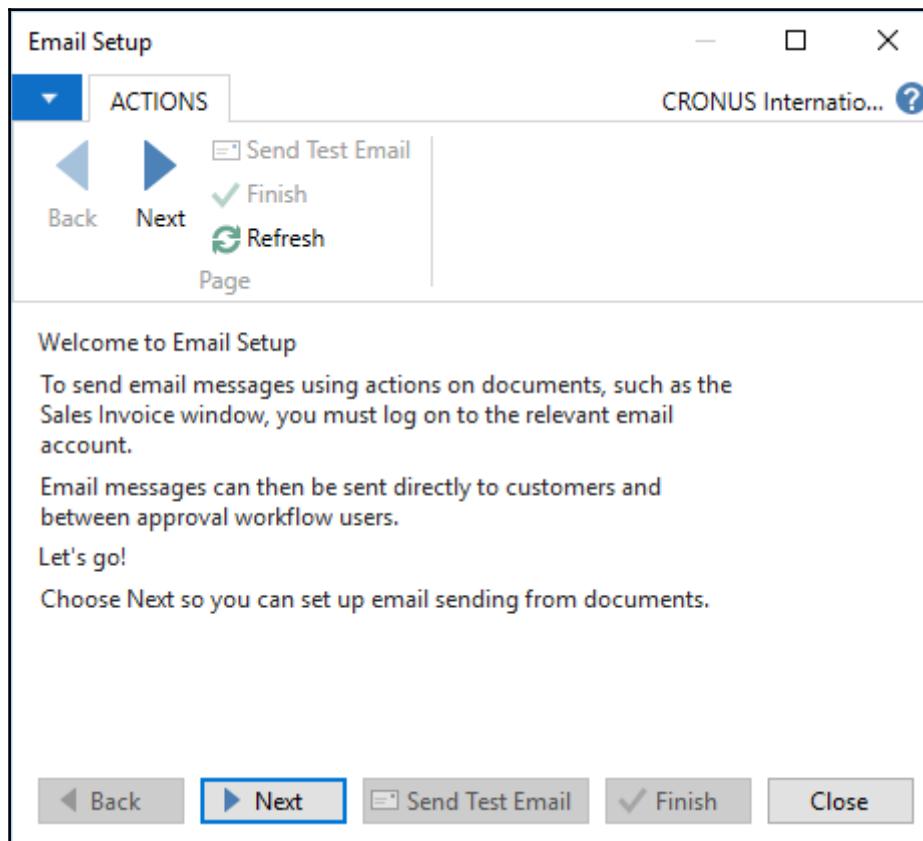
## Standard Dialog page

The **Standard Dialog** page is also a simple page format to allow the user to control a process, such as **Copy Tax Setup** (Page 476). The Standard Dialog page allows the entry of control data, such as that shown in the following screenshot:



## Navigate page

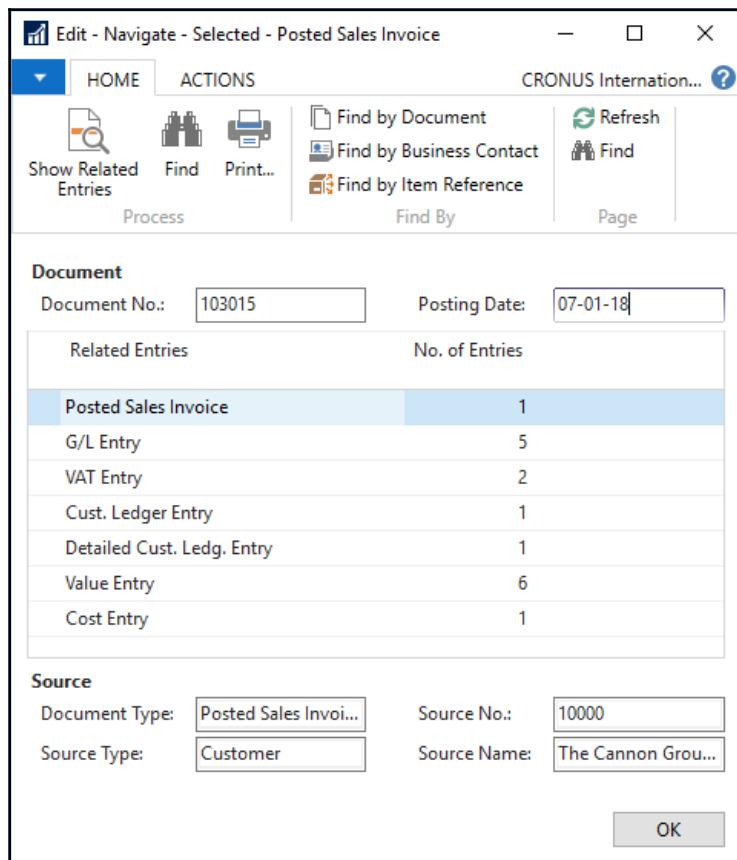
The primary use of the **Navigate** page type in NAV 2017 is as the basis for some Wizard pages. With NAV 2017, Microsoft introduced a new application feature called **Assisted Setup**. This feature uses many **Wizard** pages. A Wizard page consists of multiple user data entry screens linked together to provide a series of steps necessary to complete a task. The following screenshot shows assisted e-mail setup:



## Navigate function

The **Navigate** function has been a very powerful and unique feature of NAV since the 1990s. Somewhat confusingly, in NAV 2017, the **Navigate** function is implemented using the **Worksheet** page type, not the **Navigate** page type, which was used in earlier NAV releases.

The Navigate page (Page 344) allows the user, who may be a developer operating in user mode, to view a summary of the number and type of posted entries having the same document number and posting date as a related entry or as a user-entered value. The user can drill down to examine the individual entries. Navigate is a terrific tool for tracking down related posted entries. It can be productively used by a user, an auditor, or even by a developer. A sample Navigate page is shown in the following screenshot:



## Special pages

There are two special purpose page types. The first is a component of other objects, the second is automatically generated.

## Request page

A **Request page** is a simple page that allows the user to enter information to control the execution of a Report or **XMLport** object. Request pages can have multiple FastTabs, but can only be created as part of a Report or XMLPort object. All Request page designs will be similar to the following screenshot for the Item **Price List** (Report 715) Request page:

The screenshot shows the 'Edit - Price List' request page. At the top, there's a toolbar with 'ACTIONS' and a company name 'CRONUS International Ltd.' Below the toolbar is a sidebar with icons for 'Clear Filter' and 'Page'. The main area has several sections:

- Options**: Contains fields for 'Date' (set to 1/25/2018), 'Sales Type' (set to Customer), 'Sales Code' (empty), and 'Currency Code' (empty).
- Item**: Contains a 'Sorting' dropdown set to 'No.' (with options 'A' and 'Z').
- Show results:** A section with four filter criteria:
  - Where No. is Enter a value.
  - And Search Description is Enter a value.
  - And Assembly BOM is Select a value
  - And Inventory Posting Group is Enter a value.

Below these is a '+ Add Filter' button.
- Limit totals to:** A section with a '+ Add Filter' button.
- Item Variant**: A dropdown menu.

At the bottom right are buttons for 'Print...', 'Preview', and 'Cancel'.

## Departments page

The **Departments** page is a one-of-a-kind, system-generated page. We don't directly create a **Departments** page because it is automatically generated from the entries in the **MenuSuite** object. When we create new objects and add appropriate entries to the MenuSuite, we are providing the material needed to update the **Departments** menu/page. The look and feel of the **Departments** page cannot be changed, though individual entries can be added, changed, moved, or deleted.

The **Departments** page acts as a site map to the NAV system for the user. When we add new objects to the MenuSuite (thus, to the **Departments** menu), NAV UX design guidelines encourage the entry of duplicate links within whichever sections the user might consider looking for that page; we will later discuss the Search function that makes this task even easier. An example of the **Departments** page is shown in the following screenshot:

Departments					
Choose by department					
<b>Financial Management</b>	Payables Fixed Assets Inventory Periodic Activities Setup	<b>Manufacturing</b>	Product Design Capacities Planning	Execution Costing	
General Ledger Cash Management Cost Accounting Cash Flow Receivables					
<b>Sales &amp; Marketing</b>	Marketing Inventory & Pricing	<b>Jobs</b>			
Sales Order Processing					
<b>Purchase</b>	Inventory & Costing	<b>Resource Planning</b>			
Planning Order Processing					
<b>Warehouse</b>	Goods Handling Multipl... Inventory Assembly	<b>Service</b>	Contract Management Planning & Dispatching	Order Processing	
Orders & Contacts Planning & Execution Goods Handling Order b...					
<b>Human Resources</b>		<b>Administration</b>	IT Administration	Application Setup	

## Page parts

Several of the page types we have reviewed so far contain multiple panes, with each pane including special purpose parts. Let's look at some of the component page parts available to the developer.



Some page parts compute the displayed data on the fly, taking advantage of **FlowFields** that may require considerable system resources to process the FlowField calculations. As developers, we must be careful about causing performance problems through overuse of such displays.

## FactBox Area

The **FactBox** Area can be defined on the right side of certain page types, including Card, List, List Plus, Document, and Worksheet. A FactBox Area can contain Page parts (**CardPart** or **ListPart**), Chart parts, and System parts (Outlook, Notes, MyNotes, or Record Links). A variety of standard CardParts, ListParts, and Charts that can be used in FactBoxes are available. System parts cannot be modified. All the others can be enhanced from the standard instances, or new ones may be created from scratch.

## CardParts and ListParts

CardParts are used for FactBoxes that don't require a list. CardParts display fields or perhaps a picture control. (NAV 2017 Help contains an example of including a DotNet add-in within a FactBox to display a chart). An example of the **Customer Statistics** FactBox (Page 9082 - **Customer Statistics Factbox**) is shown in the following screenshot:

Customer Statistics - Bill-to Customer	
Customer No.:	43687129
Balance (LCY):	14,498.04
<b>Sales</b>	
Outstanding Orders (LCY):	143,011.79
Shipped Not Invd. (LCY):	0.00
Outstanding Invoices (LCY):	0.00
<b>Service</b>	
Outstanding Serv. Orders (LCY):	0.00
Serv Shipped Not Invoiced(LCY):	0.00
Outstanding Serv. Invoices (LCY):	0.00
Total (LCY):	<b>157,509.83</b>
Credit Limit (LCY):	0.00
Overdue Amounts (LCY) as of 01/29/16:	0.00
Total Sales (LCY):	14,498.04
Invoiced Prepayment Amount (LCY):	0.00

ListParts are used for FactBoxes that require a list. A list is defined as columns of repeated data. No more than two or three columns should appear in a FactBox list. The following screenshot shows the three-column **My Items** FactBox ListPart (Page 9152):

Item No.	Description	Unit Price
1000	Bicycle	4.000,00
1001	Touring Bicycle	4.000,00
1100	Front Wheel	1.000,00
1150	Front Hub	500,00
1200	Back Wheel	1.200,00

## Charts

In NAV 2017, there are two standard ways of including charts in our pages. The first one, Chart parts, is a carryover from NAV 2009. The second one, the Chart Control Add-In, was new in NAV 2013.

### Chart Part

A **Chart Part** displays list data in graphic form. A Chart Part is a default optional component of all FactBox Areas; it is not a Page Type. If a FactBox exists, it has a Chart Part option available. Chart Parts are populated by choosing one of the available charts stored in the **Chart** table (Table 2000000078). Some Charts require range parameters, others do not, as they default to a defined data range. Most of the supplied charts are two-dimensional, but a sampling of three-dimensional dynamic charts is included.

A MSDN NAV Team blog provides an extensive description of chart construction and a utility to create new charts

(<http://blogs.msdn.com/b/nav/archive/2011/06/03/chart-generator-tool-for-rtc-cgtrtc.aspx>).

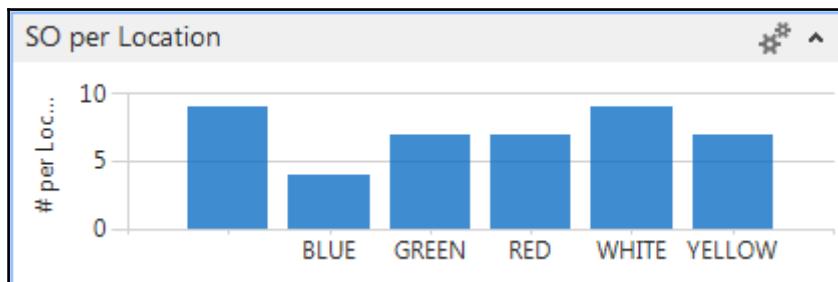


There is also a YouTube video on the topic at

([https://www.youtube.com/watch?v=RwOv3dLdXAw&x-xt-cl=85114404&x-yt-ts=1422579428](https://www.youtube.com/watch?v=RwOv3dLdXAw&x-yt-cl=85114404&x-yt-ts=1422579428)).

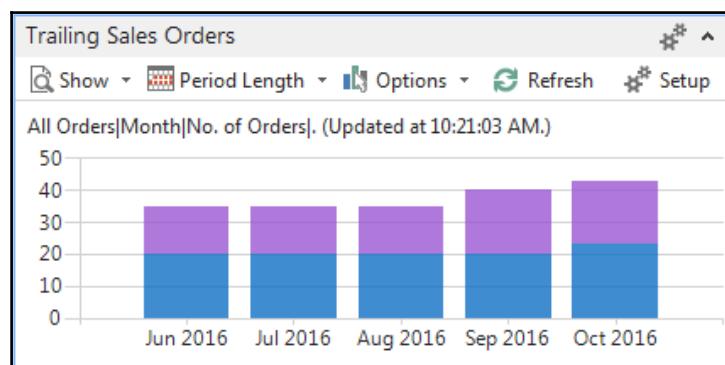
Though both of these were created for previous releases of NAV, they are useful for NAV 2017.

A sample standard chart is shown in the following screenshot:



## Chart Control Add-In

The NAV 2017 distribution includes a charting capability that is based on a Control Add-In which was created with .NET code written outside of C/SIDE and integrated into NAV. An example is the **Trailing Sales Orders** chart in a Factbox Page part (Page 760) that appears in the **Order Processor Role Center** (Page 9006), as shown in the following screenshot:



The article, *Cash Flow Chart Example*, ([https://msdn.microsoft.com/en-us/library/hh169415\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/hh169415(v=nav.90).aspx)) in the NAV 2017 Developer and IT Pro Help describes how to create charts using the Chart Control Add-In.

## Page names

Card pages are named similarly to the table with which they are associated, with the addition of the word "Card". For example, Customer table and Customer Card, Item table and Item Card, Vendor table, and Vendor Card.

List pages are named similarly to the table with which they are associated. List pages that are simple noneditable lists have the word "List" associated with the table name; for example, Customer List, Item List, and Vendor List. For each of these, 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. For example, Customer Ledger Entry table and Customer Ledger Entries page, Check Ledger Entry table and Check Ledger Entries page, Country/Region table and Countries/Regions page, Production Forecast Name table and Production Forecast Names page.

The single-record Setup tables that are used for application control information throughout NAV are named after their functional area, with the addition of 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, with the addition of the word Journal. In the standard product, several Journal pages for different purposes are associated with the same table. For example, the Sales Journal, Cash Receipts Journal, Purchases Journal, and Payments Journal all use the General Journal Line table as their Source Table (they are different pages all tied to the same table).

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 describe the relationship between the tables and the pages. However, in some cases, it's 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 use of the word **Subform** rather than **Subpage**, as in Sales Invoice Subform, is a leftover from previous versions of NAV, which had forms rather than pages. We are hopeful Microsoft will eventually fix this historical artifact and rename these pages as Subpages.

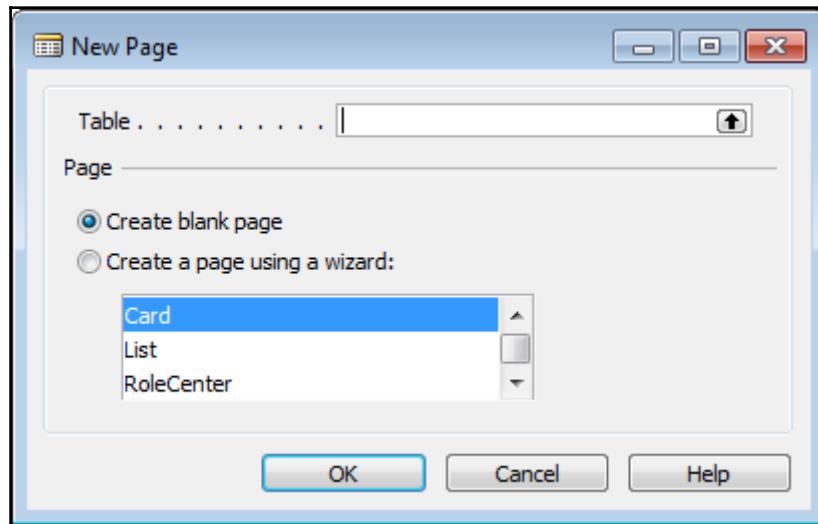
Sometimes, while naming pages, we will have a conflict between naming pages based on the associated tables and naming them based on the use of the data. For example, the menu entry Contacts invokes a main page/subpage named Contact Card and Contact Card Subform. The respective tables are the Contact table and the Contact Profile Answer table. The context usage should take precedence in the page naming as was done here.

## Page Designer

The **Page Designer** is accessed from within the Development Environment through **Tools | Object Designer | Page**. The Page Designer can be opened either for creation of a new page by using the **New** button or for editing an existing page by highlighting the target object, then clicking the **Design** button.

## New Page wizard

When we click on the **New** button or **F3** or **Edit | New**, we will bring up the **New Page** wizard, as you can see in the following screenshot:



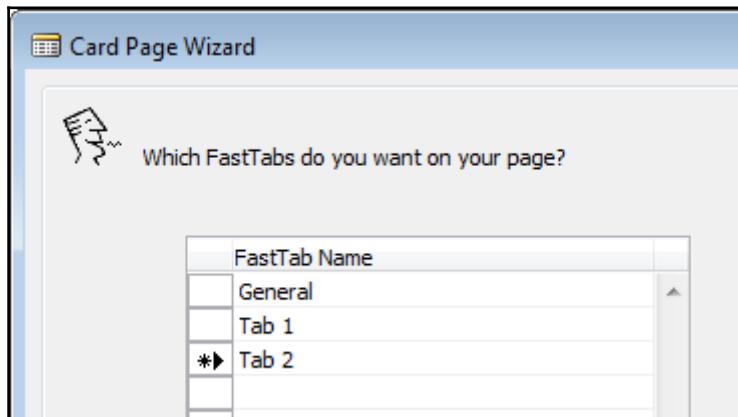
We can proceed to the Page Designer with the **Page Type** property set to **Card** and no Source Table defined by clicking on **OK** with **Create blank page** selected, but not entering a **Table** name or number, or we could enter a **Table** name or number, select **Create a page** using a wizard, and select **Page Type**. This will take us to the **Page Wizard** with the **Page Type** property set to our choice and the Source Table assigned to the table we entered. It's almost always less effort to use the wizard to create at least a rough version of a page design, and then modify the generated object structure and code to get the ultimate desired result.

To use the **Page Wizard**, first we enter the name or number of the table to which we want our page to be bound. Then, we choose what **Page Type** we want to create--**Card**, **List**, **RoleCenter**, **CardPart**, **ListPart**, **Document**, **Worksheet**, **ListPlus**, **ConfirmationDialog**, **StandardDialog**, or **NavigatePage**. The subsequent steps that the **Page Wizard** will take us through, depends on the **Page Type** chosen. The following chart shows which options are available for each of the Page Types through the use of the **Page Wizard**:

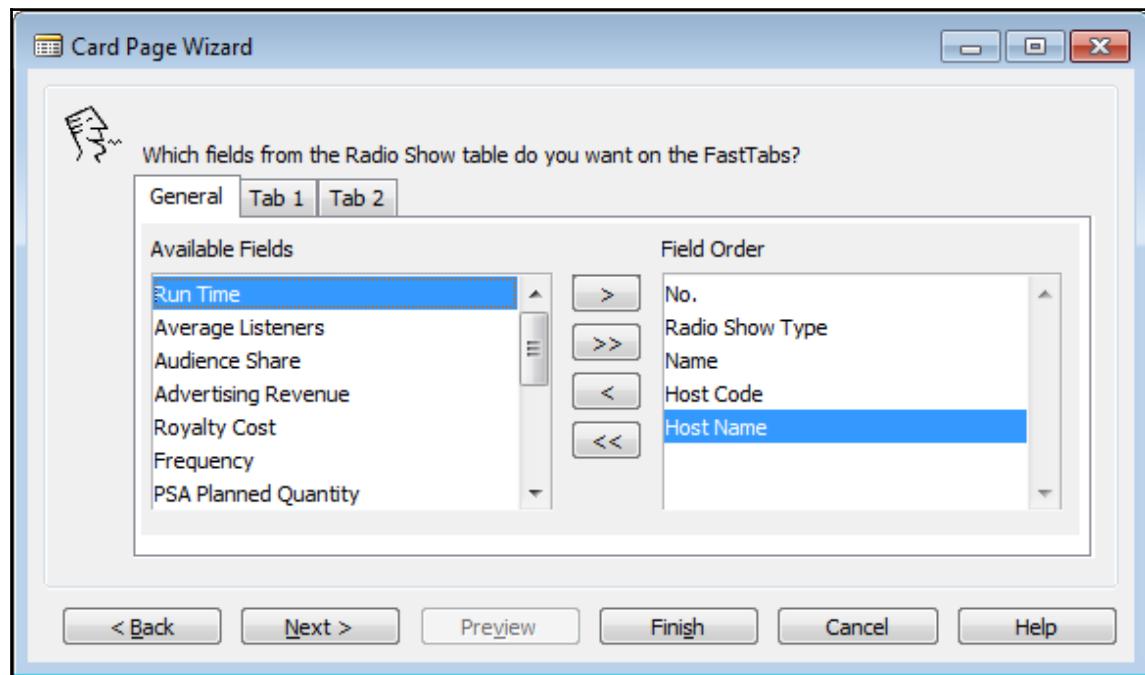
Page Type	Fast Tabs	Fields to Display	FactBoxes	Page Designer
RoleCenter	-	-	-	P
Card	P	P	P	P
List	-	P	P	P
Document	P	P	P	P
ListPlus	P	P	P	P
Worksheet	-	P	P	P
ConfirmationDialog	P	P	-	P
StandardDialog	-	-	-	P
NavigatePage	P	P	P	P
CardPart	-	P	-	P
ListPart	-	P	-	P

To see what the **Page Wizard** looks like, we'll step through an example definition of a Card page based on our Table 50000 - Radio Show. The goal of this example is only to illustrate the Page wizard process; we aren't creating a page that we'll keep for our Radio Show application.

Invoke the **Page Designer** Wizard by clicking on the New button with the **Page** button highlighted. Enter 50000 in the **Table** field, select the **Create a page using a wizard:** option, select **Card**, and click on the **OK** button. The next screen allows us to define FastTabs. It displays, with a default first FastTab titled **General**. For our example, we'll add two more FastTabs: **Tab 1** and **Tab 2**:



We will click on the **Next** button and proceed to the screen to assign fields from our bound table to each of the individual tabs. The single arrow buttons move one field either onto or off of the selected tab. The double arrow buttons move all the fields in the chosen direction. In the following screenshot, five fields have been selected and assigned to the **General** tab:



We can move fields back and forth between the chosen and available sets, as well as reposition them in **Field Order** until we are satisfied with the result. If we realize that we should have defined our tabs differently, we can click on the **Back** button and return to the FastTab definition screen to revise the tab assignments.

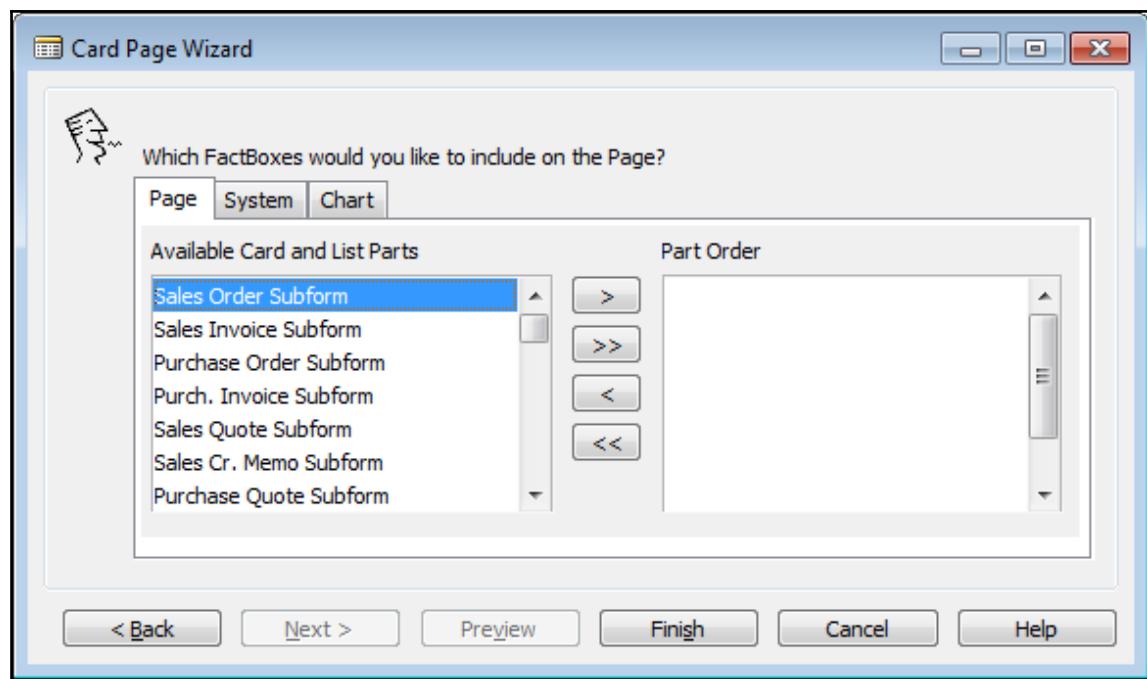
After all the desired fields have been assigned to the appropriate tab in the desired order, we will click on **Next** and move on to the screen that allows us to assign **FactBoxes** to the page. Many different page components are made available by the Wizard for assignment as FactBoxes.

As we see in the following screenshot, many of the choices are not appropriate for use in most pages. We can only select previously created page parts for **FactBoxes**. This leads to an important concept.



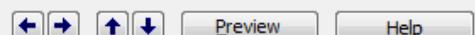
Even though we can add component parts later, it's good practice to plan our page design layout ahead of time and construct the component parts first (even if we must flesh them out or modify them later).

If we had not yet defined the **FactBoxes** for our current page design, we could pick other similar page components and then make the appropriate code replacement later, in the **Page Designer**:



Once we have done all the assignment work that is feasible within the **Page Wizard**, we will click on **Finish**. The **Page Wizard** will generate the object structure and C/AL code for our defined page and present the results to us in the **Page Designer**, as shown in the following screenshot:

E.. Type	SubType	SourceExpr	Name	Caption
Container	ContentArea		<Control1>	<Control1>
Group	Group		General	<General>
Field		"No."	<No.>	<No.>
Field		"Radio Show Type"	<Radio Show Typ...	<Radio Show Type>
Field		"Name"	<Name>	<Name>
Field		"Host Code"	<Host Code>	<Host Code>
Field		"Host Name"	<Host Name>	<Host Name>
Group	Group		Tab 1	<Tab 1>
Field		"Run Time"	<Run Time>	<Run Time>
Field		"Frequency"	<Frequency>	<Frequency>
Field		"PSA Planned Quantity"	<PSA Planned Qu...	<PSA Planned Quantity>
Field		"Ads Planned Quantity"	<Ads Planned Qu...	<Ads Planned Quantity>
Field		"News Required"	<News Required>	<News Required>
Field		"News Duration"	<News Duration>	<News Duration>
Field		"Sports Required"	<Sports Required>	<Sports Required>
Field		"Sports Duration"	<Sports Duration>	<Sports Duration>
Field		"Weather Required"	<Weather Requir...	<Weather Required>
Field		"Weather Duration"	<Weather Durati...	<Weather Duration>
Group	Group		Tab 2	<Tab 2>
Field		"Average Listeners"	<Average Listen...	<Average Listeners>
Field		"Audience Share"	<Audience Share>	<Audience Share>
Field		"Advertising Revenue"	<Advertising Rev...	<Advertising Revenue>
Field		"Royalty Cost"	<Royalty Cost>	<Royalty Cost>
Container	FactBoxArea		<Control24>	<Control24>
Part	Page		<Customer Statisti...	<Customer Statistics FactBox>
Part	Page		<Vendor Details Fa...	<Vendor Details FactBox>



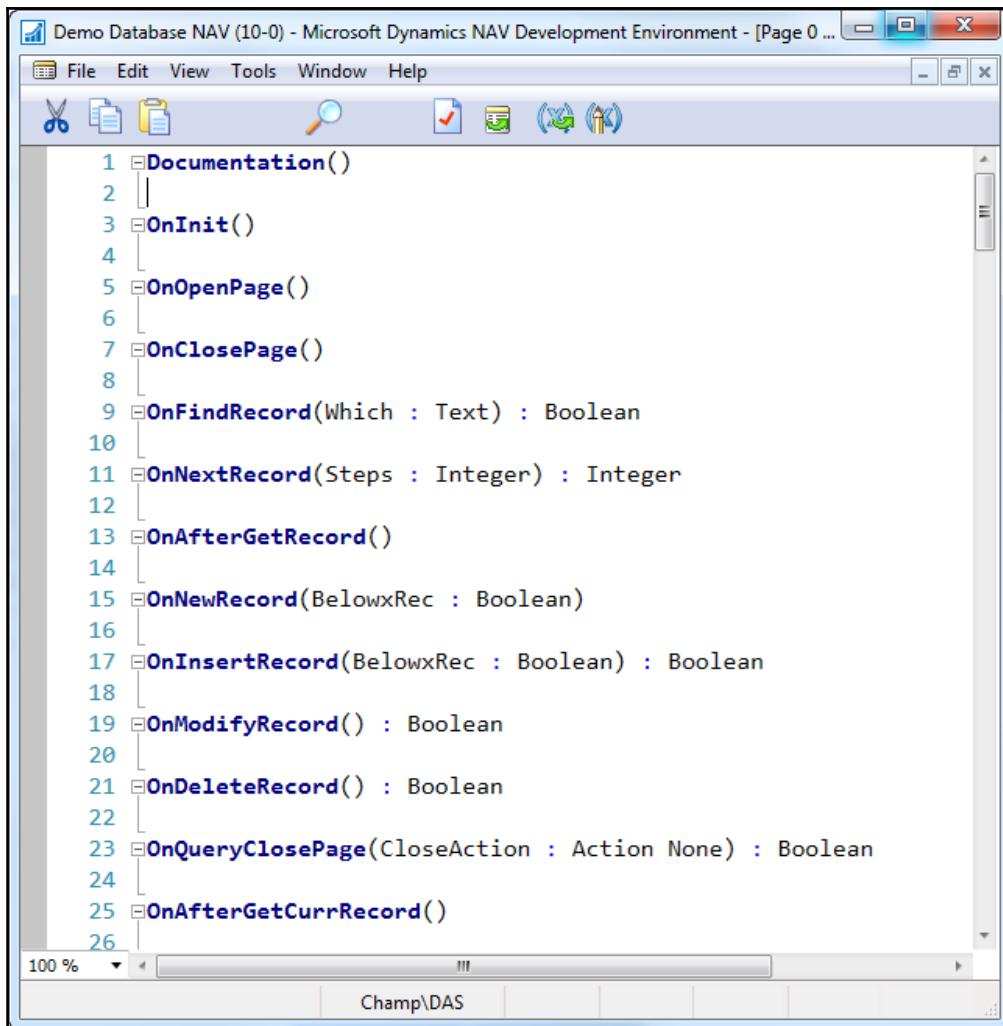
We realize now that it would have been more efficient to have planned ahead and created our custom FactBox page parts before we used the Wizard to create our card page. The alternative we used was to simply add a couple of FactBox page parts as place holders that we can replace later. This gives us the structure we want, and compensates for our lack of planning. We could wait and just add the FactBoxes later, but we wouldn't be taking advantage of the help the Wizard can provide.

## Page Components

All pages are made up of certain common components. The basic elements of a page object are the page triggers, page properties, controls, control triggers, and control properties.

## Page Triggers

The following screenshot shows the page triggers. The **Help** section's **Page and Action Triggers** provides good general guidance to the event which causes each page trigger to fire. Note that the **On Query Close Page** trigger isn't related to any Query object action:



Demo Database NAV (10-0) - Microsoft Dynamics NAV Development Environment - [Page 0 ...]

```
File Edit View Tools Window Help
1 Documentation()
2
3 OnInit()
4
5 OnOpenPage()
6
7 OnClosePage()
8
9 OnFindRecord(Which : Text) : Boolean
10
11 OnNextRecord(Steps : Integer) : Integer
12
13 OnAfterGetRecord()
14
15 OnNewRecord(BelowxRec : Boolean)
16
17 OnInsertRecord(BelowxRec : Boolean) : Boolean
18
19 OnModifyRecord() : Boolean
20
21 OnDeleteRecord() : Boolean
22
23 OnQueryClosePage(CloseAction : Action None) : Boolean
24
25 OnAfterGetCurrRecord()
26
```

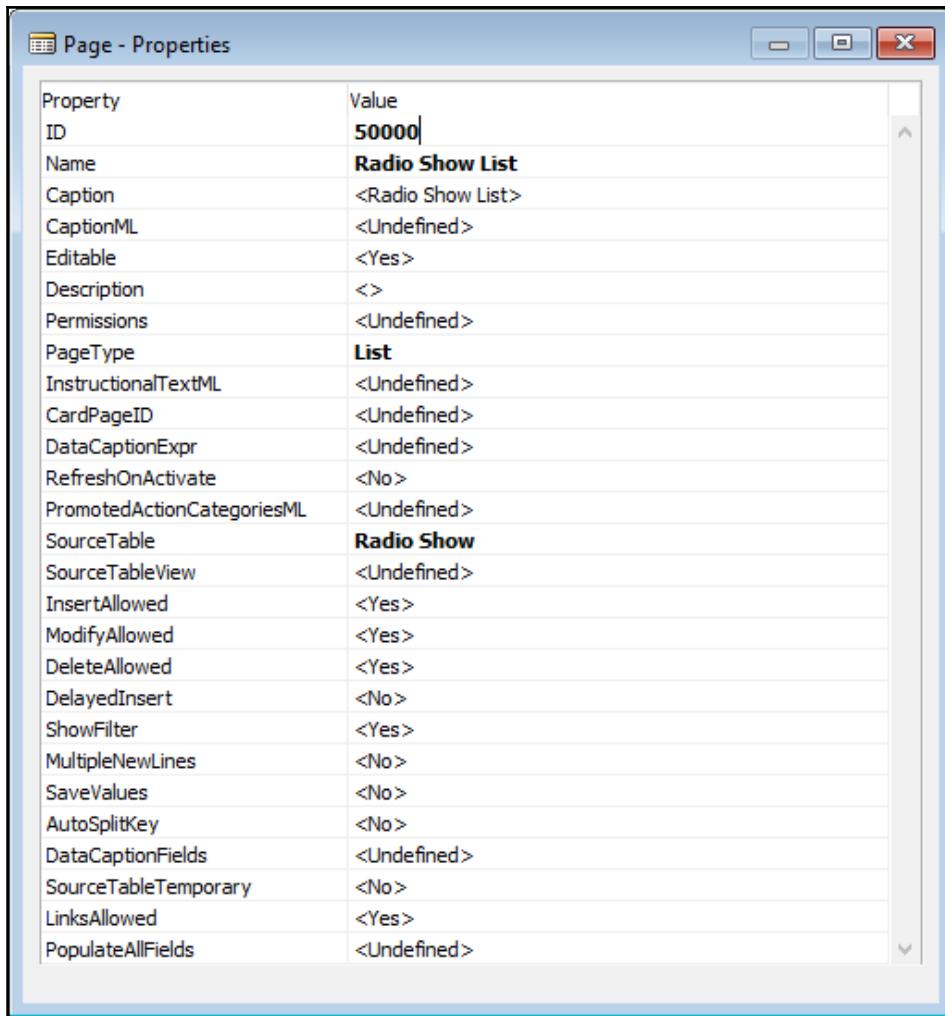
100 %    Champ\Das

In general, according to best practices, we should minimize the C/AL code placed in Page triggers, putting code in a Table or Field trigger or calling a Codeunit Library function instead. However, many standard pages include a modest amount of code supporting page-specific filter or display functions. When we develop a new page, it's always a good idea to look for similar pages in the standard product and be guided by how those pages operate internally. Sometimes, special display requirements result in complex code being required within a page. It is important that the code in a page be used only to manage the data display, not for data modification.

To execute C/AL code in pages you can also use Events. Each page object automatically generates events that you can subscribe to. Using events is preferred when adding code to page objects shipped by Microsoft to avoid any merge challenges when Microsoft changes the page in a new release. You can read more about Page Events in MSDN at [https://msdn.microsoft.com/en-us/library/mt299406\(v=nav.90\).aspx#PageEvents](https://msdn.microsoft.com/en-us/library/mt299406(v=nav.90).aspx#PageEvents).

## Page properties

We will now look at the properties of the **Radio Show List** page we created earlier. The list of available page properties is the same for all page types. The values of those properties vary considerably from one page to another, even more from one page type to another. The following screenshot shows the **Page - Properties** screen of our **Radio Show List** page (Page 50000). This screen is accessed by opening **Page 50000** in **Page Designer**, highlighting the first empty line in the **Controls** list and clicking on the **Properties** icon or **Shift + F4** or **View | Properties**:



We can see that many of these properties are still in their default condition (they are not shown in bold letters). 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 page allows the individual control to determine the appropriate Editable property value.
- **Description:** This is 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 Table Data in the specified table objects. For example, users of Page 499 (Available - Sales Lines) are allowed to only read or modify (Permissions for Sales Line = **rm**) the data in the **Sales Line** table.



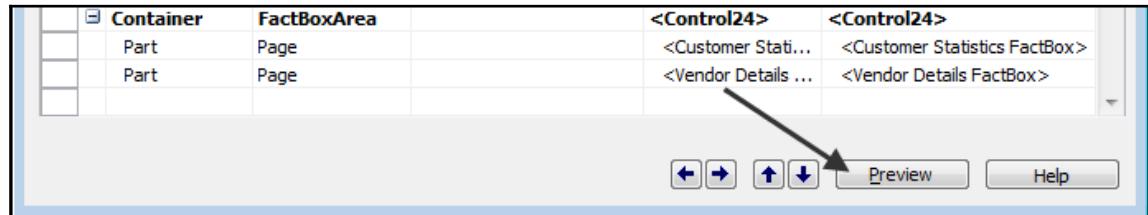
Whenever defining special permissions, be sure to test with an end-user license. In fact, it's always important to test with an end-user license.

- **PageType:** This specifies how this page will be displayed using one of the available ten page types: **RoleCenter**, **Card**, **List**, **ListPlus**, **Worksheet**, **ConfirmationDialog**, **StandardDialog**, **NavigatePage**, **CardPart**, **ListPart**.
- **CardPageID:** This is the ID of the Card page that should be launched when the user double-clicks on an entry in the list. This is only used on List pages.
- **RefreshOnActivate:** When this is set to **Yes**, it causes the page to refresh when the page is activated. This property is unsupported by the Web Client.
- **PromotedActionCategoriesML:** This allows changing the language for **Promoted ActionCategories** from the default English (ENU) to another language or to extend the number of **Promoted ActionCategories** from the standard three options--**New**, **Process**, **Reports**--to add up to seven more categories. See the Help section, *How to: Define Promoted Action Categories Captions for the Ribbon*.
- **SourceTable:** This is the name of the table to which the page is bound.
- **SourceTableView:** This can be utilized to automatically apply defined filters and/or open the page with a key other than the primary key.
- **ShowFilter:** This is set to **No** to make the **Filter** pane default to **Not Visible**. The user can still make the **Filter** pane visible.

- **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 are completed. This property is generally set to **Yes** when **AutoSplitKey** (see following) is set to **Yes**. It allows complex new data records to be entered with all the necessary fields completed.
- **MultipleNewLines:** When set to **Yes**, this property supposedly allows the insertion of multiple new lines between existing records. However, it is set to **No** in the standard Order forms from Microsoft. This indicates that this property is no longer active in NAV 2017.
- **SaveValues:** If set to **Yes**, this causes user-specific entered control values to be retained and redisplayed when the page is invoked another time.
- **AutoSplitKey:** This allows for the automatic assignment of a primary key, provided the last field in the primary key is an integer; there are rare 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 a new 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 value being used for the new entry key-terminating integer value. Because 10,000 (the automatic increment) can only be divided by 2 and rounded to a non-zero integer result 13 times, only 13 new automatically numbered entries can be inserted between two previously recorded entries by the **AutoSplitKey** function.
- **SourceTableTemporary:** This allows the use of a temporary table as the Source Table for the page. This can be very useful where there is a need to display data based on the structure of a table, but not using the table data as it persists in the database. Examples of such application usage are in [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 our code must populate the temporary table in memory.

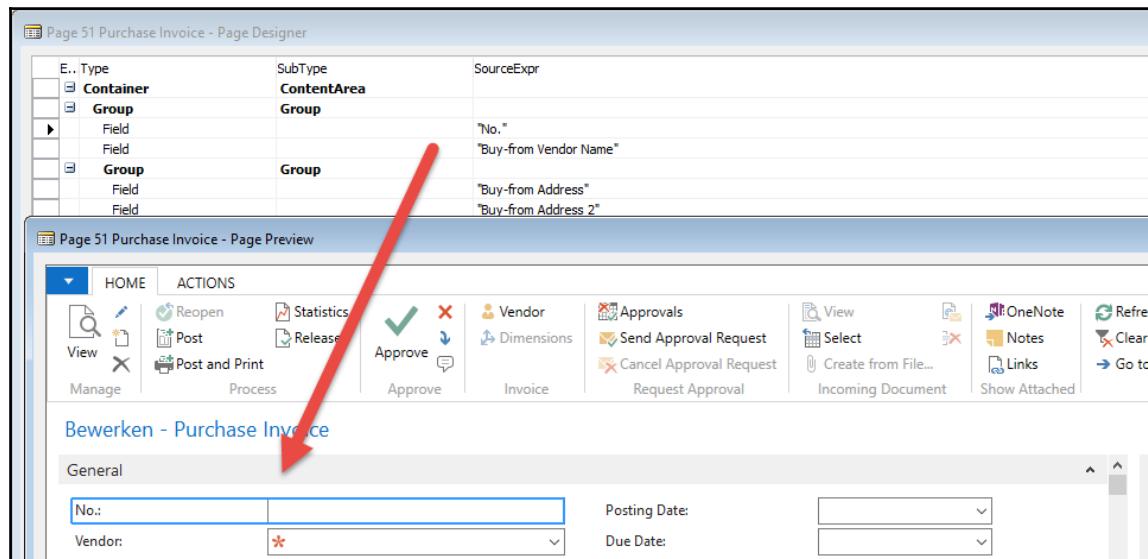
## Page Preview Tool

The **Page Designer** in NAV 2017 has a Page Preview tool that is very helpful in defining our control placement and action menu layout. The button to invoke a Page Preview is shown in the following screenshot:

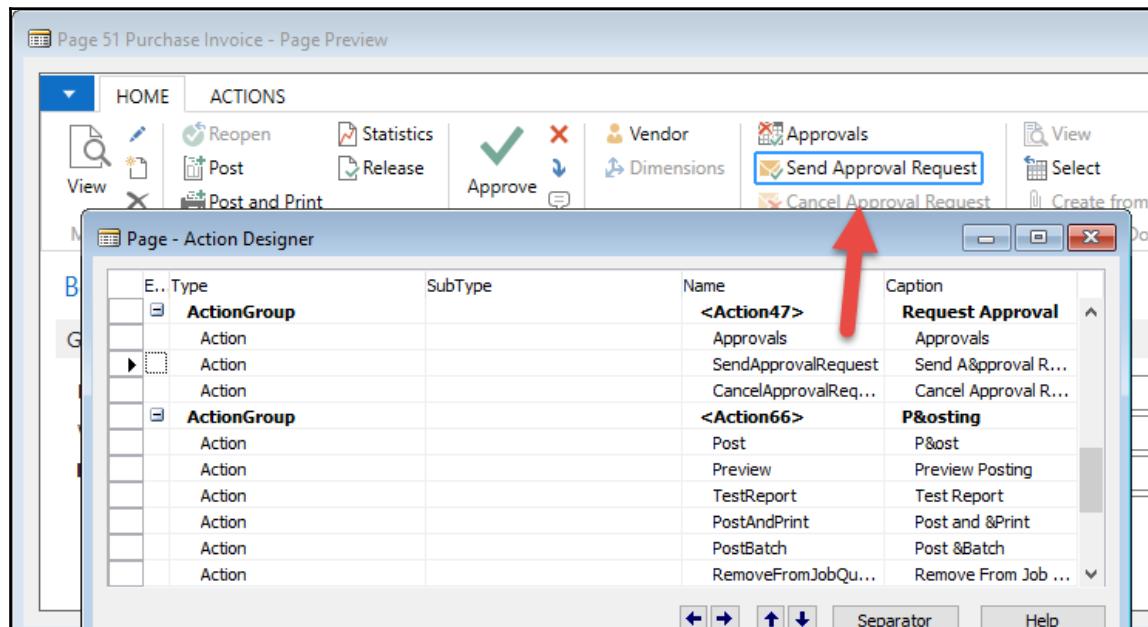


If we click on the **Preview** button while we have a page open in the **Page Designer**, a preview of that page's layout will display. The controls and actions are not active in the preview (this is display only), but we can display all the ribbon tabs and their controls.

The Preview screen is interactively linked to the **Page Designer** and its subordinate **Action Designer**. When we click on a control line in the **Page Designer**, or an Action line in the **Action Designer**, **Page Preview** highlights the generated object, or when we click on a control or action displayed in the previewed page, the corresponding line in **Page Designer** or **Action Designer** is highlighted. An example of a highlighted control is shown in the following partial page screenshot:



In the following screenshot, an action is highlighted in the Page Preview:



## Inheritance

One of the attributes of an object-oriented system is the inheritance of properties. While NAV is object based rather than object oriented, the properties that affect data validation are inherited. Properties such as decimal formatting are also inherited. If a 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 the field definition and the control definition. This basic concept applies to 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 others can be changed. This is another example of why 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.

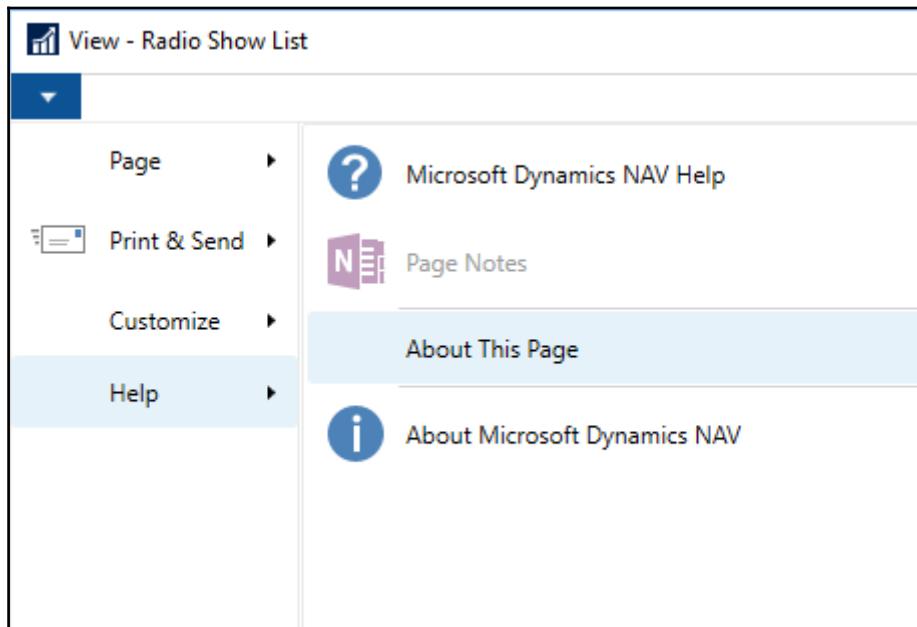
# WDTU Page Enhancement - part 1

Before we move on to learn about controls and actions, let's do some basic enhancement work on our WDTU Radio Show application. In the *Getting started with application design* section in Chapter 1, *Introduction to NAV 2017*, we created several minimal pages, and later added new fields to our **Radio Show** master table (**Table 50000**). We'll now enhance the Radio Show's List and Card Page to include those added fields.

Because our previous page development work resulted in simple pages, we have the opportunity to decide whether we want to start with **New Page Wizard** and replace our original pages, or use **Page Designer** to modify the original pages. If we had done any significant work on these pages previously in **Page Designer**, the choice to go right to the Page Designer would be easy. Let's do a quick evaluation to help us make our decision. First, let's take a look at the existing **Radio Show List** page:

No.	Radio Show Type	Name	Run Time	Host Code	Host Name	Average
RS001	TALK	CeCe and Friends	2 hours	CECE	CeCe Grace	
RS002	MUSIC	Alec Rocks and Bobs	2 hours	ALEC	Alec Benito	

We want to compare the list of fields that exist in the source table (**Radio Show - 50000**) to what we see is already in the page. If there are only a couple of fields missing, it will be more efficient to do our work in the **Page Designer**. The quickest way to inspect the fields of the source table is to use the **About This Page** option in the **Help** tab available from the drop-down menu at the left end of the **Global Command Bar**, as shown in the following screenshot:



When we click on **About This Page** or press **Ctrl + Alt + F1**, the following screen displays:

The screenshot shows a software window titled "About This Page: View - Radio Show List". The window has a standard title bar with minimize, maximize, and close buttons. Below the title bar is a toolbar labeled "ACTIONS" with a question mark icon. The main content area is divided into two sections: "Page Information" and "Table Fields".

**Page Information**

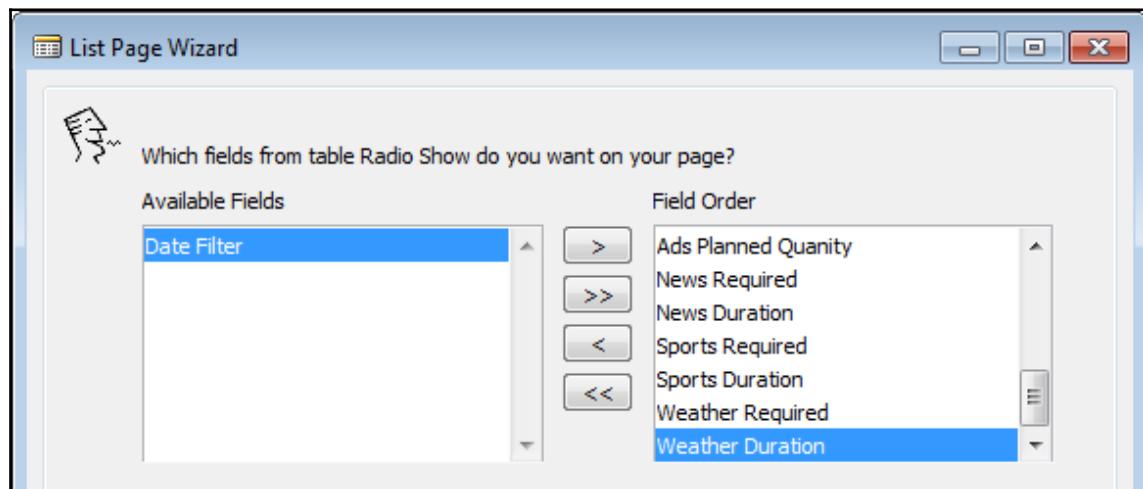
Page:	Radio Show List (50000)
Page Type:	List
Page Mode:	View
SourceTable:	Radio Show (50000)
Rec:	RS001

**Table Fields**

No. (1) (PK):	RS001
Ads Planned Quantity (1020):	0
Advertising Revenue (120):	0
Audience Share (110):	0
Average Listeners (100):	0
Frequency (1000):	

When we scroll down the list of fields in **Table 50000**, which is displayed alphabetically and not by field number or in order of placement in the page, we will see that there are quite a few fields in the table that aren't in our page. This makes it easy to conclude that we should use Page Wizard to create the new version of our **Page 50000 - Radio Show List**.

Although the wizard allows us to choose and sequence fields in our new list form, for the sake of simplicity, we will just insert all the fields at once, in the order they appear in the table. In other words, we will choose the >> button to include all fields. Then, because we know that the **Date Filter** field is only for filter control of related tables and will not contain visible data, we will remove that field. Our wizard screen will look like the following screenshot:



Finish the page, save it as **Page 50000, Radio Show List**, and overwrite the old version. If we wanted to be very safe, before making any changes, we would have done an **Export** of the original version of Page 50000 as a .fob file (using **File | Export**).

Next, we want to also create a new layout for **Radio Show Card**. We'll make the same choice for the same reasons, to use Page Wizard to create a new version of the card page. When we review the data fields, we decide that we should have three FastTabs: **General**, **Requirements**, and **Statistics**. As before, the **Date Filter** field should not be on the page. After we have generated, compiled, and saved our new **Radio Show Card**, it will look like this:

View - Radio Show Card - RS001

HOME CRONUS International Ltd. ?

RS001

General

No.: RS001

Radio Show Type: TALK

Name: CeCe and Friends

Run Time: 2 hours

Frequency: 0

Host Code: CECE

Host Name: CeCe Grace

Requirements

PSA Planned Quantity: 0

Ads Planned Quantity: 0

News Required:

News Duration:

Sports Required:

Sports Duration:

Weather Required:

Weather Duration:

Statistics

Average Listeners: 0,00 ...

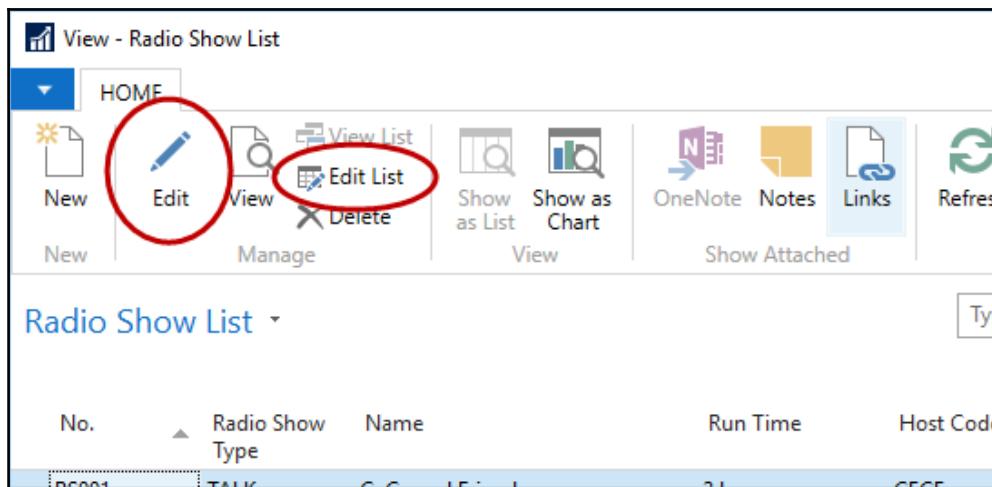
Audience Share: 0,00 ...

Advertising Revenue: 0,00 ...

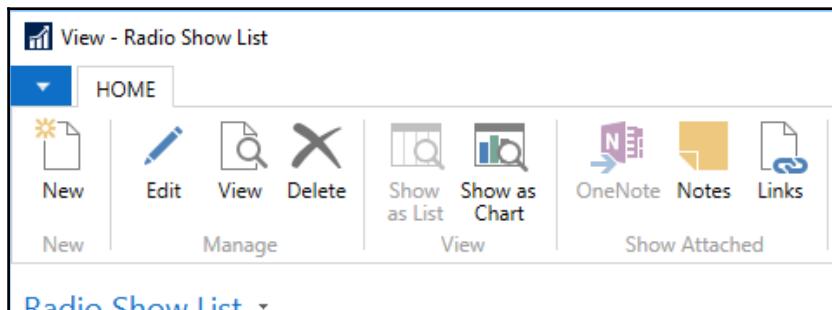
Royalty Cost: 0,00 ...

**Close**

Our final step at this point is to connect the **Radio Show Card** to the **Radio Show List** page, so that, when the user double-clicks on a list entry, the Card page will be invoked, showing the list of selected entry. This is a simple matter of opening our new Page 50000 in **Page Designer**, highlighting the first empty line in **Controls list** and clicking on the **Properties** icon or pressing *Shift + F4* or **View | Properties**. In the list of page properties display, we will find **CardPageID**. Fill in that property with either the name (Radio Show Card) or Object ID number (50001) of the target card, then save, compile, and run it. We should see a ribbon with both **Edit** and **Edit List**, as shown in the following screenshot:



Clicking on **Edit** will bring up Radio Show Card. Clicking on **Edit List** will make the line in the list editable in place within the List page. If we don't want the user to be able to edit within the list, we should change the List page property from **Editable** to **No** and the **Edit List** option will not be available, as shown in the following screenshot:



## Page Controls

Controls on pages serve a variety of purposes. Some controls display information on pages. This can be data from the database, static material, pictures, or the results of a C/AL expression. Container controls 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 also makes it easy for the user to consider a set of controls as a group. The user can make all the controls on FastTabs visible or invisible by expanding or collapsing the FastTabs. The user also has the option to show or not to show a particular FastTabs as a part of the page customizing capability. The **Help** sections' **Pages Overview** and **How to: Create a Page** provide good background guidance on the organization of controls within page types for NAV 2017.

The following screenshot from **Page Designer** shows all the data controls on **Page 5600 Fixed Asset Card**. The first column, **Expanded**, is a + or -, indicating if the section is expanded or not. **Type** and **SubType** define how this control is interpreted by the **Role Tailored Client**. **SourceExpr** defines the value of the control. **Name** is the internal reference name of the control. The contents of **Caption** is what will appear on screen. The default values for **Name** and **Caption** originate from the table definition:

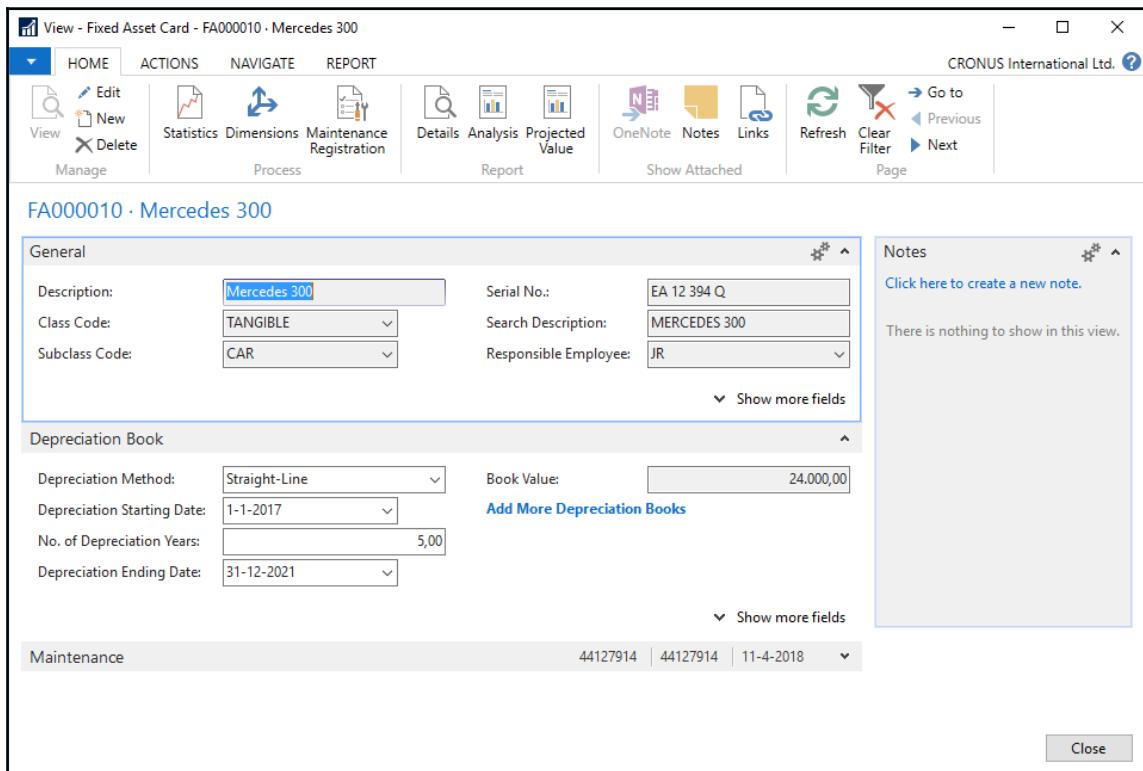
Page 5600 Fixed Asset Card - Page Designer

E.. Type	SubType	SourceExpr	Name	Caption
▶ Container	ContentArea	[...]	<Control190000...>	<Control1900000001>
Group	Group		<Control1>	General
Field		"No."	<No.>	<No.>
Field		Description	<Description>	<Description>
Group	Group		<Control34>	<Control34>
Field		"FA Class Code"	<FA Class C...	Class Code
Field		"FA Subclass Code"	<FA Subclas...	Subclass Code
Field		"FA Location Code"	<FA Location ...	Location Code
Field		"Budgeted Asset"	<Budgeted As...	<Budgeted Asset>
Field		"Serial No."	<Serial No.>	<Serial No.>
Field		"Main Asset/Component"	<Main Asset/C...	<Main Asset/Component>
Field		"Component of Main Asset"	<Component o...	<Component of Main Asset>
Field		"Search Description"	<Search Descr...	<Search Description>
Field		"Responsible Employee"	<Responsible ...	<Responsible Employee>
Field		Inactive	<Inactive>	<Inactive>
Field		Blocked	<Blocked>	<Blocked>
Field		Acquired	<Acquired>	<Acquired>
Field		"Last Date Modified"	<Last Date Mo...	<Last Date Modified>
Group	Group		<Control27>	Depreciation Book
Field		FADepreciationBook."Deprec..."	DepreciationBo...	Depreciation Book Code
Field		FADepreciationBook."FA Pos..."	FAPostingGroup	Posting Group
Field		FADepreciationBook."Deprec..."	DepreciationM...	Depreciation Method
Group	Group		<Control33>	<Control33>
Field		FADepreciationBook."Deprec..."	Deprecation...	Deprecation Starting Date
Field		FADepreciationBook."No. of ..."	NumberOfDe...	No. of Depreciation Years
Field		FADepreciationBook."Deprec..."	Deprecation...	Deprecation Ending Date
Field		FADepreciationBook."Book V..."	BookValue	Book Value
Group	Group		<Control38>	<Control38>
Field		AddMoreDeprBooksLbl	AddMoreDep...	<AddMoreDeprBooks>
Part	Page		DepreciationBook	Depreciation Books
Group	Group		<Control1903...>	Maintenance
Field		"Vendor No."	<Vendor No.>	<Vendor No.>
Field		"Maintenance Vendor No."	<Maintenance ...	<Maintenance Vendor No.>
Field		"Under Maintenance"	<Under Mainte...	<Under Maintenance>
Field		"Next Service Date"	<Next Service ...	<Next Service Date>
Field		"Warranty Date"	<Warranty Da...	<Warranty Date>
Field		Insured	<Insured>	<Insured>
Container	FactBoxArea		<Control190000...>	<Control1900000007>
Part	System		<RecordLinks>	<RecordLinks>
Part	System		<Notes>	<Notes>

[ Previous ] [ Next ] [ First ] [ Last ] [ Home ] [ Help ]

The Page's control structure can be seen in the indented format shown in the preceding screenshot. The **Container** controls define the primary parts of the page structure. The next level of structure is the **Group** control level. In this page, those are the **General**, **Depreciation Book** and **Maintenance** groups, each of which represents a FastTab. Indented under each Group control are Field controls.

The **Fixed Asset Card** page, including the action ribbon, is displayed in the following screenshot:



## Control types

There are four primary types of page controls: **Container**, **Group**, **Field**, and **Part**. Container, Group, and Field controls are used to define the content area of pages. Part controls are used to define FactBoxes and embedded subpages. When designing pages that may be used by different client types, such as the Web Client, we will need to be aware that some controls operate differently or are not supported in all clients. Each customized page should be thoroughly tested in each client environment where it may be used.

## Container controls

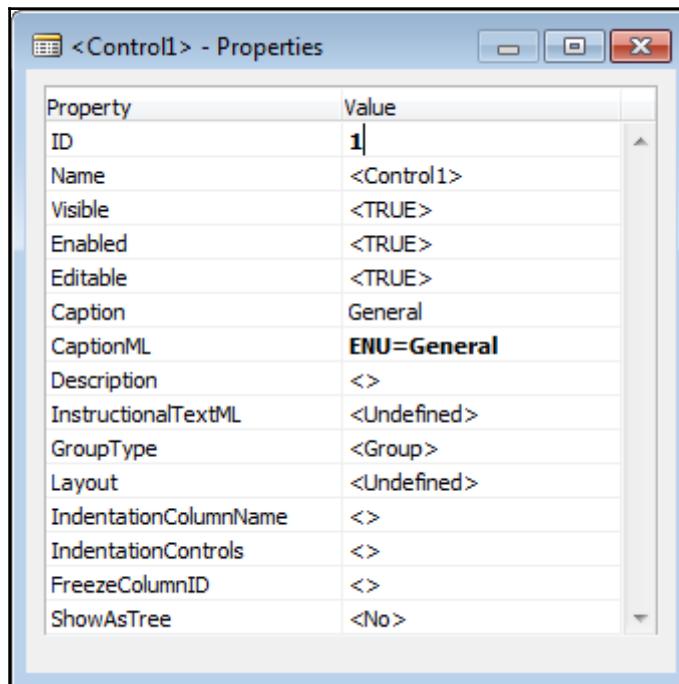
Container controls can be one of three subtypes: **ContentArea**, **FactBoxArea**, or **RoleCenterArea**. Container controls define the root-level primary structures within a page. All page types start with a Container control. The **RoleCenterArea** Container control can only be used on a **RoleCenter** page type. A page can only have one instance of each container subtype.

## Group controls

Group controls provide the second level of structure within a page. **Group** controls are the home for fields. Almost every page has at least one **Group** control. The following screenshot from **Page 5600 Fixed Asset Card**, with all the Group controls collapsed, shows two Container controls and three Group controls. Also showing is a page Part control that displays a Page Part as FastTabs:

E...	Type	SubType	SourceExpr	Name	Caption
▶	Container	ContentArea	[.....]	<Control190000...	<Control1900000001>
+	Group	Group		<Control1>	<i>General</i>
+	Group	Group		<Control27>	<i>Depreciation Book</i>
	Part	Page		DepreciationBook	Depreciation Books
+	Group	Group		<Control19035...	<i>Maintenance</i>
+	Container	FactBoxArea		<Control190000...	<Control1900000007>

The properties of a **Group** control are shown in the following screenshot:



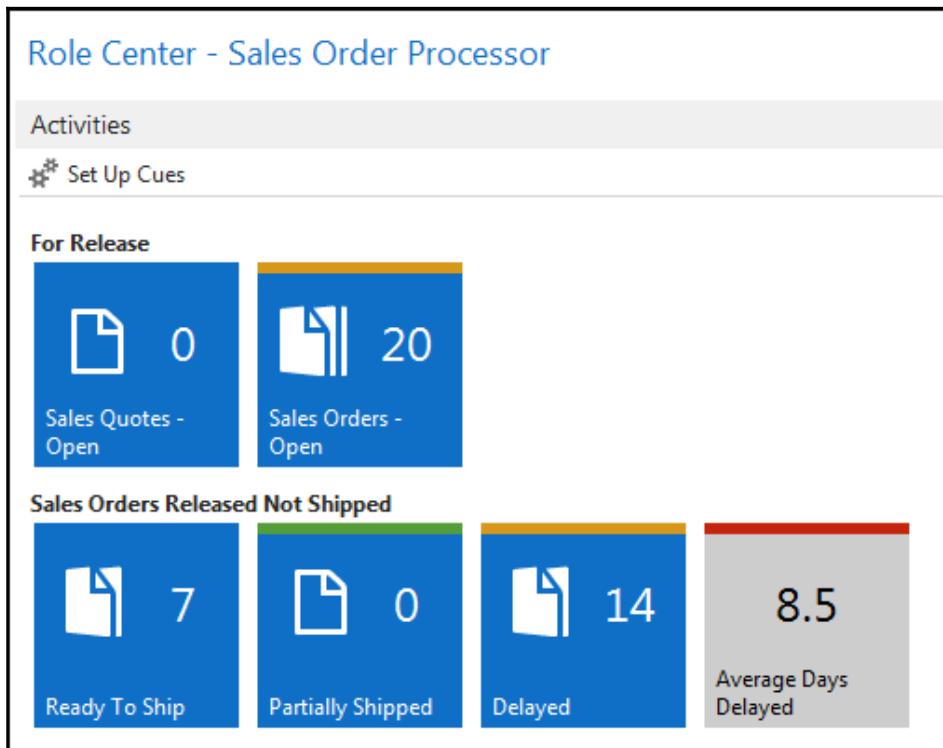
Several of the **Group** control properties are particularly significant because of their effect on all the fields within the group:

- **Visible:** TRUE or FALSE, defaulting to TRUE. The **Visible** property can be assigned a Boolean expression, which can be evaluated during processing. This allows to dynamically change the visibility of a group of fields during the processing based on some variable condition (dynamic processing must occur in either the **OnInit**, **OnOpenPage**, or **OnAfterGetCurrRecord** trigger and the variable must have its **IncludeInDataSet** property set to Yes).
- **Enabled:** TRUE or FALSE, defaulting to TRUE. The **Enabled** property can be assigned a Boolean expression to allow dynamically changing the enabling of a group of fields.

- **Editable:** TRUE or FALSE, defaulting to TRUE. The **Editable** property can be assigned a Boolean expression to allow dynamically changing the editability of a group of fields.
- **GroupType:** This can be one of the five choices: **Group**, **Repeater**, **CueGroup**, **FixedLayout**, or **GridLayout**. The **GroupType** property is visible on the **PageDesigner** screen in the column headed **SubType** (see the preceding **PageDesigner** screenshot).
- **Group:** This is used in Card type pages as the general structure for fields, which are then displayed in the sequence in which they appear in the **Page Designer** group.
- **Repeater:** This is used in List type pages as the structure within which fields are defined and then displayed as repeated rows.
- **CueGroup:** This is used for Role Center pages as the structure for the actions that are the primary focus of a user's work day. The **CueGroups** are found in page parts, typically having the word **Activities** in their name and included in Role Center page definitions. The following screenshot shows two **CueGroups** defined in **Page Designer**:

Page 9060 SO Processor Activities - Page Designer					
E..	Type	SubType	SourceExpr	Name	Caption
	Container	ContentArea		<Control1900000...	<Control1900000001>
	Group	CueGroup		<Control1>	For Release
		Field	"Sales Quotes - Open"	<Sales Quotes - ...	<Sales Quotes - Open>
		Field	"Sales Orders - Open"	<Sales Orders - ...	<Sales Orders - Open>
	Group	CueGroup		<Control8>	Sales Orders Released Not Shipp...
		Field	"Ready to Ship"	ReadyToShip	Ready To Ship
		Field	"Partially Shipped"	PartiallyShipped	Partially Shipped
		Field	Delayed	DelayedOrders	Delayed
		Field	"Average Days Delayed"	<Average Days ...	<Average Days Delayed>

These two CueGroups are shown displayed in the following Role Center screenshot:



- **FixedLayout**: This **GroupType** is used at the bottom of List pages, following a Repeater group. The **FixedLayout** group typically contains totals or additional line-related detail fields. Many Journal pages, such as **Page 39 - General Journal**, **Page 40 - Item Journal**, and **Page 201 - Job Journal** have **FixedLayout** groups. The **Item JournalFixedLayout** group only shows the **Item Description**, which is also available in a Repeater column, but can easily display other fields as well. A **FixedLayout** group can also display a lookup or calculated value, like many of the Statistics pages, for example, **Page 151 - Customer Statistics** and **Page 152 - Vendor Statistics**.
- **GridLayout**: This **GroupType** provides additional formatting capabilities to layout fields, row by row, column by column, spanning rows or columns, and hiding or showing captions. **Page 970 - Time Sheet Allocation** contains an example of **GridLayout** use. To learn more about **GridLayout** use, search Help for **Gridlayout**.

- **IndentationColumnName** and **IndentationControls**: These properties allow a group to be defined in which fields will be indented, as shown in the following screenshot of the **Chart of 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		Type to filter (F3)	No.		
No.	Name	Income/B...	Account Subcategory	Account Type	Total
1000	<b>BALANCE SHEET</b>	Balance Sh...		Heading	
1002	<b>ASSETS</b>	Balance Sh...	Assets	Begin-Total	
1003	<b>Fixed Assets</b>	Balance Sh...	Equipment	Begin-Total	
1005	<b>Tangible Fixed Assets</b>	Balance Sh...	Equipment	Begin-Total	
1100	<b>Land and Buildings</b>	Balance Sh...	Equipment	Begin-Total	
1110	Land and Buildings	Balance Sh...	Equipment	Posting	
1120	Increases during the Year	Balance Sh...	Equipment	Posting	
1130	Decreases during the Year	Balance Sh...	Equipment	Posting	
1140	Accum. Depreciation, Buildings	Balance Sh...	Accumulated Depreciation	Posting	
1190	<b>Land and Buildings, Total</b>	Balance Sh...	Equipment	End-Total	1100.1
1200	<b>Operating Equipment</b>	Balance Sh...	Equipment	Begin-Total	
1210	Operating Equipment	Balance Sh...	Equipment	Posting	
1220	Increases during the Year	Balance Sh...	Equipment	Posting	

- **FreezeColumnID**: This freezes the identified column, and all of the columns to the left, so 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. Users can also freeze columns as part of personalization.
- **ShowAsTree**: This works together with the indentation property. **ShowAsTree** allows an indentation set of rows to be expanded or collapsed dynamically by the user for easier viewing. Example pages that use this property are Page 583 - XBRL Taxonomy Lines, Page 634 - Chart of Accounts Overview, and Page 5522 - Order Planning.

## Field controls

All field controls appear in the same format in **Page Designer**. The **SubType** column is empty and the **SourceExpr** column contains the data expression that will be displayed.

All the field control properties are listed for each field, but individual 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 following is a split screenshot of the properties for field controls:

Property	Value
BlankZero	<No>
AutoFormatType	<0>
AutoFormatExpr	<>
QuickEntry	<TRUE>
AccessByPermission	<Undefined>
ApplicationArea	<>
SourceExpr	"No."
TableRelation	<Undefined>
Importance	<Standard>
CaptionClass	<>
DrillDownPageID	<Undefined>
LookupPageID	<Undefined>
Lookup	<Undefined>
DrillDown	<Undefined>
AssistEdit	<Undefined>
ClosingDates	<No>
Numeric	<No>
RowSpan	<Undefined>
ColumnSpan	<Undefined>
DateFormula	<No>
ControlAddIn	<>
Style	<None>
StyleExpr	<FALSE>
ExtendedDatatype	<None>
Image	<Stack>

We'll review the field control properties that are more frequently used or that are more significant in terms of effect as follows:

- **Visible, Enabled, and Editable:** These have the same functionality as the identically named group controls, but they 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 over (override) other settings in a page, control group, or control.
- **HideValue:** This allows the value of a field to be optionally displayed or hidden, based on an expression that evaluates to TRUE or FALSE.
- **CaptionandCaptionML:** These define the caption that will be displayed for this field (in English or the current system language if not English).
- **ShowCaption:** This is set to Yes or No; it determines whether or not the caption is displayed.
- **MultiLine:** This must be set to TRUE if the field is to display multiple lines of text.
- **OptionCaption and OptionCaptionML:** These set the text string options that are displayed to the user. 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:** This 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.
- **Width:** This allows setting a specific field display width - number of characters. Especially useful for Control SubType of GridLayout.
- **ShowMandatory:** This shows a red asterisk in the field display to indicate a required (mandatory) data field. **ShowMandatory** can be based on an expression that evaluates to TRUE or FALSE. This property does not enforce any validation of the field. Validation is left to the developer.

- **ApplicationArea:** This allows us to hide controls based on the **Application Area** feature, which was introduced in NAV2017.



This blog entry explains how the Application Area property is implemented by Microsoft. You can check it out by clicking on the following link:

<https://markbrummel.wordpress.com/2016/11/19/nav2017-applicationarea-the-mystery-of-disappearing-controls/>

- **QuickEntry:** This allows the field to optionally receive focus or be skipped, based on an expression that evaluates to TRUE or FALSE.
- **AccessByPermission:** This determines the permission mask required for a user to view or access this field.
- **Importance:** This controls the display of a field. This property only applies to individual (non-repeating) fields located within a FastTabs. Importance can be set to **Standard** (the default), **Promoted**, or **Additional**, which are briefly described here:
  - **Standard:** This is the normal display. Implementations of the rendering routines for different targets may utilize this differently.
  - **Promoted:** If the property is set to **Promoted** and the page is on a collapsed FastTab, then the field contents will be displayed on the FastTab line. If FastTab is expanded, the field will display normally.
  - **Additional:** If the property is set to **Additional** and FastTab is collapsed, there is no effect on the display. If 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 FastTab.
- **RowSpan** and **ColumnSpan:** These are used in conjunction with the GridLayout controls as layout parameters.
- **ControlAddIn:** When the field represents a control add-in, this contains the name and public token key of the control add-in.

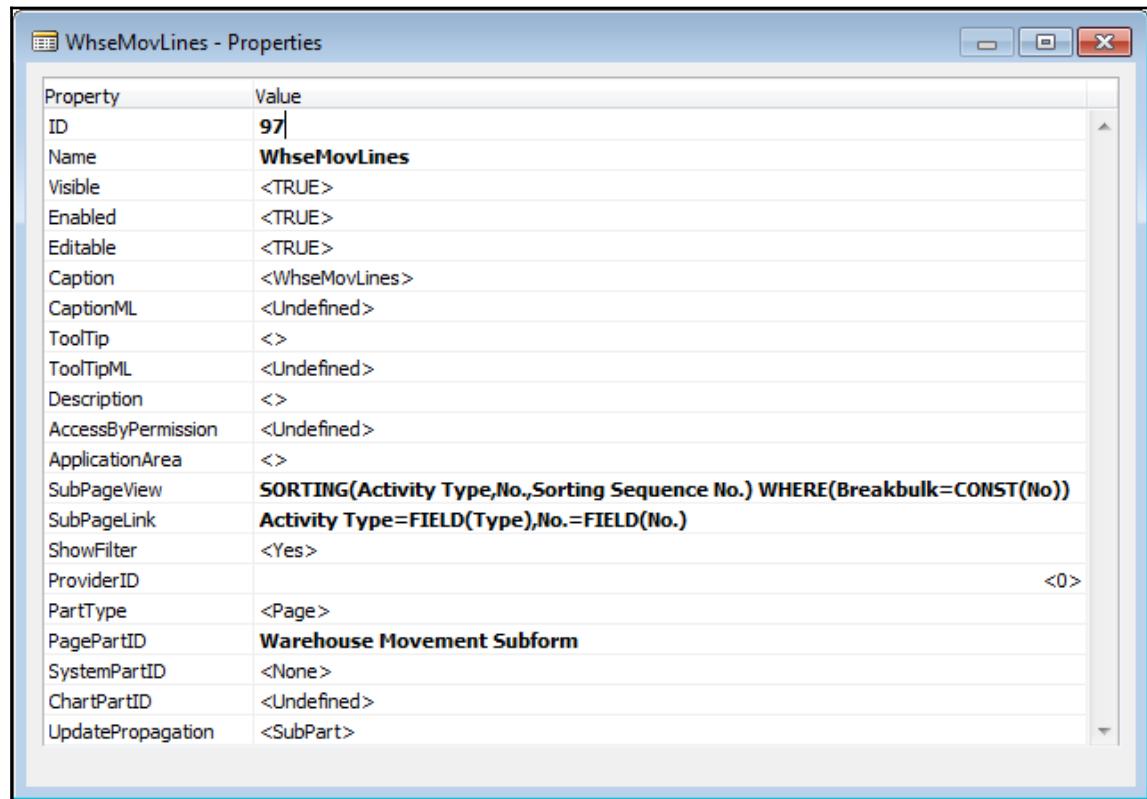
- **ExtendedDatatype:** This allows a text field to be categorized as a special data type. The default value is **None**. If **ExtendedDatatype** is selected, it can be any one of the following data types:
  - **Phone No.:** This displays a regionally appropriate phone number format
  - **URL:** This displays a formatted URL
  - **E-MailFilter:** This is used on reports
  - **Ratio:** This displays a progress bar
  - **Masked:** This fills the field with bold dots 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 set with **Phone No.**, **URL**, or **E-Mail** data, 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 e-mail client. Setting **ExtendedDatatype** will also define the validation that will automatically be applied to the field.
  - **Person:** This allows rendering an image as a person in the Web Client.
  - **Resource:** (Not used, according to Microsoft).
- **Image:** This allows the display of an image on a Cue for a Field control in a **CueGroup** control. It only applies to a **CueControl** field of an integer data type. If no image is wanted, choose a value of **None**.

## Page Part controls

**Page Parts** are used for FactBoxes and SubPages. Many of the properties of Page Parts are similar to the properties of other NAV components and operate essentially the same way in a Page Part as they operate elsewhere. Those properties include **ID**, **Name**, **Visible**, **Enabled**, **Editable**, **Caption**, **CaptionML**, **ToolTip**, **ToolTipML**, and **Description**.

Other properties that are specific to the Page Part controls are as follows:

- **SubPageView:** This defines the table view that applies to the named subpage as shown in the following screenshot of a Part in Page 7135 - Warehouse Movement:



- **SubPageLink:** This defines the fields that link to the subpage and the link that is based on a constant, a filter, or another field. This is also shown in the preceding screenshot.
- **ProviderID:** This contains the ID of another Page Part control within the current page. This enables us to link a subordinate part to a controlling parent part. For example, **Page 42 - Sales Order** uses this property to update **Sales Line FactBox** by defining a **ProviderID** link from FactBox to the **Sales Lines** FastTab. Other pages with similar links include **Page 41 - Sales Quote** and Pages 43, 44, 50, 507, and 5768. In the following screenshot, we can see the **Sales Lines** PagePart (Control ID 58) linked to **Sales LineFactbox** by means of the **ProviderID** value of 58:

The screenshot shows two separate instances of the Microsoft Dynamics NAV Page Designer interface.

**Top Window:** Page 42 Sales Order - Page Designer

E.. Type	SubType	SourceExpr	Name	Caption
Part	Page		SalesLines	<SalesLines>
<b>Group</b>	<b>Group</b>			I-to Customer No. > I-to Contact No. > I-to Name > I-to Address > I-to Address 2 > I-to Post Code > I-to City > I-to Contact >
Field				

**Properties Dialog (SalesLines - Properties):**

Property	Value
ID	58
Name	SalesLines
Visible	<TRUE>
Enabled	<TRUE>
Editable	DynamicEditable
Caption	<SalesLines>

**Bottom Window:** Page 42 Sales Order - Page Designer

E.. Type	SubType	SourceExpr	Name	Caption
Field		"Prepmt. Pmt. Discount ...	<Prepmt. Pmt. Discount D...	<Prepmt. Pmt. Discount D...
<b>Container</b>	<b>FactBoxArea</b>		<Control190000...>	<Control1900000007>
Part	Page		<Sales Hist. Sell-...	<Sales Hist. Sell-to FactBox>
Part	Page		<Customer Stati...	<Customer Statistics FactBox>
Part	Page		<Customer Deta...	<Customer Details FactBox>
Part	Page		<Sales Line Fact...	<Sales Line FactBox>
Part	Page		<Item Invoicing FactBox>	<Item Invoicing FactBox>
Part	Page		<Approval FactBox>	<Approval FactBox>
Part	Page		<Resource Details FactBox>	<Resource Details FactBox>
Part	Page		<Item Warehouse FactBox>	<Item Warehouse FactBox>

**Properties Dialog (<Sales Line FactBox> - Proper...):**

Property	Value
ID	1906127307
Name	<Sales Line FactBox>
Visible	TRUE
Enabled	<TRUE>
Editable	<TRUE>
Caption	<Sales Line FactBox>
CaptionML	<Undefined>
ToolTip	<>
ToolTipML	<Undefined>
Description	<>
AccessByPermission	<Undefined>
SubPageView	<Undefined>
SubPageLink	<Undefined>
ShowFilter	<Yes>
ProviderID	58
PartType	Page
PagePartID	Sales Line FactBox
SystemPartID	<None>

- **PartType:** This defines the type of part to be displayed in a FactBox. There are three options and each option also requires another related property to be defined as follows:

PartType option	Required property
Page	PagePartID
System	SystemPartID
Chart	ChartPartID

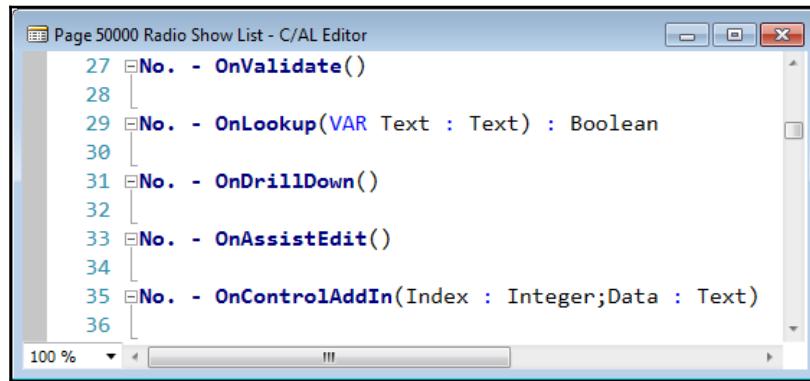
- **PagePartID:** This must contain the page object ID of a FactBox part if **PartTypeOption** is set to Page.
- **SystemPartID:** This must contain the name of a predefined system part if **PartTypeOption** is set to System. The available choices are: Outlook, Notes, MyNotes, and RecordLinks.
- **ChartPartID:** This must contain a Chart ID if **PartTypeOption** is set to Chart. The Chart ID is a link to the selected entry in the Chart table (Table number 2000000078).
- **UpdatePropagation:** This allows us to update the parent page from the child (subordinate) page. A value of **Subpage** updates the subpage only. A value of both will cause the parent page to be updated and refreshed at the same time as the subpage.



The NAV 2017 Chart Control Add-in provides significant additional charting capability. Information can be found in the **Help** section, *Displaying Charts Using the Chart Control Add-in* ([https://msdn.microsoft.com/en-us/library/hh167009\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/hh167009(v=nav.90).aspx)).

## Page Control triggers

There are five triggers for each Field control. The Container, Group, and Part controls do not have associated triggers. The following screenshot shows the Page Control triggers:



The screenshot shows the C/AL Editor window titled "Page 50000 Radio Show List - C/AL Editor". The code pane displays the following triggers:

```
27  No. - OnValidate()
28
29  No. - OnLookup(VAR Text : Text) : Boolean
30
31  No. - OnDrillDown()
32
33  No. - OnAssistEdit()
34
35  No. - OnControlAddIn(Index : Integer;Data : Text)
36
```

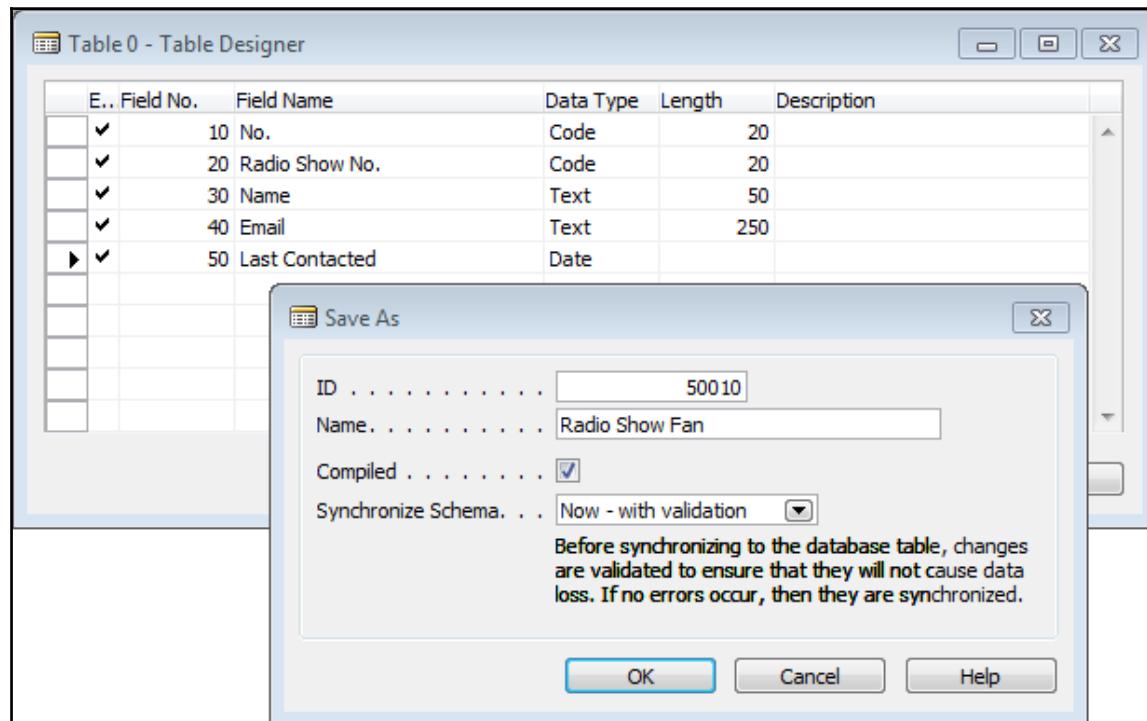
The guideline for the use of these triggers is the same as the guideline for Page triggers--if there is a choice, don't put C/AL code in a Control trigger. Not only will this make our code easier to upgrade in the future, but it will also make it easier to debug, and it will be easier for the developer following us to decipher our changes.

## Bound and Unbound Pages

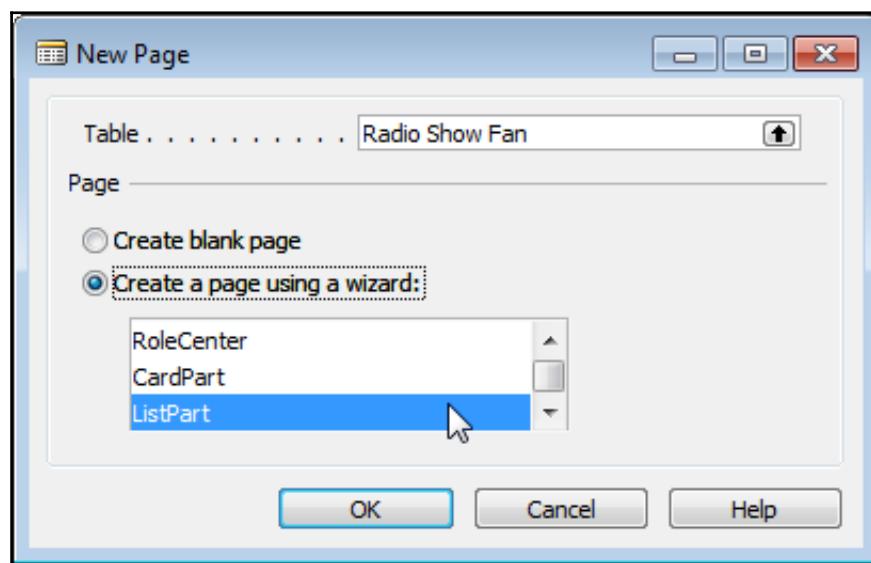
Pages can be created as **bound** (associated with a specific table) or **unbound** (not specifically associated with any table). Typically, a Card or List page will be bound, but the Role Center pages will be unbound. Other instances of unbound pages are rare. Unbound pages may be used to communicate status information or initiate a process. Examples of unbound pages are Page 476 - Copy Tax Setup and Page 1040 - Copy Job, both of which have a PageType property of StandardDialog.

## WDTU Page Enhancement - part 2

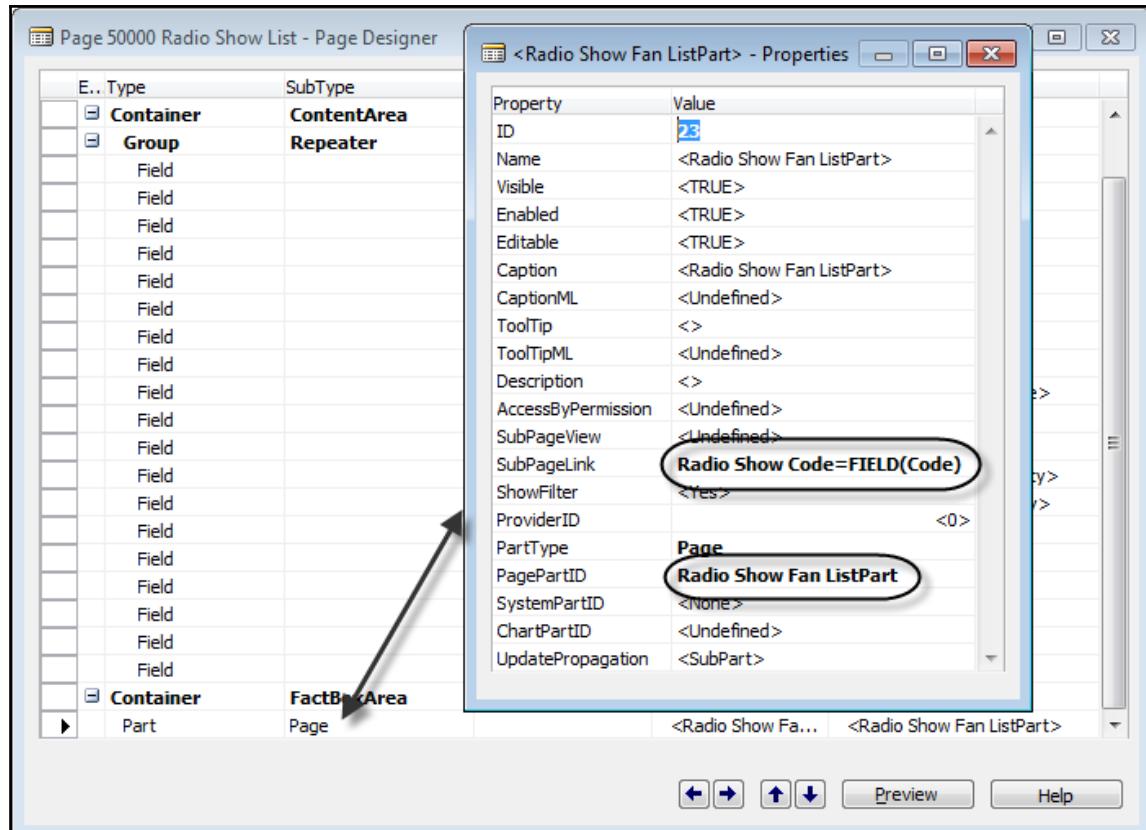
Now that we have an additional understanding of page structures, let's do a little more enhancing of our WDTU application pages. We've decided that it will be useful to keep track of specific listener contacts, a fan list. First, we will need to create a table of Fan information that we will save as Table with ID as 50010 and name it Radio Show Fan, which will look like the following screenshot:



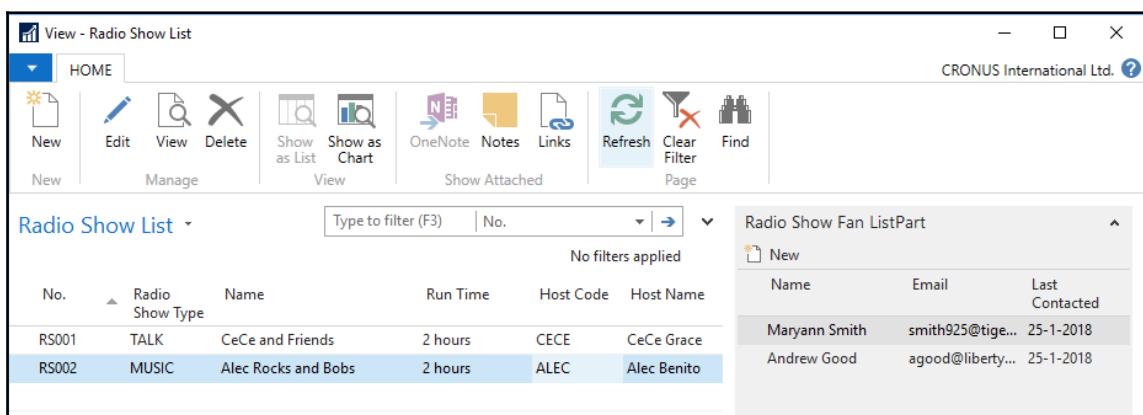
We want to be able to review the Fan list as we scan the list, Radio Show List. This requires adding a FactBox area to Page 50000. In turn, that requires a Page Part that will be displayed in the FactBox. The logical sequence is to create the Page Part first, then add the FactBox to Page 50000. Because we just want a simple ListPart with three columns, we can use the Page Wizard to create our Page Part, which we will save as Page 50080 - Radio Show Fan ListPart, including just the Name, Email, and Last Contacted fields. The following screenshot shows the **New Page** wizard:



Next, we will use the Page Designer to add a FactBox area to Page 50000, populate the FactBox area with our PagePart - Page 50080, and set the properties for the Page Part to link to the highlighted record in the **Radio Show List** page as shown in the following screenshot:

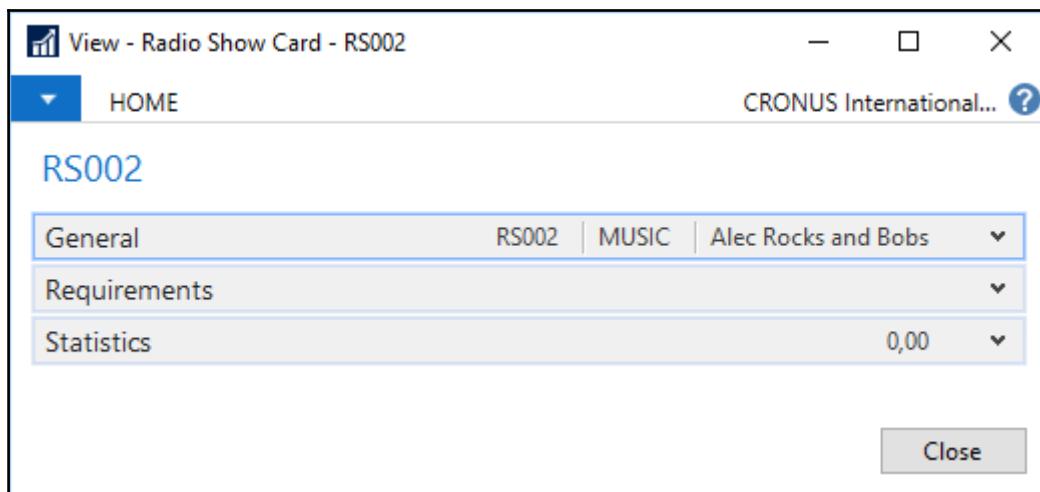


If we Run Table 50010 and insert a few test records first, and then Run Page 50000, we should see something like the following screenshot:



Before finishing this part of our enhancement effort, we will create a List page that we can use to view and maintain the data in Table 50010 in the future (assign it as Page 50009 - Radio Fan List).

One other enhancement we can do now is promote some fields in **Radio Show Card** so they can be seen when FastTabs are collapsed. All we have to do is choose the fields we want to promote and then change the page control property of **Importance** to **Promoted**. If we chose the fields and promote No., Type, Description, and Avg. Listener Share, our card with collapsed FastTabs will look like the following screenshot; we don't have any Listener Share data yet:



## Page Actions

Actions are the menu items of NAV 2017. **Action** menus can be found in several locations. The primary location is the Ribbon appearing at the top of most pages. Other locations for actions are the Navigation Pane, Role Center, CueGroups, and the **Action** menu on FactBox page parts.

**Action Designer**, where actions are defined, is accessed from the **Page Designer** form by clicking on **View** and selecting **PageActions** or **ControlActions** (Control Actions can only be used for Role Center, CueGroup actions, and for Navigate Page wizard actions). When we click on **PageActions** or **Ctrl + Alt + F4** for the Fixed Asset page (Page 5600), we will see a list of Ribbon actions, which are shown in the following screenshot as they appear in **Action Designer**:

The screenshot shows the 'Page - Action Designer' window. It displays a tree view of actions categorized by type and subtype, along with their names, captions, and descriptions. The categories include ActionContainer, ActionGroup, and ActionItems. The 'Caption' column provides a brief description of each action, such as 'Fixed &Asset' for the first action under 'ActionContainer'. The 'Description' column provides more detailed information, like 'Depreciation &Books' for the first action under 'ActionContainer'.

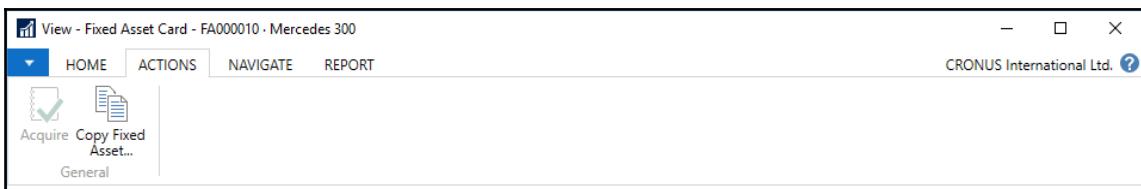
E.. Type	SubType	Name	Caption
ActionContainer	RelatedInformation	<Action190000003>	<Action190000000...>
ActionGroup		<Action47>	<b>Fixed &amp;Asset</b>
Action		<Page FA Depreciation Books>	Depreciation &Books
Action		<Page Fixed Asset Statistics>	Statistics
Action		<Page Default Dimensions>	Dimensions
Action		<Page Maintenance Registration>	Maintenance &Re...
Action		<Page Fixed Asset Picture>	Picture
Action		<Page FA Posting Types Overview>	FA Posting Types ...
Action		<Page Comment Sheet>	Co&ments
ActionGroup		<Action3>	<b>Main Asset</b>
Action		<Page Main Asset Components>	Main Asset Comp...
Action		<Page Main Asset Statistics>	Ma&n Asset Statis...
Separator			-----
ActionGroup		<Action5>	<b>Insurance</b>
Action		<Page Total Value Insured>	Total Value Insured
ActionGroup		<Action11>	<b>History</b>
Action		<Page FA Ledger Entries>	Ledger E&ntries
Action		<Page FA Error Ledger Entries>	Error Ledger Entries
Action		<Page Maintenance Ledger Entries>	Main&tenance Led...
ActionContainer	ActionItems	<Action190000004>	<Action190000000...>
Action		Acquire	Acquire
Action		<Action57>	C&opy Fixed Asset
ActionContainer	Reports	<Action190000006>	<Action190000000...>
Action		<Report Fixed Asset - Details>	Details
Action		<Report Fixed Asset - Book Value>	FA Book Value
Action		<Report Fixed Asset - Book Value>	FA Book Val. - Appr....
Action		<Report Fixed Asset - Analysis>	Analysis
Action		<Report Fixed Asset - Projected>	Projected Value
Action		<Report Fixed Asset - G/L Analys>	G/L Analysis
Action		<Report Fixed Asset Register>	Register

The associated Ribbon tabs for the preceding Page Action list are as follows:

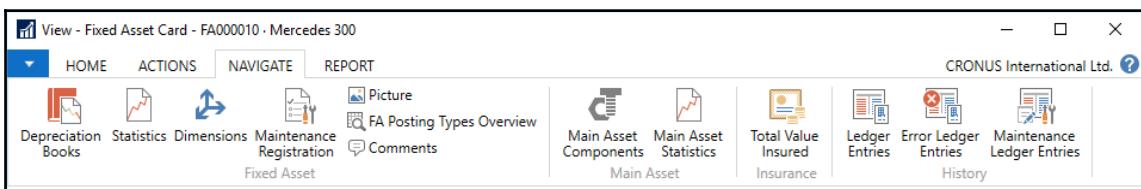
- First, the **HOME** tab, as you can see in the following screenshot:

The screenshot shows the 'View - Fixed Asset Card - FA000010 - Mercedes 300' window. The ribbon at the top has tabs for HOME, ACTIONS, NAVIGATE, and REPORT. The HOME tab is selected. Below the ribbon are several groups of icons: Manage (View, New, Delete), Process (Statistics, Dimensions, Maintenance Registration), Report (Details, Analysis, Projected Value), Show Attached (OneNote, Notes, Links), and Navigation (Refresh, Clear Filter, Go to, Previous, Next, Page).

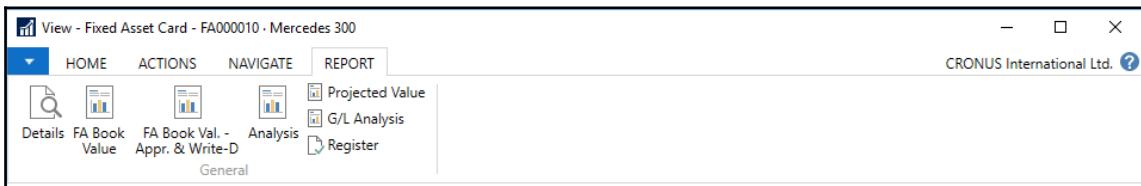
- Second, the **ACTIONS** tab shown in this screenshot:



- Third, the **NAVIGATE** tab, as shown in the following screenshot:



- Finally, the **REPORT** tab, as shown here:



There are two default Ribbon tabs created for every Ribbon: **HOME** and **ACTIONS**. What actions appear by default is dependent on the Page Type.

Actions defined by the developer appear on a Ribbon tab and tab submenu section based on a combination of the location of the action in the Page Actions structure and on the property settings of the individual action. There are a lot of possibilities, so it is important to follow some basic guidelines, some of which are listed here:

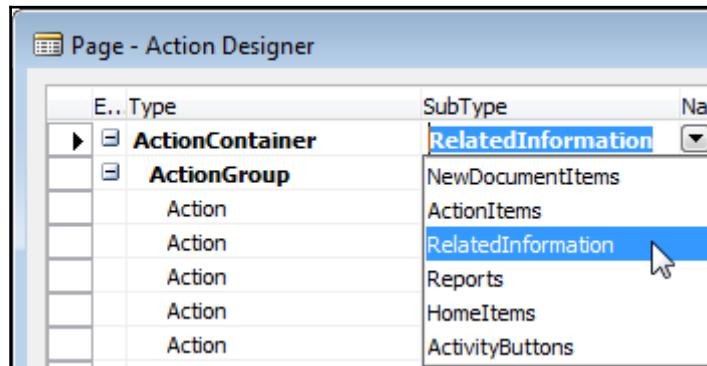
- Maintain the look and feel of the standard product wherever feasible and appropriate
- Put the actions on the **HOME** tab that are expected to be used most
- Be consistent in organizing actions from tab to tab and page to page
- Provide the user with a complete set of action tools, but don't provide so many options that it's hard to figure out which one to use

## Page Action types and subtypes

Page Action entries can have one of four types: **ActionContainer**, **ActionGroup**, **Action**, or **Separator**. At this time, separators don't seem to have any effect in the rendered pages. A Page Action type uses the following indented hierarchical structure:

Action types	Description
ActionContainer	Primary Action grouping
ActionGroup	Secondary Action grouping
Action	Action
ActionGroup	Secondary groups can be set up within an Action list for drop-down menus of Actions (a tertiary level)
Action	The indentation indicates this is part of a drop-down menu
Separator	
Action	Action
Separator	
ActionGroup	Back to the Secondary grouping level
ActionContainer	Back to the Primary grouping level

An **ActionContainer** action line type can have one of six **SubType** values, as shown in the following screenshot. Subtypes of **HomeItems** and **ActivityButtons** only apply to Role Center pages:

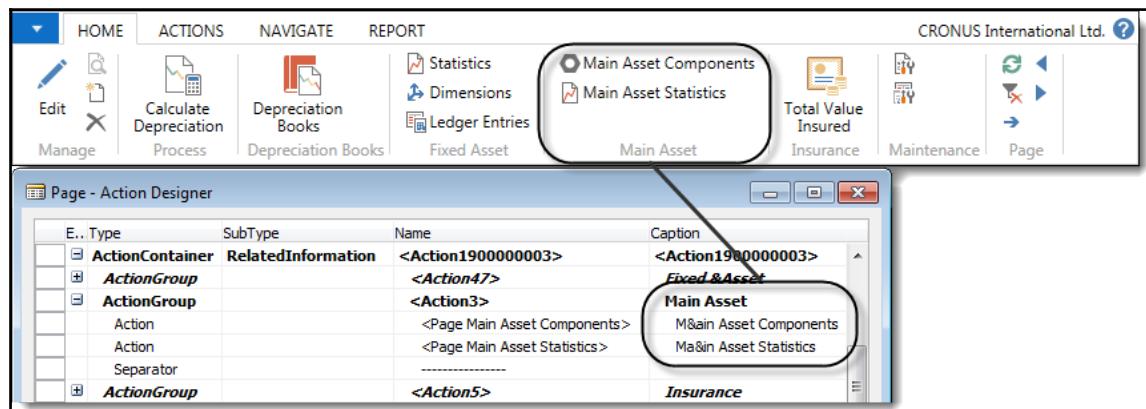


Those Action options display in different sections of the Ribbon as follows:

- Actions in **ReportsSubType** will appear on the Ribbon **REPORT** tab
- Actions in **RelatedInformationSubType** will appear on the Ribbon **NAVIGATE** tab
- Actions in **ActionItemsSubType** will appear on the Ribbon **ACTIONS** tab
- Actions in **NewDocumentItemsSubType** will appear on the Ribbon in a **New Documents** submenu section on the **ACTIONS** tab

## Action Groups

Action Groups provide a submenu grouping of actions within the assigned tab. In the following screenshot, the Page Preview is highlighting the submenu **Main Asset**, which is the Caption for the associated **ActionGroup**. In **RelatedInformation ActionContainer**, we can see the other **ActionGroup**, **Fixed Asset**, **Insurance**, and **History** matching the submenu groups on the ribbon's **NAVIGATE** tab:



The following are the **ActionContainer** properties:

- **ID**: This is the automatically assigned unique object number of the action
- **Name and Caption**: These are displayed in Action Designer
- **CaptionML**: This is the action name displayed, depending on the language option in use
- **Description**: This is for internal documentation

The following are the **ActionGroup** properties:

- **ID**: This is the automatically assigned unique object number of the action.
- **Name and Caption**: These are displayed in the **Action Designer**.
- **Visible** and **Enabled**: These are TRUE or FALSE, defaulting to TRUE. These properties can be assigned Boolean expressions, which can be evaluated during processing.
- **CaptionML**: This is the action name displayed, depending on the language option in use.
- **ToolTip** and **ToolTipML**: This is for helpful display to the user
- **Description**: This is for internal documentation.

- **Image:** This can be used to assign an icon to be displayed. The icon source is the Activity Button Icon Library, which can be viewed in detail in the Developer and IT Pro Help. Here is a screenshot of the Properties for an action from Page 5600 - Fixed Asset Card followed by explanations of those properties:

<Page FA Depreciation Books> - Properties	
Property	Value
ID	51
Name	<Page FA Depreciation Books>
Visible	<TRUE>
Enabled	<TRUE>
RunPageMode	<Edit>
Caption	Deprecation &Books
CaptionML	<b>ENU=Deprecation &amp;Books</b>
ToolTip	View or edit the depreciation book or books that must...
ToolTipML	<b>ENU=View or edit the depreciation book or bo...</b>
Description	<>
AccessByPermission	<Undefined>
ApplicationArea	<b>#FixedAssets</b>
Image	<b>DepreciationBooks</b>
Promoted	<No>
PromotedCategory	<New>
PromotedIsBig	<No>
PromotedOnly	<No>
Scope	<Page>
Ellipsis	<No>
ShortCutKey	<>
RunObject	<b>Page FA Depreciation Books</b>
RunPageView	<Undefined>
RunPageLink	<b>FA No.=FIELD(No.)</b>
RunPageOnRec	<No>
InFooterBar	<No>
Gesture	<None>

The following are the Action properties shown in the preceding screenshot:

- **ID:** This is the automatically assigned unique object number of the action.
- **Name and Caption:** These are displayed in the Action Designer.
- **CaptionML:** This is the action name displayed, depending on the language option in use.
- **Visible and Enabled:** These are TRUE or FALSE, defaulting to TRUE. These properties can be assigned Boolean expressions, which can be evaluated during processing.
- **RunPageMode:** This can be View (no modification), Edit (the default), or Create (New).
- **ToolTip and ToolTipML:** These are for helpful display to the user.
- **Description:** This is for internal documentation.
- **ApplicationArea:** This allows us to hide controls based on the **Application Area** feature introduced in NAV2017.
- **Image:** This can be used to assign an icon to be displayed. The icon source is the Action Icon Library, which can be viewed in detail in the *Developer and IT Pro Help* at <https://msdn.microsoft.com/en-us/dynamics-nav/action-icon-library>.
- **Promoted:** If Yes, show this action in the ribbon **Home** tab.
- **PromotedCategory:** If **Promoted** is Yes, this defines the category in the **Home** tab in which to display this action.
- **PromotedIsBig:** If **Promoted** is Yes, this indicates if the icon is to be large (Yes) or small (No - the default).
- **Ellipsis:** If Yes, this displays an ellipsis after the caption.
- **ShortCutKey:** This provides a shortcut key combination for this action.
- **RunObject:** This defines what object to run to accomplish the action.
- **RunPageView:** This defines the table view for the page being run.
- **RunPageLink:** This defines the field link for the object being run.
- **RunPageOnRec:** This defines a linkage for the run object to the current record.
- **InFooterBar:** This places the action icon in the page footer bar. This only works on pages with a PageType of NavigatePage.
- **Gesture:** This allows an action to be executed, swiping left or right using the universal client.

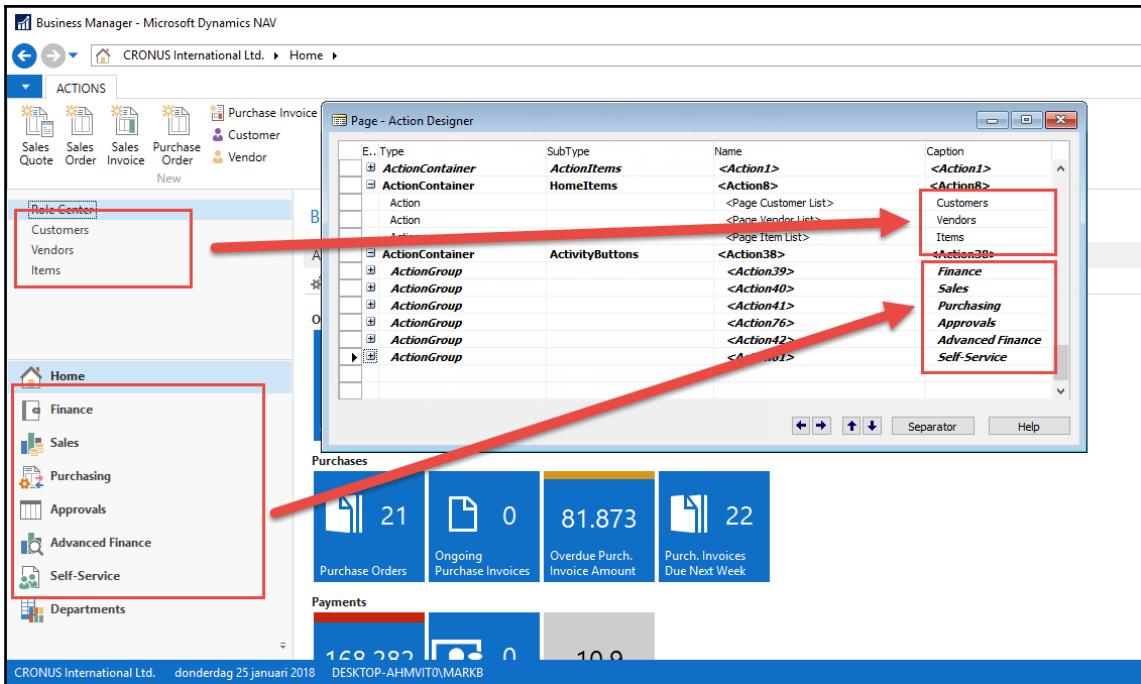
To summarize a common design choice, individual Ribbon actions can be promoted to the ribbon **Home** tab based on two settings. First, set the **Promoted** property to **Yes**. Second, set the **PromotedCategory** property to define the category on the **Home** tab where the action is to be displayed. This promotion results in the action appearing twice in the ribbon, once where defined in the action structure hierarchy and once on the **Home** tab. See the preceding screenshot for example settings assigning the **Depreciation Books** action to the **Process** category in addition to appearing in the **Fixed Assets** category on the **Navigation** tab.

## Navigation Pane Button actions

In the **Navigation** pane, on the left side of the Windows Client display, there is a **Home** button where actions can be assigned as part of the Role Center Page definition. The **Navigation** pane definition is part of the Role Center. When defining the actions in a Role Center page, we can include a group of actions in an **ActionContainer** group with **SubType** of **HomeItems**. These actions will be displayed in the **Home** button menu on the **Navigation** pane.

Additional **Navigation** pane buttons can also easily be defined in a Role Center page action list. First, define an **ActionContainer** with **SubType** of **ActivityButtons**. Each **ActionGroup** defined within this **ActionContainer** will define a new Navigation Pane's **Activity** button. The following screenshot is a combination image showing the RTC for the Sales Order Processor Role Center on the left (focused on the Navigation Pane), and showing the **Action Designer** contents that define the **Home** and **Posted Documents** buttons on the right.

The actions showing on the right that aren't visible in the **Navigation** pane on the left are in submenus indicated by the small outline arrowheads to the far left of several entries, including **Sales Order** and **Sales Quotes**. NAV will automatically do some of this grouping for us based on the pages referenced by the actions, including CueGroup Actions:



## Actions Summary

The primary location where each user's job role based actions should appear is the **Navigation** pane. The Role Center action list provides detailed action menus for the **Home** button and any appropriate additional **Navigation** pane button. Detailed page/task specific actions should be located in the Ribbon at the top of each page.

As mentioned earlier, a key design criterion for the NAV Windows Client is for a user to have access to the actions they need to get their job done; in other words, to tailor the system to the individual users' roles. Our job as developers is to take full advantage of all of these options and make life easier for the user. In general, it's better to go overboard in providing access to useful capabilities than to make 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 the important user tools no more than one click away.

## Learning more

The following section gives a description of several excellent ways to learn more about pages.

## Patterns and creative plagiarism

When we want to create new functionality, the first task is to obviously create functional specifications. Once those are in hand, we should look for guidelines to follow. Some of the sources that are readily available are listed here:

- The NAV Design Patterns Wiki  
(<https://community.dynamics.com/nav/w/designpatterns/default.aspx>)
- C/AL Coding Guidelines, as used internally by Microsoft in the development of NAV application functionality  
(<https://community.dynamics.com/nav/w/designpatterns/156.cal-coding-guidelines.aspx>)
- Blogs and other materials available in the Microsoft Dynamics NAV Community  
(<https://community.dynamics.com/nav/default.aspx>)
- Design Patterns on Mark Brummel's blog  
(<https://markbrummel.wordpress.com/category/nav-architecture-patterns-code/design-patterns/>)
- The NAV system itself as distributed by Microsoft

It's always good to start with an existing pattern or object that has capabilities similar to our requirements, and study the existing logic and the code. In many lines of work, the term *plagiarism* is a nasty term. However, when it comes to modifying an existing system, plagiarism is a very effective research and design tool. This approach allows us to build on the hard work of the many skilled and knowledgeable people who have contributed to the NAV product. In addition, this is working software. This eliminates at least some of the errors we would make if starting from scratch.



The book, *Learning Dynamics NAV Patterns*, explains in depth how to implement Design Patterns in Dynamics NAV 2017  
(<https://www.packtpub.com/big-data-and-business-intelligence/learning-dynamics-nav-patterns>).

When designing modifications for NAV, studying how the existing objects work and interact is often the fastest way to create new working models. We should allocate some time, both for studying the material in the NAV Design Patterns Wiki and for exploring the NAV Cronus demo system.

Search through the **Cronus demonstration system** (or an available production system) in order to find one or more pages that have the feature we want to emulate (or a similar feature). If there are both complex and simple instances of pages that contain this feature, we should concentrate our research on the simple instance first. Make a test copy of the page. Read the code. Use the Page Preview feature. Run the page. Make a minor modification. Preview again; then run it again. Continue this until our ability to predict the results of changes eliminates surprises or confusion.

## Experimenting on your own

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 pages in NAV 2017.

Don't forget to make liberal use of the Help information while you are experimenting. A majority of the available detailed NAV 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 built into the NAV help system. The product team responsible for Help pays close attention to the feedback they receive and use it to improve the product. Thus, we will all benefit from your feedback.

## Experimentation

Start with the blank slate approach because that allows focus on specific features and functions. Because we've already gone through the mechanical procedures of creating new pages of the card and list types and using the Page Designer to add controls and modify control properties, we won't detail those steps here. However, as you move the focus for experimentation from one feature to another, you may want to review what was covered in this chapter.

Let's walk through some examples of experiments you could do now, then build on as you get more adventuresome. Each of the objects you create at this point should be assigned into an object number range that you are reserving for testing. Follow these steps:

1. Create a new Table 50050 (try using 50009 if your license won't allow 50050). Do this by opening Table 50004 in Table Designer, and then save it as 50050 with the name **Playlist Item Rate Test**.
2. Enter a few test records into Table 50050, Playlist Item Rate Test. This can be done by highlighting the table, then clicking on **Run**.
3. Create a list page for Table 50050 with at least three or four fields.
4. Change the **Visible** property of a field by setting it to **False**.
5. Save and run the page.
6. Confirm that the page looks as what was expected. Go into **Edit** mode on the page. See if the field is still invisible.
7. Use the page **Customization** feature (from the drop-down icon on the upper-left corner of the page) in order to add the invisible field; also remove a field that was previously visible. Exit customization. View the page in various modes, such as **View**, **Edit**, **New**, and so on.
8. Go back into the Page Designer and design the page again.
9. One or two at a time, experiment with setting the **Editable**, **Caption**, **ToolTip**, and other control properties.
10. Don't just focus on text fields. Experiment with other data types as well. Create a text field that is 200 characters long. Try out the **MultiLine** property.
11. After you get comfortable with the effect of changing individual properties, try changing multiple properties to see how they interact.

When you feel 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 the test 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 ones which do not override them.

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 Chart Parts as well. There is a wealth of prebuilt parts that come with the system. Even if the parts that are supplied aren't exactly right for the 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 that you have a grasp of the different types of controls in the context of cards and lists, consider checking 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 ListPlus, List Parts, Card Parts, and, to a lesser extent, even Document pages.

You may now decide to learn by studying samples of the page objects that are part of the standard product. You could start by copying an object, such as Page 22 - Customer List, to another object number in your testing range and then begin to analyze how it is put together and how it operates. Again, you should 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. Also, it's a good idea to back up your work one way or another before making additional changes. An easy way to back up individual objects is to highlight the object, then export it into a .fob file (**File | Export**). The restore method is the reverse, importing of the fob file.

Another excellent learning option is to choose one of the patterns that has a relationship with the area about which you want more knowledge. If, for example, you are going to create an application that has a new type of document, such as a Radio Program Schedule, you should study the **Document Pattern**. You might also want to study **Create Data from Templates Pattern**. At this point, it has become obvious that there are a variety of sources and approaches to supplement the material in this text.

## Review questions

1. Once a Page has been developed using the Page Wizard, the developer has very little flexibility in the layout of the Page. True or False?
2. Actions appear on the Role Center screen in several places. Choose two:
  - a) Address Bar
  - b) Ribbon
  - c) Filter Pane
  - d) Navigation Pane
  - e) Command Bar
3. A user can choose their Role Center when they login. True or False?
4. An Action can only appear in one place in the Ribbon or in the Navigator Pane. True or False?
5. When developing a new page, the following Page Part types are available. Choose two:
  - a) Chart part
  - b) Map part
  - c) Social part
  - d) System part
6. All page design and development is done within the C/SIDE Page Designer. True or False?
7. Document pages are for word processing. True or False?
8. Two Activity Buttons are always present in the Navigation Pane. Which of the following two are those?
  - a) Posted Documents
  - b) Departments
  - c) Financial Management
  - d) Home

9. The Filter Pane includes "Show results - Where" and "Limit totals to" options. True or False?
10. C/AL code place in pages should only be to control display characteristics, not to modify data. True or False?
11. 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?
12. Which of the following are true about the control property Importance? Choose two:
  - a) Applies only to Card and CardPart pages
  - b) Can affect FastTab displays
  - c) Has three possible values: Standard, Promoted, and Additional
  - d) Applies to Decimal fields only
13. FactBoxes are delivered as part of the standard product. They cannot be modified nor can new FactBoxes be created. True or False?
14. RTC Navigation Pane entries always invoke which one of the following page types?
  - a) Card
  - b) Document
  - c) List
  - d) Journal/Worksheet
15. The Page Preview tool can be used as a drag and drop page layout design tool. True or False?

16. Some field control properties can be changed dynamically as the object executes.  
Which ones are they? Choose three:
  - a) Visible
  - b) HideValue
  - c) Editable
  - d) Multiline
  - e) DecimalPlaces
17. Which property is normally used in combination with the AutoSplitKey property? Choose one:
  - a) SaveValues
  - b) SplitIncrement
  - c) DelayedInput
  - d) MultipleNewLines
18. Ribbon tabs and menu sections are predefined in NAV and cannot be changed by the developer. True or False?
19. Inheritance between tables and pages operates two ways--tables can inherit attributes from pages and pages can inherit from tables. True or False?
20. For the purpose of testing, pages can be run directly from the Development Environment. True or False?

## Summary

You should now be relatively comfortable in the navigation of NAV and with the use of the Object Designer. You should be able to use the Page Wizard as an advanced beginner. If you have taken full advantage of the various opportunities to create tables and pages, both with our guidance and experimentally on your own, you are beginning to become a NAV Developer.

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 had a good introduction to Page Designer and a significant insight into the structure of some types of pages. With the knowledge gained, we have expanded our WDTU application system, enhancing our pages for data maintenance and inquiry.

In the next chapter, we will learn our way around the NAV Query and Report Designers. We will dig into the various triggers and controls that make up reports. We will also perform some Query and Report creation work to better understand what makes them tick and what we can do within the constraints of the Query and Report Designer tools.

# 5

# Queries and Reports

*"Data helps solve problems."*

- Anne Wojcicki

*"The greatest value of a picture is when it forces us to notice what we never expected to see."*

- John Tukey

In Microsoft Dynamics NAV 2017, Reports and Queries are two ways to extract and output data for the purpose of presentation to a user (Reports can also modify data). Each of these objects use tools and processes that are NAV based for the data extraction (XMLports, which can also extract and modify data, will be covered in Chapter 8, *Advanced NAV Development Tools*). In this chapter, we will focus on understanding the strengths of each of these tools and when and how they might be used. We will cover the NAV side of both Queries and Reports in detail to describe how to obtain the data we need to present to our users. We will cover output formatting and consumption of that data in less detail. There are currently no wizards available for either Query building or Report building, therefore all the work must be done step by step using programming tools and our skills as designer/developers. The following are the topics we will cover in this chapter:

- Queries and Reports
- Report Components - overview
- Report Data Flow
- Report Components - detail
- Creating and modifying Reports

## Queries

Reports have always been available in NAV as a data retrieval tool. Reports are used to process and/or manipulate the data through the Insert, Modify, or Delete functions with the option of presenting the data in a formatted, printable format. Prior to NAV 2013, data selection could only be done using C/AL code or DataItem properties to filter individual tables as datasets, and to perform loops to find the data required for the purpose. Then the Query object was created with performance in mind. Instead of multiple calls to SQL to retrieve multiple datasets to then be manipulated in C/AL, Queries allow us to utilize familiar NAV tools to create advanced T-SQL queries.

An NAV developer can utilize the new Query object as a source of data both in NAV and externally. Some of the external uses of NAV Queries are as follows:

- A web service source for SOAP, OData, and OData
- Feeding data to external reporting tools, such as Excel, SharePoint, and SSRS

Internally, NAV Queries can be used as follows:

- A direct data source for Charts.
- As providers of data to which Cues (displayed in Role Centers) are bound. See the Help **Walkthrough: Creating a Cue Based on a Normal Field and a Query**.
- As a dataset variable in C/AL to be accessed by other object types (Reports, Pages, Codeunits, and so on). See <https://msdn.microsoft.com/en-us/dynamics-nav/read-function--query-for> guidance on using the READ function to consume data from a Query.

Query objects are more limited than SQL stored procedures. Queries are more similar to an SQL View. Some compromises in the design of Query functionality were made for better performance. Data manipulation is not supported in Queries. Variables, subqueries, and dynamic elements, such as building a query based on selective criteria, are not allowed within the Query object.

The closest SQL Server objects that Queries resemble are SQL Views. One of the new features that allow NAV to generate advanced T-SQL statements is the use of **SQL Joins**. These include the following Join methods for two tables, A and B:

- **Inner:** This query compares each row of table A with each row of table B to find all the pairs of rows that satisfy the Join criteria.
- **Full Outer:** This join does not require each record in the two Joined tables to have a matching record so that all records from both A and B will appear at least once.

- **Left Outer Join:** In this join, every record from A will appear at least once, even if matching B is not found.
- **Right Outer Join:** In this join, every record from B will appear at least once, even if matching A is not found.
- **Cross Join:** This join returns the Cartesian product of the sets of rows from A and B. The Cartesian product is a set made up of rows that include the columns of each row in A along with the columns of each row in B for number of rows; in other words, including the columns of the rows in A plus those in B.



Note that Union Join, which joins all records from A and B without the Join criteria, is not available at this time.

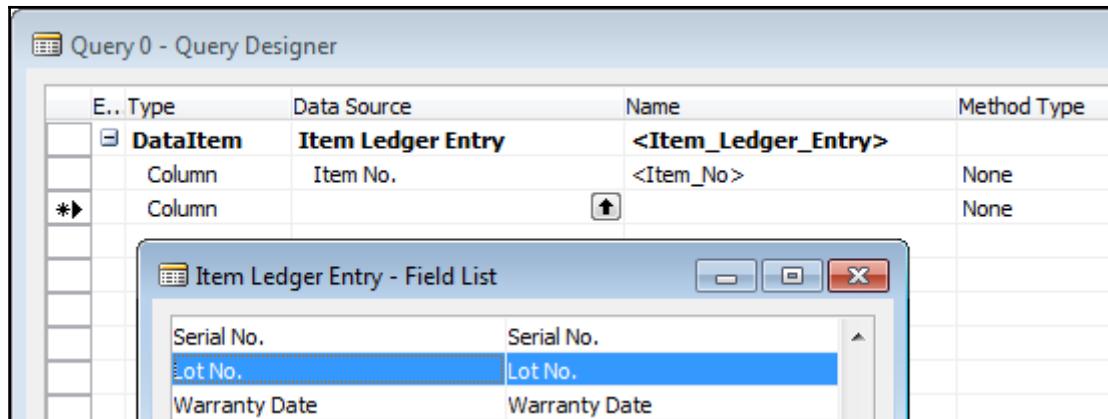
## Building a simple Query

Sometimes, it is necessary to quickly retrieve detailed information from one or more ledgers that may contain hundreds of thousands to many millions of records. The Query object is the perfect tool for such a data selection as it is totally scalable and can retrieve selected fields from multiple tables at once. The following example (using Cronus data) will show the aggregated quantity per bin of lot-tracked items in stock. This query can be presented to a user by means of either a report or a page:

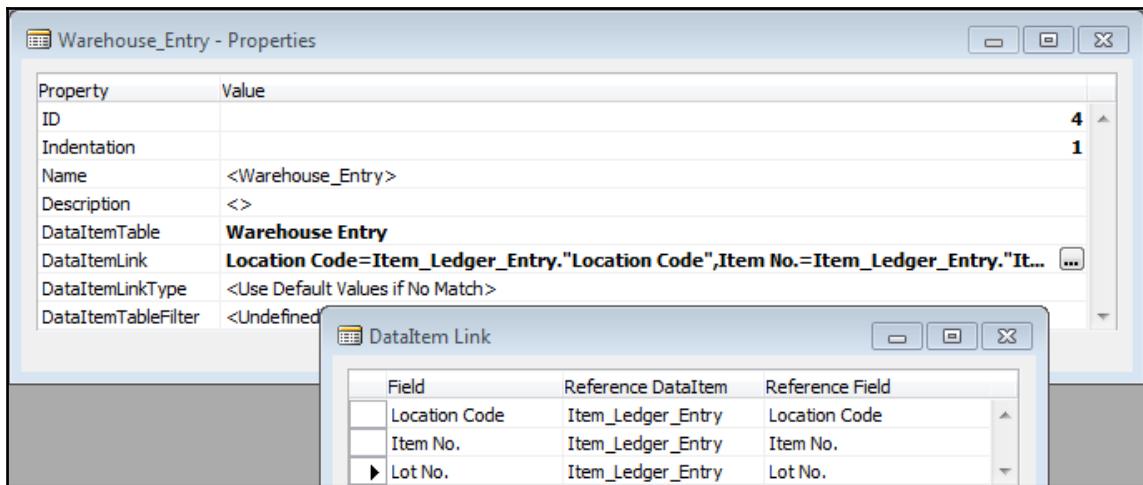
1. We will define the logic we need to follow and the data required to support that logic, then we will develop the Query. It is necessary to know what inventory is in stock and contains a lot number. This is accomplished using the **Item Ledger Entry** table.
2. However, the **Item Ledger Entry** record does not contain any bin information. This information is stored in the **Warehouse Ledger Entry** table.
3. The **Location Code**, **Item No.**, and **Lot No.** columns are used to match the **Item Ledger Entry** and **Warehouse Ledger Entry** records to make sure that the correct Items are selected.
4. In order to determine which bins are designated as pick bins, the **Bin Type** records that are marked as **Pick = True** need to be matched with the bins in **Warehouse Ledger Entry**.
5. Lastly, **Quantity** on each **Warehouse Entry** needs to be summed per **Location Code**, **Zone Code**, **Bin Code**, **Item No.**, and **Lot No.** to show the amount available in each bin.

Now that we have defined the necessary logic and data sources, we can create the desired Query object as follows.

1. The **Query Designer** is accessed from within the Development Environment through **Tools | Object Designer | Query**. The Query Designer can be opened either for creation of a new query by using the **New** button or for editing an existing query by highlighting the target object, then clicking the **Design** button.
2. Now define the primary DataItem in the **Data Source** column. The first DataItem is the **Item Ledger Entry** table. We can either type in the table name or the table number (in this case, table number 32). The Query may select from multiple tables (as we do in this example). All DataItems except the first must be indented. Each successively indented DataItem must have a link defined to a lesser-indented DataItem (because Union joins are not supported).
3. After defining the first DataItem, we will move focus to the first blank line, and the **Type** will default to **Column**. **Column** is a field from the **DataItem** table that will be output as an available field from the query dataset. The other **Type** option is **Filter**, which allows us to use the source column as a filter and does not output this column in the dataset. Use the Lookup arrow or the **Field** Menu to add the two following fields under **Item Ledger Entry: Item No.** and **Lot No.**, as shown in the following screenshot:



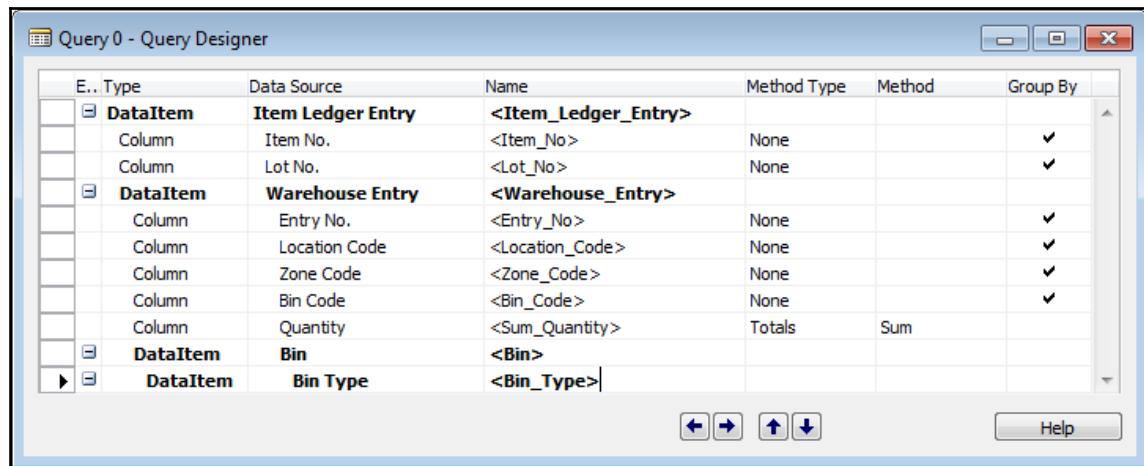
4. The next DataItem we need is the **Warehouse\_Entry** table. We must join it to the **Item Ledger Entry** by filling in the **DataItemLink** property. Link the **Location Code**, **Item No.**, and **Lot No.** fields between the two tables, as shown in the following screenshot:



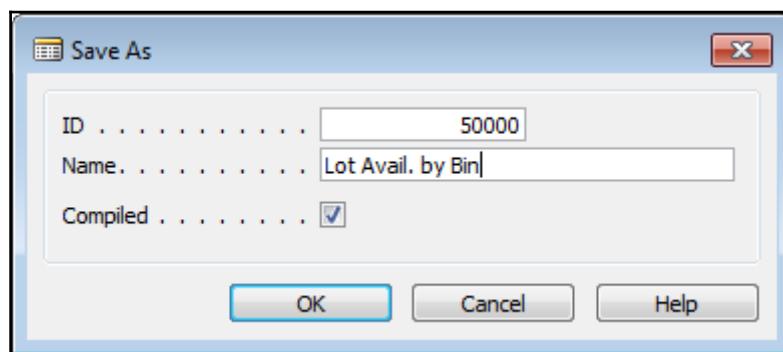
The following steps will define the rest of the DataItems, Columns, and Filters for this query:

1. Select **Entry No.**, **Location Code**, **Zone Code**, **Bin Code**, and **Quantity** as Columns under Warehouse Entry DataItem.
2. Add **Bin** table as the next DataItem.
3. Set the DataItem Link between the **Bin** and **Warehouse Entry** as the **Bin** table Code field linked to the **Bin Code** field for the **Warehouse Entry** table.
4. Add the **BinType** table as the last DataItem for this query. Create a DataItem Link between the **Bin Type** table Code field and the Bin table's **Bin Type Code** field.
5. Set the **DataItem Filter** as `Pick = CONST(Yes)` to only show the quantities for bins that are enabled for picking.

6. For the dataset returned by the Query, we will only want the total quantity per combination of the Location, Zone, Bin, Item, and Lot number. For the Column **Quantity** in the Warehouse Entry DataItem, set the **Method Type** column to **Totals**. The **Method** will default to **Sum** and the columns above **Quantity** will be marked with **Group By** checked. This shows the grouping criteria for aggregation of the **Quantity** field, as shown in the following screenshot:

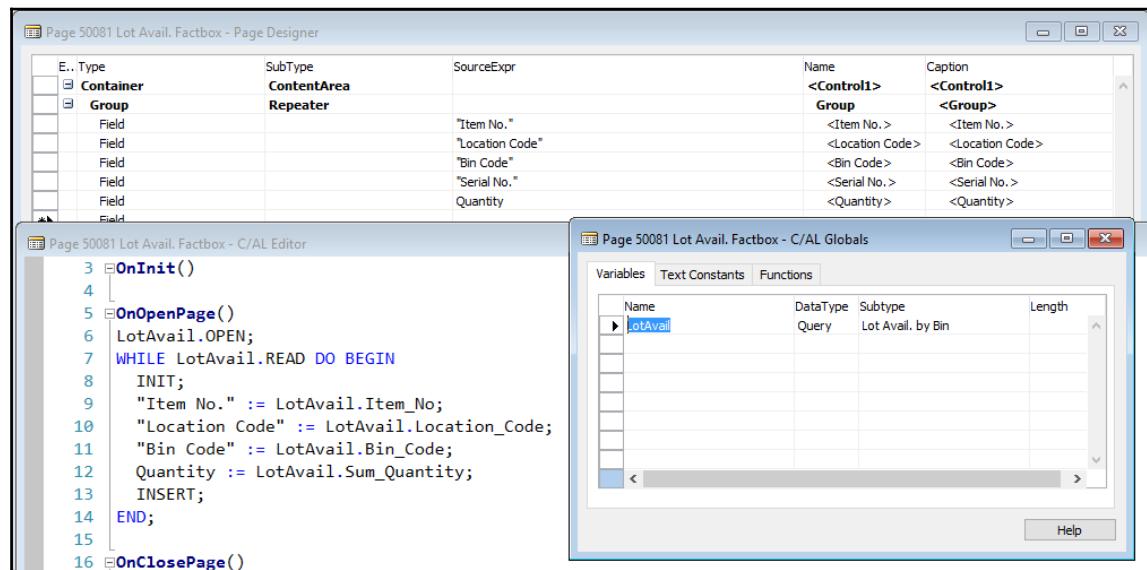


7. Once the DataItems and Columns are selected, the Query can be compiled and saved in the same manner as **Tables** and **Pages** are compiled and saved. The Query can be tested simply by highlighting it in the **Object Designer** and clicking on **Run**. Number and name the Query object, as shown in the following screenshot:



This query can be utilized internally in NAV 2017 as an indirect data source in a Page or a Report object. Although DataItems in Pages and Reports can only be database tables, we can define a Query as a variable, then use the Query dataset result to populate a temporary **Sourcetable**. In a page we define the **SourceTableTemporary** property to Yes and then load the table via C/AL code located in the **OnOpenPage** trigger, or in a report we might utilize a virtual table, such as the Integer table, to step through the Query result.

In our example, we use the **Warehouse Entry** table to define our temporary table because it contains all the fields in the Query dataset. In the **Page Properties**, we set the **SourceTableTemporary** to Yes (if we neglect marking this table as temporary, we are quite likely to corrupt the live data in the Warehouse Entry table). In the **OnOpenPage** trigger, the Query object (**LotAvail**) is filtered and opened. As long as the Query object has a dataset line available for output, the Query column values can be placed in the temporary record variable and be available for display, as shown in the following screenshot. Because this code is located in the **OnOpenPage** trigger, the temporary table is empty when this code begins execution. If the code were invoked from another trigger, the statement **Rec.DELETEALL** would be needed at the beginning to clear any previously loaded data from the table:



As the Query dataset is read, the temporary record dataset will be displayed on the page, as shown in the following screenshot:

The screenshot shows a Microsoft Dynamics NAV application window titled "Lot Avail. Factbox". At the top right, there is a search bar labeled "Type to filter (F3)" and a dropdown menu set to "Item No.". Below the search bar, it says "No filters applied". The main area displays a table with five columns: "Item No.", "Location Code", "Bin Code", "Serial No.", and "Quantity". There are three data rows:

Item No.	Location Code	Bin Code	Serial No.	Quantity
LS-75	WHITE	W-11-0001		12
LS-120	WHITE	W-11-0001		6
LS-150	WHITE	W-11-0001		7

When a Query is used to supply data to a Report, an `Integer DataItem` is defined to control stepping through the Query results. Before the report read loop begins, the Query is filtered and invoked so that it begins processing. As long as the Query object continues to deliver records, the `Integer DataItem` will continue looping. At the end of the Query output, the report will proceed to its `OnPostDataItem` trigger processing, just as though it had completed processing a normal table rather than a Query created dataset. This approach is a faster alternative to a design that would use several FlowFields, particularly if those FlowFields were only used in one or two periodic reports.

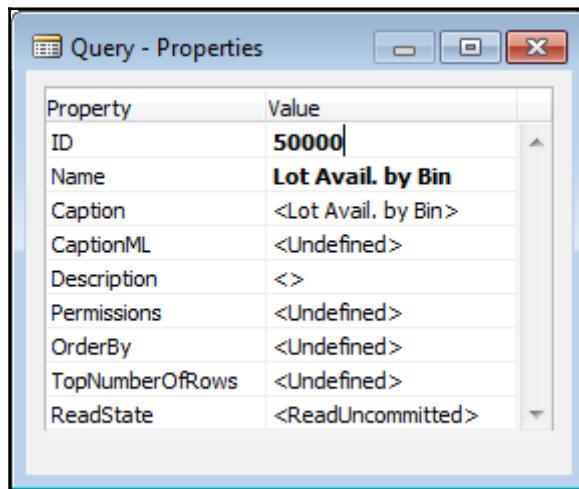
A similar approach to using a Query to supply data to a report is described in *Mark Brummel's Blog Tip #45* (<https://markbrummel.blog/2015/03/24/tip-45-nav2015-report-temporary-property/>).

## Query and Query Component properties

There are several Query properties we should review. Their descriptions follow.

## Query properties

The properties of the Query object can be accessed by highlighting the first empty line and clicking on the **Properties** icon, clicking on *Shift + F4*. or by navigating to **View | Properties**. The Properties of the Query we created earlier will look like this:

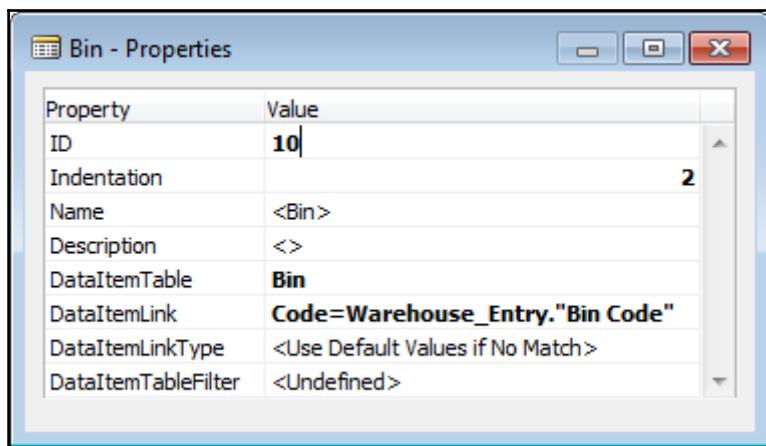


We'll review three of these properties:

- **OrderBy**: This provides the capability to define a sort, data column by column, ascending or descending, giving the same result as if a key had been defined for the Query result, but without the requirement for a key.
- **TopNumberOfRows**: This allows specification of the number of data rows that will be presented by the Query. A blank or 0 value shows all rows. Specifying a limit can make the Query complete much faster. This property can also be set dynamically from the C/AL code.
- **ReadState**: This controls the state (committed or not) of data that is included and the type of lock that is placed on the data read.

## DataItem properties

A Query Line can be one of three types: DataItem, Column, and Filter. Each has its own property set. Query DataItem properties can be accessed by highlighting a DataItem line and clicking on the **Properties** icon, clicking on *Shift + F4*, or by navigating to **View | Properties**:



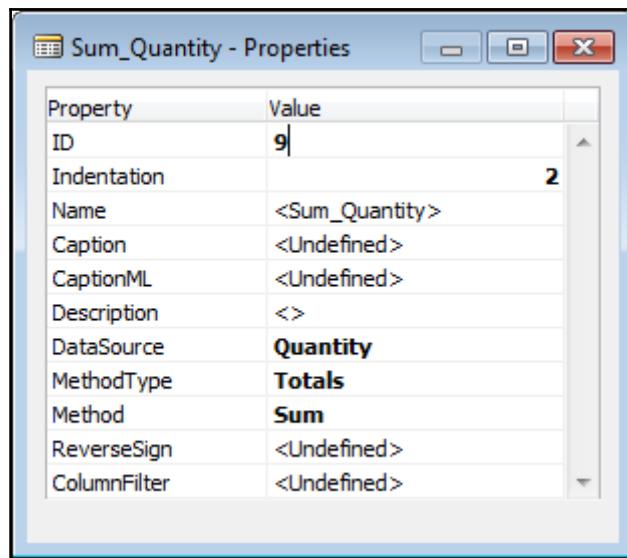
Again, we'll review a selected subset of these properties:

- **Indentation:** This indicates the relative position of this line within the Query's data hierarchy. The position in the hierarchy, combined with the purpose of the line, such as data, lookup, or total, determines the sequence of processing within the Query.
- **DataItemLinkType:** This can only be used for the subordinate DataItem relative to its parent DataItem, in other words, it only applies in a Query that has multiple DataItems. There are three value options:
  - **Use Default Values if No Match:** This includes the parent DataItem row, even when there is no matching row in the subordinate DataItem
  - **Exclude Row if No Match:** This skips the parent DataItem row if there is no matching row in the subordinate DataItem
  - **SQL Advanced Options:** This enables another property: the **SQLJoinType** property

- **SQLJoinType**: If enabled by the **DataItemLinkType** property, this property allows the specification of one of five different SQL Join Types (Inner, Left Outer, Right Outer, Full Outer, or Cross Join). More information is available in the **Help** sections, **SQLJoinType Property** and **SQL Advanced Options for Data Item Link Types**.
- **DataItemTableFilter**: This provides the ability to define filters to be applied to the DataItem.

## Column properties

The following screenshot is a Column **Property** screen showing the **Quantity** Column for our simple Query (the **MethodType** and **Method** properties are in use):



The properties shown in the preceding for a Query Column control are as follows:

- **MethodType**: This controls the interpretation of the following **Method** property. This can be Undefined/None, Date, or Totals.
- **Method**: This is dependent on the value of the **MethodType** property:
  - If **MethodType** = Date, then **Method** assumes that the Column accesses a date value. The value of **Method** can be Day, Month, or Year and the Query result for the Column will be the extracted day, month, or year from the source date data.

- If **MethodType** = **Totals**, then **Method** can be **Sum**, **Count**, **Avg**, **Min**, or **Max**. The result in the Column will be based on the appropriate computation. See the **Help** section for **Method** property for more information.
- **ReverseSign**: This reverses the sign of the Column value for numeric data.
- **ColumnFilter**: This allows us to apply a filter to limit the rows in the Query result. Filtering here is similar to, but more complicated than, the filtering rules that apply to **DataItemTableFilter**. Static **ColumnFilters** can be dynamically overridden and can also be combined with DataItemTableFilters. See the **Help** section for **ColumnFilters** property for more detailed information.

## Reports

Some consider the standard library of reports provided in the NAV product distribution from Microsoft to be relatively simple in design and limited in its features. Others feel that the provided reports satisfy most needs because they are simple, but flexible. Their basic structure is easy to use. They are made much more powerful and flexible by taking advantage of NAV's filtering and SIFT capabilities. There is no doubt that the existing library can be used as foundations for many of the special reports that customers require to match their own specific business management needs.

The fact is that NAV's standard reports are basic. In order to obtain more complex or more sophisticated reports, we must use features that are part of the product or feed processed data to external reporting tools such as Excel. Through creative use of these features, many different types of complex report logic may be implemented.

First, 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 make up NAV report data processing. Visual Studio Community Edition is the tool for our report layout work. Alternatively, the SQL Server Report Builder can be used if enabled as described in Chapter 1, *Introduction to NAV 2017*.

We'll create our reports in this book with Visual Studio Community Edition. We'll modify a report or two using the **C/SIDE 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). In addition, we'll take a look at the **Microsoft Word Report Layout** design capability.

## What is a report?

A **report** is a vehicle for organizing, processing, and displaying data in a format suitable for outputting to the user. Reports may be displayed on-screen in Preview mode, output to a file in Word, Excel, or PDF format (or, when appropriately designed, output in HTML, CSV or XML format), emailed to a user (or other consumer of information) or printed to hard-copy the old-fashioned way. All of the report screenshots in this book were taken from Preview mode reports.

Once generated, the data contents of a report are static. When an NAV 2017 Report is output in Preview mode, the report can have interactive capabilities. Those capabilities only affect the presentation of the data; they do not change the actual data contents included in the report dataset. Interactive capabilities include dynamic sorting, visible/hidden options and detail/summary expand/collapse functions. All specification of the data selection criteria for a report must be done at the beginning of the report run, before the report view is generated. NAV 2017 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 be classified as **Processing Only**, such as report 795 Adjust Cost - Item Entries, by setting the correct report property, that is, by setting the **ProcessingOnly** property to Yes. A Processing Only report will not display data to the user, but it will simply process and update data in the tables. Report objects are convenient to use for processing because the report's automatic **read-process-write** loop and the built-in Request page reduce coding that would otherwise be required. A report can add, change, or delete data in tables, whether the report is Processing Only or a typical report that generates output for viewing.

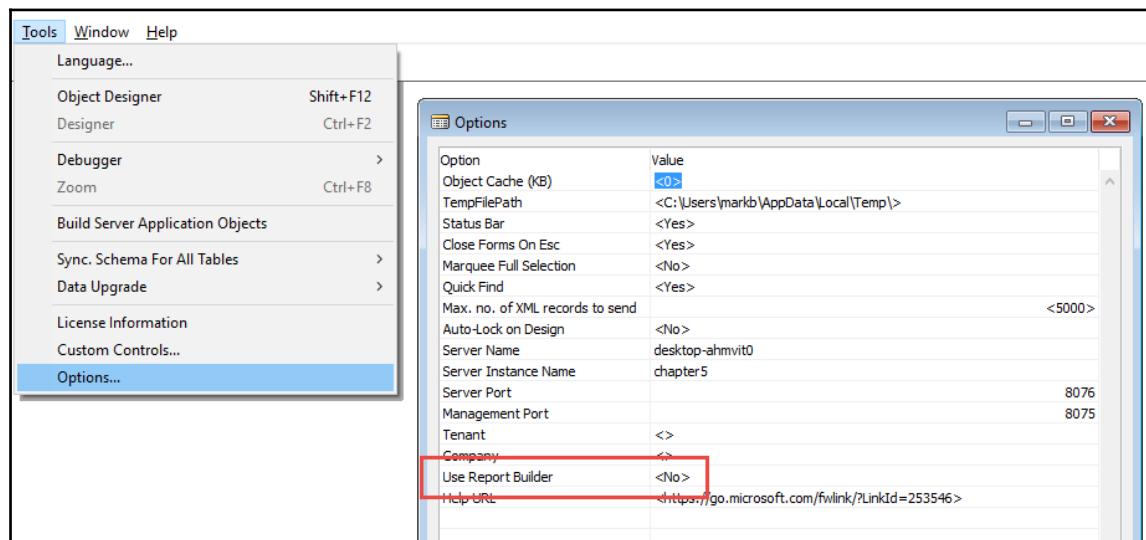
In general, reports are associated with one or more tables. A report can be created without being externally associated with any table, but that is an exception. Even if a report is associated with a particular table, it can freely access and display data from other referenced tables.

## Four NAV report designers

Any NAV 2017 report design project uses at least two Report Designer tools. The first is the Report Designer that is part of the C/SIDE development environment. The second is the developer's choice of **Visual Studio** or the **SQL Server Report Builder** or **Microsoft Word**. Refer to the Microsoft Dynamics NAV Development Environment Requirements for information about the choice of tools for handling RDLC report layouts for NAV 2017. We must also be careful to make sure we obtain the proper version of the report tool compatible with the NAV Version we are using. The SQL Server Report Builder is installed by default during a NAV system install. There is also a free version of Visual Studio, the Community Edition, available at

<https://msdn.microsoft.com/en-us/visual-studio-community-vs.aspx>.

For our work, we will use the combination of the **C/SIDE Report Designer** and Visual Studio Community Edition. If you want to access the SQL Server Report Builder, navigate to **Tools | Options** and set **Use Report Builder** to Yes, as shown in the following screenshot:



The option using **Microsoft Word** is aimed at the goal of allowing customers to be more self-sufficient handling quick, simple changes in format while requiring less technical expertise. Because our focus in this book is on becoming qualified NAV Developers, we will leave discussion of layout formatting with Word for later.



The report development process for a NAV 2017 report begins with data definition in the C/SIDE Report Designer. All the data structure, working data elements, data flow, and C/AL logic are defined there. We must start in the Report Designer to create or modify report objects. Once all of the elements of the dataset definition and Request page are in place, the development work proceeds to the SQL Report Builder, Visual Studio, or Word, where the display layout work is done, including any desired dynamic options.

When a report layout is created, SQL Report Builder or Visual Studio (whichever tool you are using) builds a definition of the report layout in the XML-structured

**ReportDefinitionLanguageClient-side (RDLC)**. If Word is used to build a NAV 2017 Report Layout, the result is a custom XML part that is used to map the data into a report at runtime. When we exit the layout design tool, the latest copy of the RDLC code is stored in the current C/SIDE Report object. When we exit the C/SIDE Report Designer and save our Report object, Report Designer saves the combined set of report definition information, C/SIDE and RDLC, in the database.

If we export a report object in text format, we can see the two separate sets of report definition. The XML-structured RDLC is quite obvious (beginning with the heading RDLDATA), as shown in the following code snippet:

```
>
CODE
{
VAR
    LastFieldNo@1000 : Integer;
    EmployeeFilter@1001 : Text;
    Employee__BirthdaysCaptionLbl@7301 : TextConst 'ENU=Employee - Birthdays';
    CurrReport_PAGENOCaptionLbl@8565 : TextConst 'ENU=Page';
    Full_NameCaptionLbl@8418 : TextConst 'ENU=Full Name';
    Employee__Birth__Date__CaptionLbl@1647 : TextConst 'ENU=Birth Date';

BEGIN
END.
}
RDLDATA
{
    <?xml version="1.0" encoding="utf-8"?>
<Report xmlns:rd="http://schemas.microsoft.com/SQLServer/reporting/reportdesigner"
<AutoRefresh>0</AutoRefresh>
<DataSources>
    <DataSource Name="DataSource">
        <ConnectionProperties>
            <DataProvider>SQL</DataProvider>
            <ConnectionString />
        </ConnectionProperties>
        <rd:SecurityType>None</rd:SecurityType>
        <rd:DataSourceID>b3d0e809-2738-4fe0-ae5d-c8879f2acc45</rd:DataSourceID>
    </DataSource>
</DataSources>
```

For the experienced NAV Classic Client report developer who is moving to Role Tailored Client projects, it is initially a challenge to learn exactly which tasks are done using which report development tool and to learn the intricacies of the SQL Server or Visual Studio report layout tools. The biggest challenge is the fact that there are no wizards to help with NAV 2017 report layout. All our report development must be done manually, one field or format at a time. If we would like Microsoft to invest in report layout wizards for future releases, we should tell them.



We can submit suggestions on any Dynamics NAV related topic through Microsoft Connect at <http://connect.microsoft.com/directory/>.

NAV allows us to create reports of many types with different look and feel attributes. The consistency of report look and feel does not have the same level of design importance as it has for pages. There may be patterns developed that relate to reports, so before starting a new format of report, it is best to check if there is an applicable pattern.

Good design practice dictates that enhancements should integrate seamlessly both in process and appearance unless there is an overwhelming justification for being different. There are still many opportunities for reporting creativity. The tools available within NAV to access and manipulate data for reports are very powerful. Of course, there is always the option to output report results to other processing/presentation tools, such as Excel or third-party products. Outputting a report to Excel allows the user to take advantage of the full Excel toolset to further manipulate and analyze the reported data.

## NAV report types

The standard NAV application uses only a few of the possible report styles, most of which are in a relatively basic format. The following are the types of reports included in NAV 2017:

- **List:** This is a formatted list of data. A standard list is the **Inventory - List** report (Report 701):

No.	Description	BOM	Base Unit of Measure	Inventory Posting Group	Shelf No.	Vendor Item No.	Lead Time Calculation	Reorder Point	Last Modified
1000	Bicycle	No	PCS	FINISHED	F4			0	2017-06-20 10:15:00
1001	Touring Bicycle	No	PCS	FINISHED	F5			0	2017-06-20 10:15:00
1100	Front Wheel	No	PCS	FINISHED	F6		2W	100	2017-06-20 10:15:00
1110	Rim	No	PCS	FINISHED	F1	266666		200	2017-06-20 10:15:00
1120	Spokes	No	PCS	RAW MAT	A1	45455		5,000	2017-06-20 10:15:00
1150	Front Hub	No	PCS	FINISHED	F7			100	2017-06-20 10:15:00
1151	Axle Front Wheel	No	PCS	RAW MAT	A2	11111		100	2017-06-20 10:15:00
1155	Socket Front	No	PCS	RAW MAT	A3	A-12122		100	2017-06-20 10:15:00

- **Document:** This is formatted similarly to a pre-printed form, where a page (or several pages) contains a header, detail, and footer section with dynamic content. Examples of Document reports 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 preview:

<b>Sales - Invoice</b>							
Customer Information				Invoice Details			
The Cannon Group PLC Mr. Andy Teal 192 Market Square Birmingham, B27 4KT Great Britain				CRONUS International Ltd. 5 The Ring Westminster W2 8HG London			
Bill-to Customer No.	10000	Phone No.	0666-666-6666	VAT Registration No.	789456278	E-Mail	
Invoice No.	103001	Home Page		Posting Date	January 25, 2016	VAT Reg. No.	G8777777777
Due Date	February 25, 2016	Giro No.	888-9999	Document Date	January 25, 2016	Bank	World Wide Bank
Payment Terms	1 Month/2% 8 days	Account No.	99-99-888	Shipment Method	Ex Warehouse	Salesperson	Peter Saddow
Prices Including VAT	No						
No.	Description	Posted Shipment Date	Quantity	Unit of Measure	Unit Price	Discount %	VAT Identifier
TIMOTHY	Assembling Furniture, January	01/25/16	25	Hour	54.00	VAT10	1,350.00
TIMOTHY	Assembling Furniture, January	01/25/16	120	Miles	54.00	VAT10	6,480.00
						Subtotal	7,830.00
						Invoice Discount Amount	-391.50
						Total GBP Excl. VAT	7,438.50
						10% VAT	743.85
						Total GBP Incl. VAT	8,182.35
VAT Amount Specification							
VAT Identifier	VAT %	Line Amount	Invoice Discount Base Amount	Invoice Discount Amount	VAT Base	VAT Amount	
VAT10	10	7,830.00	7,830.00	391.50	7,438.50	743.85	
<b>Total</b>		<b>7,830.00</b>	<b>7,830.00</b>	<b>391.50</b>	<b>7,438.50</b>	<b>743.85</b>	

List and Document report types are defined based on their layouts. The next three report types are defined based on their usage rather than their layouts, which are as follows:

- **Transaction:** This provides 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:

The screenshot shows a Microsoft Dynamics NAV application window titled "G/L Register". The window displays a list of general ledger entries for CRONUS International Ltd. on Sunday, February 15, 2015, page 52, by user ARTHUR/DAVE. The table has 11 columns: Posting Date, Document Type, Document No., G/L Account No., Name, Description, VAT Amount, Gen. Postin g Type, Gen. Bus. Group, Gen. Prod. Postin g Group, Amount, and Entry No.

Posting Date	Document Type	Document No.	G/L Account No.	Name	Description	VAT Amount	Gen. Postin g Type	Gen. Bus. Group	Gen. Prod. Postin g Group	Amount	Entry No.
<b>Register No. 127</b>											
12/07/15	Invoi	103029	6110	Sales, Retail - Do	Invoice 1001	-265.78	Sale	NATIO	RETAIL	-1,063.10	2815
12/07/15	Invoi	103029	5610	Sales VAT 25 %	Invoice 1001	0.00				-265.78	2816
12/07/15	Invoi	103029	2310	Customers Domes	Invoice 1001	0.00				1,328.88	2817
<b>Register No. 128</b>											
11/29/15	Invoi	103030	6110	Sales, Retail - Do	Invoice 1002	-133.35	Sale	NATIO	RETAIL	-533.40	2818
11/29/15	Invoi	103030	5610	Sales VAT 25 %	Invoice 1002	0.00				-133.35	2819
11/29/15	Invoi	103030	2310	Customers Domes	Invoice 1002	0.00				666.75	2820

- **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 **GeneralJournal - Test** batch:

General Journal - Test													
1 of 2?      100%   Find   Next <b>General Journal - Test</b> CRONUS International Ltd.													
2/15/2015 8:22 PM Page 1 ARTHUR\DAVE													
Journal Template Name: GENERAL Journal Batch: DEFAULT													
Gen. Journal Line: Journal Template Name: GENERAL, Journal Batch Name: DEFAULT, Document No.: G00001													
Posting Date	Document Type	Document No.	Account Type	Account No.	Name	Description	Gen. Post Type up	Post ing	Pro d. Gro up	Bus. d. Gro up	Amount	Bal. Account No.	Balance (LCY)
01/29/16	G00001	G/L Ac	1220	Increases du	Packing Machine	Puro NATI MIS					110.97		110.97
01/29/16	G00001	G/L Ac	8210	Office Suppli	Boxes for Packin	Puro NATI MIS					24.62		24.62
01/29/16	G00001	G/L Ac	8210	Office Suppli	Glue for Packing	Puro NATI MIS					27.75		27.75
01/29/16	G00001	Bank A	WWB-OPER	World Wide	Materials for Pac						-163.34		-163.34
										Total (LCY)	0.00		0.00
Reconciliation													
No.	Name				Net Change in Jnl.	Balance after Posting							
5310	Revolving Credit				-163.34	-1,383,484.01							

- **Posting:** These reports are printed as an audit trail as part of a **Post and Print** process. Posting report printing is controlled by the user's choice of either a **PostingOnly** option or a **PostandPrint** option. The Posting portions of both options work the same. Post and Print runs a report that is selected in the application setup (in the applicable Templates page in columns that are hidden by default). This type of posting audit trail report, which is often needed by accountants, can be regenerated completely and accurately at any time. The default setup uses the same report that one would use as a Transaction (history) report, similar in format to the **G/L Register** shown in the following screenshot:

G/L Register										
1 of 1     100% Find   Next										
G/L Register							Tuesday, March 07, 2017			
CRONUS International Ltd.							Page 1			
G/L Register No.: 125										
Posting Date	Document Type	Document No.	G/L Account No.	Name	Description	VAT Amount	Gen. Postin g Type	Bus. Postin g Group	Prod. Postin g Group	Amount Entry No.
Register No.	125									
01/25/18	2805	8320	Consultant Service	Payment, Acco		11.05	Purchas e	DOME	SERVI	110.52
01/25/18	2805	5631	Purchase VAT 10	Payment, Acco		0.00				11.05
01/25/18	2805	5310	Revolving Credit	Payment, Acco		0.00				-121.57
01/25/18	2807	8910	Other Costs of Op	Packing Tape 2		4.92	Purchas e	DOME	MISC	19.70

## Report types summarized

The following list describes the different basic types of reports available in NAV 2017:

Type	Description
List	This is used to list volumes of like data in a tabular format, such as a list of Sales Order Lines, a list of Customers, or a list of General Ledger Entries.
Document	This is used in <i>record-per-page header plus line item detail</i> dual layout situations, such as a Sales Invoice, a Purchase Order, a Manufacturing Work Order, or a Customer Statement.
Transaction	This generally presents a list of transactions in a nested list format, such as a list of General Ledger Entries grouped by G/L Account, Physical Inventory Journal Entries grouped by Item, or Salesperson To-Do List by Salesperson.
Test	This prints in a list format as a pre-validation 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	This prints in a 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.

Processing Only	This type of report only processes data and does not generate a report output. It has the <b>ProcessingOnly</b> report property set to Yes.
-----------------	---

Many reports in the standard system don't fit neatly within the preceding categories but are variations or combinations. Of course this is also true of many custom reports.

## 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. Common key report purpose names include the words Journal, Register, List, Test, and Statistics. Some examples: **GeneralJournal-Test**, **G/LRegister**, and **Customer - Order Detail**.

When there are conflicts between naming based on the associated tables and naming based on the use of the data, the usage context should take precedence in naming reports, just as it does with pages. One absolute requirement for names is that they must be unique; no duplicate names are allowed for a single object type. Remember, the **Caption** (what shows on the printed report heading) is not the same as the **Name** (the internal name of the report object). Each one is a different report property.

## Report components - overview

What we generally refer to as the report or report object created with SQL Server Report Builder or Visual Studio Report Designer is technically referred to as an **RDLC Report**. (From here on we will focus on Visual Studio report layout tool. The tools are somewhat different, but the end results are basically the same.) An RDLC Report includes the information describing the logic to be followed when processing the data (the data model), the dataset structure that is generated by C/SIDE, and the output layout designed with Visual Studio. RDLC Reports are stored in the NAV database. Word report XML layouts are also stored in the NAV database. NAV 2017 allows there to be multiple RDLC and Word formats for a single report. 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 of the triggers, and even some of the properties. Where those parallels exist, we should take notice of them. Where there is consistency in the NAV toolset, it is easier to learn and use.

# Report Structure

The overall structure of an NAV RDLC Report consists of the following elements:

- Report properties
- Report triggers
- Request page
  - Request Page Properties
  - Request Page Triggers
  - Request Page Controls
    - Request Page Control Triggers
- DataItems
  - DataItem Properties
  - DataItem Triggers
  - Data Columns
    - Data Column Properties
- SSRB (RDLC) Layout
  - SSRB (RDLC) Controls
    - SSRB (RDLC) Control Properties
- Word Layout
  - Word layout template
  - Word Controls
    - Word Control Properties

## Report Data overview

Report components, Report Properties and Triggers, Request Page Properties and Triggers, and DataItems and their Properties and Triggers define the data flow and overall logic for processing the data. Another set of components, Data Fields, and Working Storage, are defined subordinate to the DataItems (or Request Page). These are all designed/defined in the **C/SIDE Report Dataset Designer**.



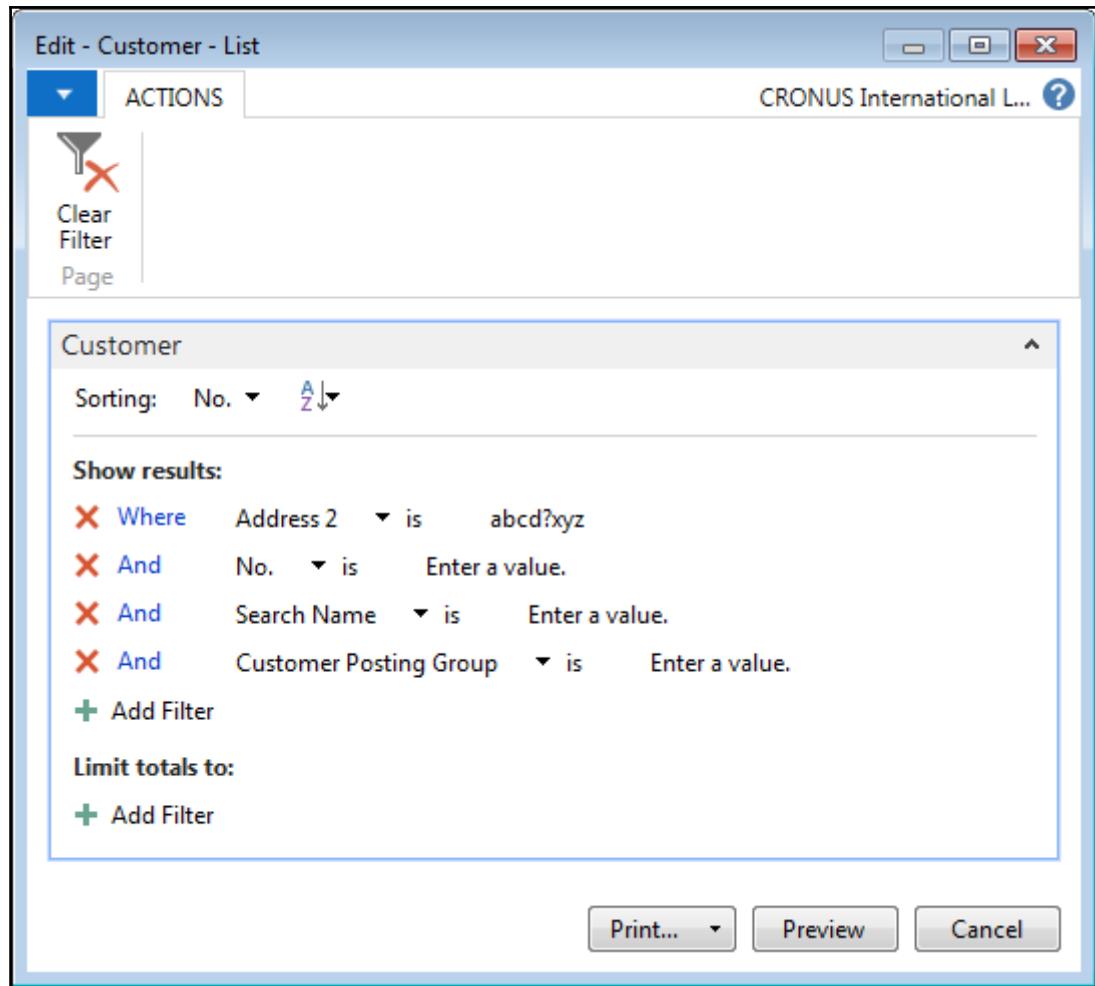
Data Fields are defined in this book as the fields contained in the DataItems (application tables). Working Storage (also referred to as Working Data or variables) fields are defined in this book as the data elements that are defined within a report (or other object) for use in that object. The contents of Working Storage data elements are not permanently stored in the database. All of these are collectively referred to in the NAV Help as columns.

These components define the data elements that are made available to Visual Studio as a Dataset to be used in the layout and delivery of results to the user. In addition, Labels (text literals) for display can be defined separately from any DataItem, but also included in the Dataset passed to Visual Studio. Labels must be **Common Language Specification (CLS)**-compliant names, which means labels can contain only alpha, decimal, and underscore characters and must not begin with an underscore. If the report is to be used in a multi-language environment, the **CaptionML** label must be properly defined to support the alternate languages.



Visual Studio cannot access any data elements that are not defined within the Report Designer. Each data element passed in the Dataset, whether a Data Field or Working Data, must be associated with a DataItem (except for Labels).

The Report Request Page displays when a report is invoked. Its purpose is to allow users to enter information to control the report. Control information entered through a Request Page may include filters, control dates, other parameters, and specifications as well as formatting or processing options to use for this report run. The Request Page appears once at the beginning of a report at runtime. The following screenshot shows a sample Request Page, the one associated with the Customer - List (Report 101):



## Report Layout overview

The Report Layout is designed in Visual Studio using data elements defined in Dataset DataItems by the Report Designer, then made available to Visual Studio. The Report Layout includes the Page Header, Body, and Page Footer.

In most cases, the Body of the report is based on an RDLC Table layout control defined in Visual Studio. The Table control is a data grid used for layout purposes and is not the same as a data table stored in the NAV database. The terminology can be confusing. When the NAV Help files regarding reports refer to a table, we 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 rows.

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, color, and many other display details.

Of course, if the display target changes dramatically, for example, from a desktop workstation display to a browser on a phone, the appearance of the Report Layout will change dramatically as well. One of the advantages of the NAV reporting layout toolset is to support the required flexibility. Because we must expect significant variability in our user's output devices (desktop video, browser, tablet, phone), we should design and test accordingly.

## 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, we will define the DataItems (the tables) that the report will process. We can create a processing-only report that has no data items (if no looping through database data is required), but that situation often calls for a code unit to be used. In a report, NAV automatically creates a data flow process for each DataItem or table reference. This automatically created data flow provides specific triggers and processing events for each data item, as follows:

- Preceding the DataItem
- After reading each record of the DataItem
- Following the end of the DataItem

The underlying **black-box** report logic (the part we can't see or affect) automatically loops through the named tables, reading and processing one record at a time. Therefore, any time we need a process that steps through a set of data one record at a time, it is often easier to use a report object.

The reference to a database table in a report is referred to as a DataItem. The report data flow structure allows us to nest data items to create a hierarchical grandparent, parent, and child structure. If DataItem 2 is nested within DataItem 1, and related to DataItem 1, then for each record in DataItem 1, all of the related records in DataItem 2 will be processed.

The following example uses tables from our WDTU system. The design is for a report to list all the scheduled instances of a **Radio Show Playlist** DataItem grouped by **Radio Show**, in turn grouped by **Radio Show Type**. Thus, **Radio Show Type** is the primary table (DataItem1). For each **Radio Show Type**, we want to list all the **Radio Show** DataItems of that type (DataItem2). And for each **Radio Show**, we want to list all the scheduled instances of that show recorded in the **Playlist Header** (DataItem3).

Open the Object Designer, select the **Report** object type, and click the New button. On the **Report Dataset Designer** screen, we will first enter the table name, **Radio Show Type** (or table number 50001), as we can see in the following screenshot. The DataItem Name, to which the C/AL code will refer to, is DataItem1 here in our example. Then, we will enter the second table, **Radio Show**, which is automatically indented relative to the data item above, the superior or parent data item. This indicates the nesting of the processing of the indented (child) data item within the processing of the superior (parent) data item.

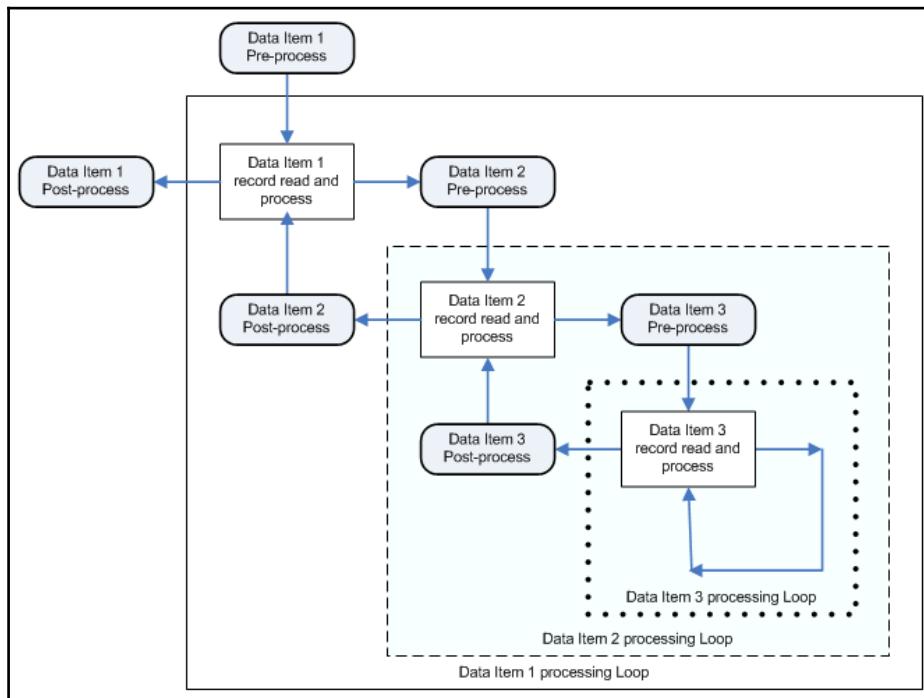
**For our example, we have renamed the DataItems to better illustrate report data flow.** The normal behavior would be for the **Name** in the right column to default to the table name shown in the left column (for example, the Name for Radio Show would be <**Radio Show**> by default). This default DataItem Name would only need to be changed if the same table appeared twice within the DataItem list. If there were a second instance of Radio Show for example, we could simply give it the Name RadioShow2, but it would be much better to give it a name describing its purpose in context.

For each record in the parent data item, the indented data item will be fully processed, dependent 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 parent-child definition.

For our example, we enter a third table, `Playlist Header`, and our example name of `DataItem3` as shown in the following screenshot:

Expanded	Data Type	Data Source	Name
	<b>DataItem</b>	<b>Radio Show Type</b>	<b>DataItem1</b>
	<b>DataItem</b>	<b>Radio Show</b>	<b>DataItem2</b>
▶	<b>DataItem</b>	<b>Playlist Header</b>	<b>DataItem3</b>

The following diagram shows the data flow for the preceding DataItem structure. The chart boxes are intended to show the nesting that results from the indenting of the DataItems in the preceding screenshot. The **Radio Show** DataItem is indented under the **Radio Show Type** DataItem. This means for every processed **Radio Show Type** record, all of the selected **Radio Show** records will be processed. This same logic applies to the **Playlist Header** records and **Radio Show** records, that is, for each **Radio Show** record processed, all selected **Playlist Header** records are processed:



The blocks visually illustrate how the data item nesting controls the data flow. As we can see, the full range of processing for **Data Item2** occurs for each **DataItem1** record. In turn, the full range of processing for **Data Item3** occurs for each **Data Item2** record.

In NAV 2017, report processing occurs in two separate steps, the first tied primarily to what has been designed in the C/SIDE Report Designer, the second tied to what has been designed in Visual Studio. The data processing represented in the preceding diagram all occurs in the first step, yielding a complete dataset containing all the data that is to be rendered for output.

If the output is to be displayed in the NAV Client for Windows, once the dataset is processed for display by the RDLC code created by Visual Studio, the results are handed off to the Microsoft Report Viewer. The Microsoft Report Viewer provides the NAV 2017 reporting capabilities, such as various types of graphics, interactive sorting and expand/collapse sections, output to PDF, Word and Excel, and other advanced features. Other clients are served by rendering tools that address each client's capabilities and limitations.

## **Report components - detail**

Earlier, we reviewed a list of the components of a Report object. Now we'll review detailed information about each of those components. Our goal here is to understand how the pieces of the report puzzle fit together.

## C/SIDE Report Properties

The Report Designer's Report Properties are shown in the following screenshot. Some of these properties have essentially the same purpose as similarly named properties in pages and other objects:

The screenshot shows the 'Report - Properties' dialog box. It contains a table with two columns: 'Property' and 'Value'. The 'Property' column lists various report settings, and the 'Value' column shows their current configurations. The properties listed include ID, Name, Caption, CaptionML, Description, UseRequestPage, UseSystemPrinter, EnableExternalImages, EnableHyperlinks, EnableExternalAssemblies, ProcessingOnly, ShowPrintStatus, TransactionType, Permissions, PaperSourceFirstPage, PaperSourceDefaultPage, PaperSourceLastPage, PreviewMode, DefaultLayout, WordMergeDataItem, and PDFFontEmbedding.

Property	Value
ID	50000
Name	<b>Radio Show List</b>
Caption	<Radio Show List>
CaptionML	<Undefined>
Description	<>
UseRequestPage	<Yes>
UseSystemPrinter	<No>
EnableExternalImages	<No>
EnableHyperlinks	<No>
EnableExternalAssemblies	<No>
ProcessingOnly	<No>
ShowPrintStatus	<Yes>
TransactionType	<UpdateNoLocks>
Permissions	<Undefined>
PaperSourceFirstPage	<Undefined>
PaperSourceDefaultPage	<Undefined>
PaperSourceLastPage	<Undefined>
PreviewMode	<Normal>
DefaultLayout	<RDLC>
WordMergeDataItem	<>
PDFFontEmbedding	<Default>

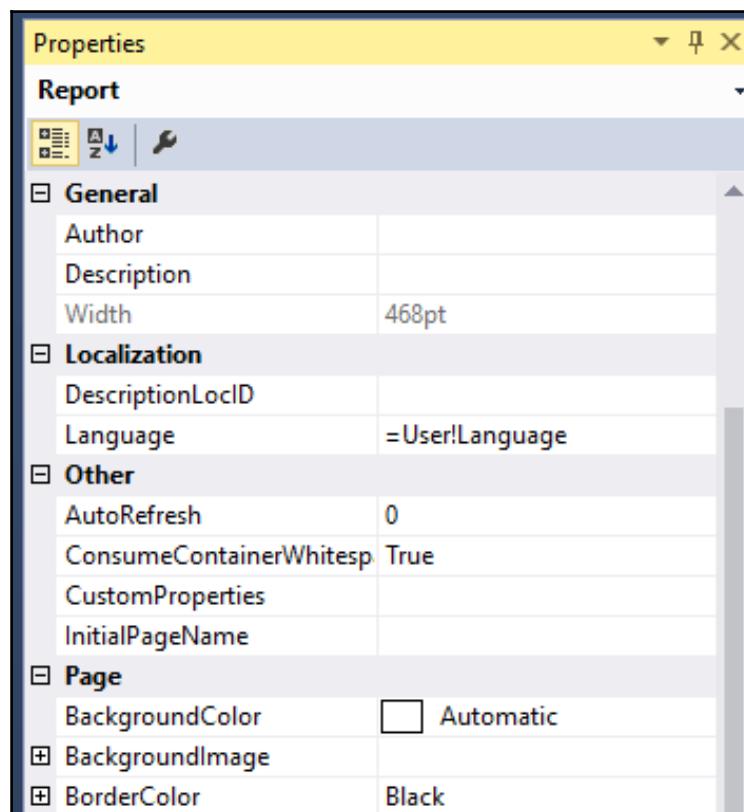
The properties in the preceding screenshot are defined as follows:

- **ID:** This is the unique report object number.
- **Name:** This is the name by which this report is referred to within the C/AL code.
- **Caption:** This is the name that is displayed for this report; **Caption** defaults to **Name**.
- **CaptionML:** This is the **Caption** translation for a defined alternative language.
- **Description:** This is for internal documentation.
- **UseRequestPage:** This can either be **Yes** or **No**, controlling whether or not the report will begin with a Request Page for user parameters to be entered.
- **UseSystemPrinter:** This 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. More on User/Report printer setup can be found in the NAV application help.
- **EnableExternalImages:** If **Yes**, this allows links to external (non-embedded) images on the report. Such images can be outside the NAV database.
- **EnableHyperLinks:** If **Yes**, this allows links to other URLs including other reports or pages.
- **EnableExternalAssemblies:** If **Yes**, this allows the use of external custom functions as part of the report.
- **ProcessingOnly:** This is set to **Yes** when the report object is being used only to process data and no reporting 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.
- **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 we want a Report Progress Window, we must create our own dialog box.

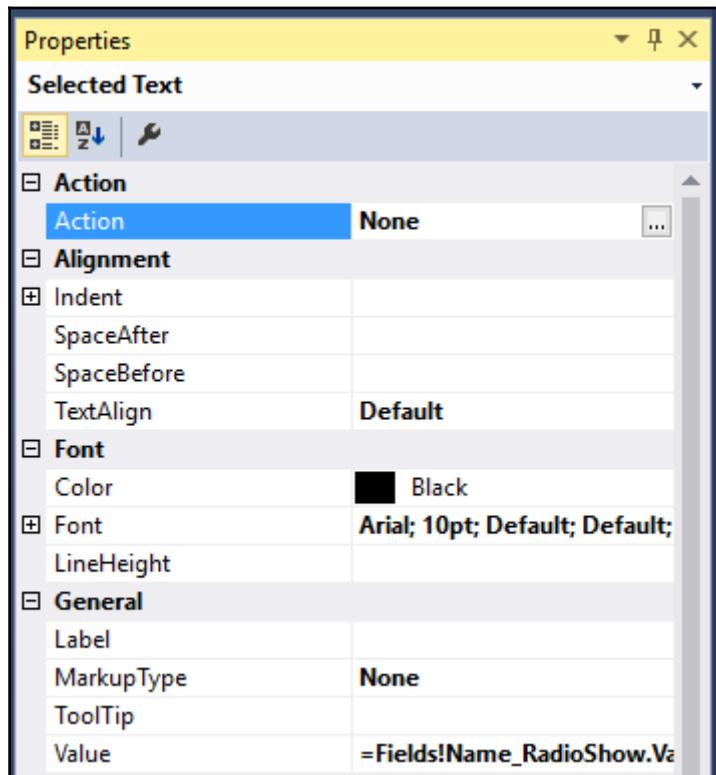
- **TransactionType:** This can be in one of four basic options: **Browse**, **Snapshot**, **UpdateNoLocks**, and **Update**. These control the record locking behavior to be applied in this report. The default is **UpdateNoLocks**. This property is generally only used by advanced developers.
- **Permissions:** This provides report-specific setting of permissions, which are the rights to access data tables, 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.
- **PaperSourceFirstPage**, **PaperSourceDefaultPage**, and **PaperSourceLastPage**: These allow us a choice of paper source tray based on information in the fin.stx file (fin.stx is an installation file that contains the active language messages and reserved words in addition to system control parameters that are defined by Microsoft and are not modifiable by anyone else).
- **PreviewMode:** This specifies choice of the default Normal or Print Layout, causing the report to open up in either the interactive view allowing manipulation or in the fixed format that it will appear on a printer.
- **DefaultLayout:** This specifies whether the report will default to using either the Word or the RDLC layout.
- **WordMergeDataItem:** This defines the table on which the outside processing loop will occur for a Word layout, which is equivalent to the first DataItem's effect on an RDLC layout).
- **PDFFontEmbedding:** This specifies if the font will be embedded in generated PDF files from the report.

## Visual Studio - Report Properties

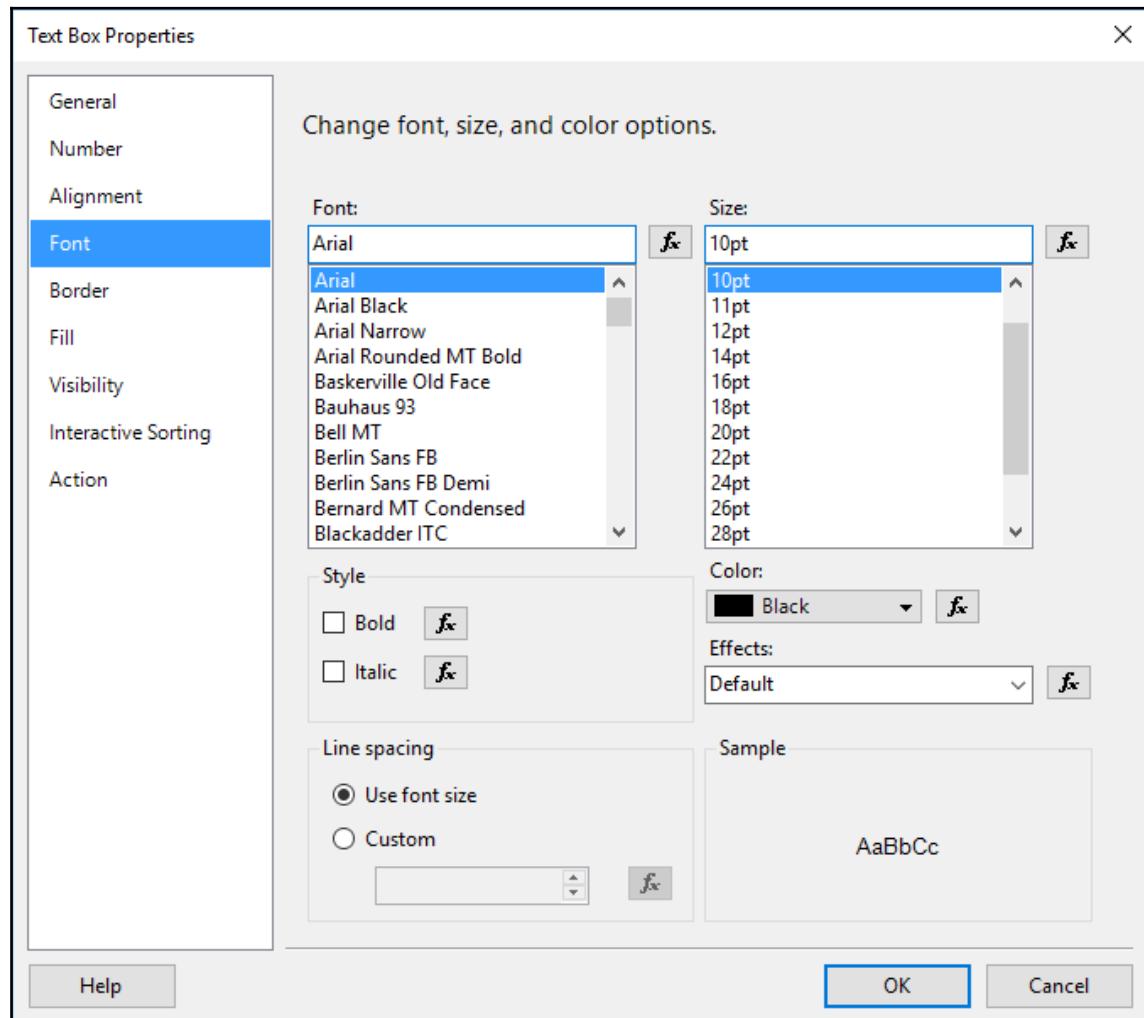
The Visual Studio **Properties** Window is docked by default on the right side of the screen and is as shown (as with most Visual Studio windows, it can be redocked, hidden, or float):



By highlighting various report elements (individual cells, rows, columns, and others), we can choose which set of properties we want to access, those of the whole Report, of the Body, of the Tablix, or of individual textboxes. Once we have chosen the set of properties, we can access those properties in one of two ways. The obvious way is through the window that opens up when we choose which control properties to display. The following screenshot shows the properties of a textbox with the **Font** properties expanded. **Font** properties are the properties most often accessed to change the appearance of a report layout, although a variety of other field formatting properties are also available:



Another way to access property detail is to highlight the element of interest; right click and select the **Properties** option for that element. This opens up a control-specific **Properties** window that includes a menu of choices of property categories to access. In the following screenshot, **Text Box Properties** is opened and the available options for the **Font** properties accessed are displayed:

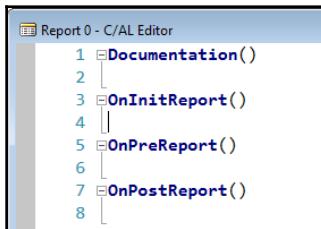


Some may feel it's easier to find and change all the properties options here. This applies to Report-level properties controlling macroelements, such as **Paper Orientation**, as well as Control-level properties controlling microelements, such as font size. Report-level properties also include the facility to include expressions that affect the behavior or controls. The expression shown in the following code snippet screenshot is generated by NAV to support C/SIDE field formatting properties, such as **Blank When Zero**:



## Report triggers

The following screenshot shows the Report Designer's Report triggers available in a report:

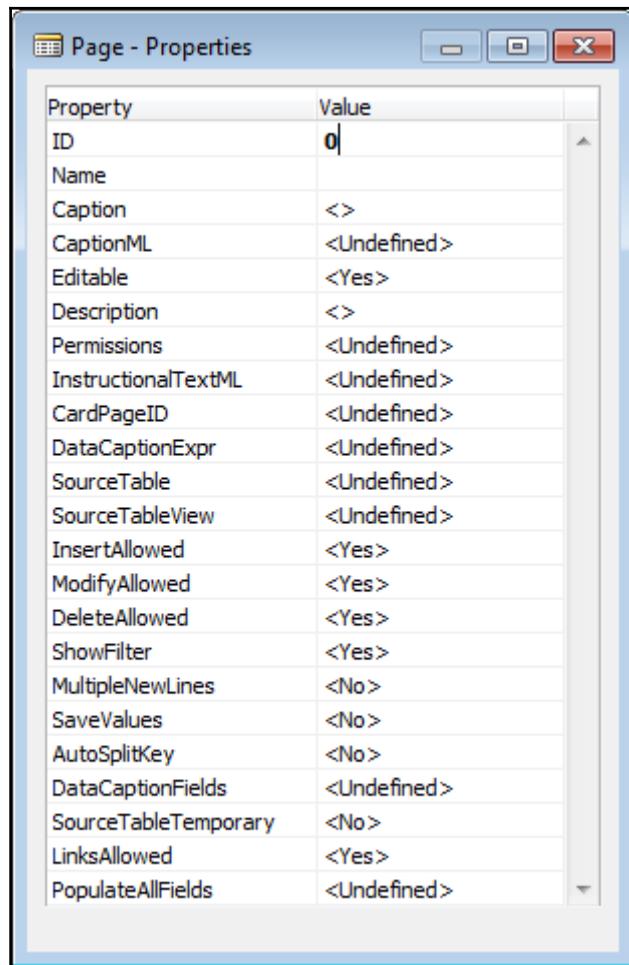


Descriptions for the Report Triggers follow:

- Documentation (): Documentation is technically not a trigger because it can hold no executable code. It will contain whatever documentation we care to put there. There are no format restrictions.
- OnInitReport (): This executes once when the report is opened.
- OnPreReport (): This executes once after the Request Page completes. All the DataItem processing follows this trigger.
- OnPostReport (): This trigger executes once at the end of all of the other report processing, if the report completes normally. If the report terminates with an error, this trigger does not execute. All the DataItem processing precedes this trigger.

## Request Page Properties

The Request Page properties are a subset of Page properties, which are covered in detail in Chapter 4, *Pages - the Interactive Interface*. Usually, most of these properties are not changed simply because the extra capability is not needed. An exception is the **SaveValues** property which, when set to Yes, causes entered values to be retained and redisplayed when the page is invoked another time. A screenshot of the Request Page properties follows:



## Request Page Triggers

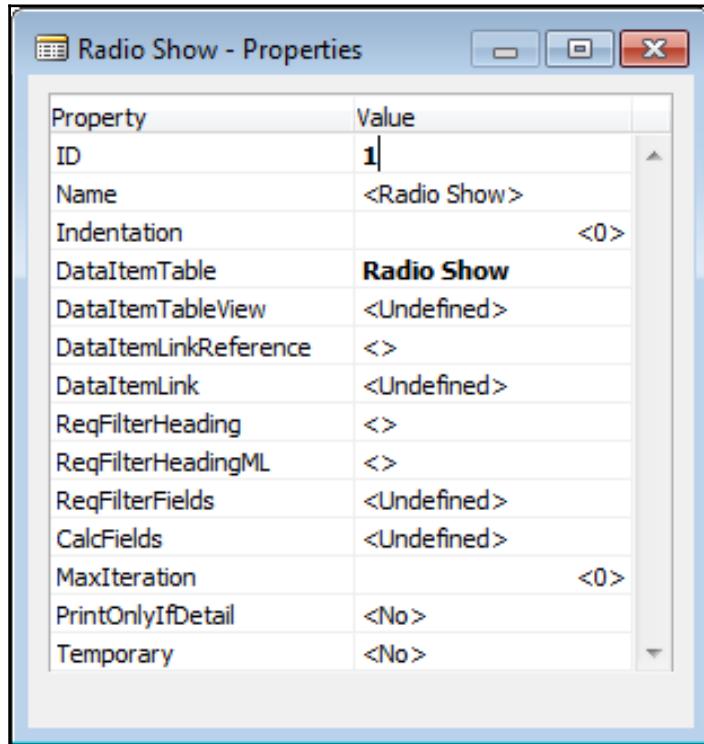
Request Pages have a full complement of triggers, thus allowing complex pre-report processing logic. Because of their comparatively simplistic nature, Request Pages seldom need to take advantage of these trigger capabilities. The following screenshot shows the triggers of a Request Page:

The screenshot shows the C/AL Editor interface with the title "Report 50000 Radio Show List - C/AL Editor". The code window displays a series of numbered trigger definitions, each preceded by a line number and a trigger name. The triggers listed are:

```
3  [ ] OnInit()
4  [ ]
5  [ ] OnOpenPage()
6  [ ]
7  [ ] OnClosePage()
8  [ ]
9  [ ] OnFindRecord(Which : Text) : Boolean
10 [ ]
11 [ ] OnNextRecord(Steps : Integer) : Integer
12 [ ]
13 [ ] OnAfterGetRecord()
14 [ ]
15 [ ] OnNewRecord(BelowxRec : Boolean)
16 [ ]
17 [ ] OnInsertRecord(BelowxRec : Boolean) : Boolean
18 [ ]
19 [ ] OnModifyRecord() : Boolean
20 [ ]
21 [ ] OnDeleteRecord() : Boolean
22 [ ]
23 [ ] OnQueryClosePage(CloseAction : Action None) : Boolean
24 [ ]
25 [ ] OnAfterGetCurrRecord()
```

## DataItem properties

The following screenshot shows the properties of a report DataItem:



The following are descriptions of frequently used report DataItems properties:

- **Indentation:** This shows the position of the referenced DataItem in the hierarchical structure of the report. A value of 0 (zero) indicates that this DataItem is at the top of the hierarchy. Any other value indicates the subject DataItem is subordinate to (that is nested within) the preceding DataItem.
- **DataItemTable:** This is the name of the NAV table assigned to this DataItem.
- **DataItemTableView:** This is the definition of the fixed limits to be applied to the DataItem (the key, ascending or descending sequence, and what filters to apply to this field).



If we don't define a key in the **DataItemTableView**, then the users can choose a key to control the data sequence to be used during processing. If we do define a key in the DataItem properties and, in the **ReqFilterFields** property, we do not specify any Filter Fieldnames to be displayed; this DataItem will not have a FastTab displayed as part of the Request Page. This will stop the user from filtering this DataItem, unless we provide the capability in C/AL code.

- **DataItemLinkReference:** This names the parent DataItem to which this one is linked.
- **DataItemLink:** This identifies the field-to-field linkage between this DataItem and its parent DataItem. 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. If no field linkage filter is defined, all the records in the child table will be processed for each record processed in its parent table.
- **ReqFilterFields:** This property allows us 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 DataItem, the user can choose any available field in the table to filter, regardless of what is specified here.
- **CalcFields:** This names the **FlowFields** that are to be calculated for each record processed. Because 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, we must instigate the calculation. This 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 the report will make through this DataItem. For example, we would set this to 7 to make the **virtual Date table** process one week's data.
- **PrintOnlyIfDetail:** This should only be used if this DataItem has a child DataItem, that is, one indented/nested below it. If **PrintOnlyIfDetail** is Yes, then controls associated with this DataItem will only print when data is processed for the child DataItem.
- **Temporary:** This specifies that a temporary table is supplying the dataset to populate the columns for this DataItem.

## DataItem triggers

Each DataItem has the following triggers available:



The data item triggers are where most of the flow logic is placed for a report. Developer defined functions may be freely added but, generally, they will be called from within these following three triggers:

- `OnPreDataItem()`: This is the logical place for any preprocessing to take place that can't be handled in report or data item properties or in the two report preprocessing triggers.
- `OnAfterGetRecord()`: This is the data **read and process** loop. The 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 dataitem. This is where we would typically access data in related tables. This trigger is represented on our report **Data Flow diagram** as any one of the boxes labeled Data Item processing Loop.
- `OnPostDataItem()`: This executes after all the records in this data item are processed, unless the report is terminated by means of a User Cancelor by execution of a C/AL BREAK or QUIT function or by an error.

## Creating a Report in NAV 2017

Because our NAV report layouts will all be developed in Visual Studio, our familiarity with NAV C/SIDE will only get us part way to having NAV report development expertise. We've covered most of the basics for the C/SIDE part of NAV report development. Now, we need to dig into the RDLC part. If you are already a Visual Studio reporting expert, you will not spend much time on this part of the book. If you know little or nothing about the RDLC layout tools, you will need to experiment and practice.

## Learn by experimentation

One of the most important learning tools available is experimentation. Report development is one area where experimentation will be extremely valuable. We need to know which Report layouts, control settings, and field formats work well for our needs and which do not. The best way to find out is by experimentation.

Create a variety of test reports, beginning with the very simple and getting progressively more complex. Document what you learn as you make discoveries. You will end up with your own personal Report development Help documentation. Once we've created a number of simple reports from scratch, we should modify test copies of some of the standard reports that are part of the NAV system.



We must always make sure that we are working on test copies, not the originals!

Some reports will be relatively easy to understand, and others that are very complex will be difficult to understand. The more we test, the better we will be able to determine which standard NAV report designs can be borrowed for our work, and where we are better off starting from scratch. Of course, we should always check to see if there is a Pattern that is applicable to the situation on which we are working.

## Report building - Phase 1

Our goal is to create a report for our WDTU data that will give us a list of all the scheduled radio show instances organized within Radio Show organized by **Radio Show Type**, as shown in the following screenshot:

Expanded	Data Type	Data Source	Name	I...
	DataItem	Radio Show Type	<Radio Show Type>	
	DataItem	Radio Show	<Radio Show>	
▶	DataItem	Playlist Header	<Playlist Header>	

An easy way to recreate the preceding screenshot is to simply type the letter D in the **Data Type** column, enter the target table number (in this case 50001, then 50000, then 50002) in the **Data Source** column, drop to the next line, and do it again. Then, save the new report skeleton as Report 50001, Shows by Type.

Before we go any further, let's make sure we've got some test data in our tables. To enter data, we can either use the pages we built earlier or, if those aren't done yet, we can just **Run** the tables and enter some sample data. The specifics of our test data aren't critical. We simply need a reasonable distribution of data so our report test will be meaningful. The following triple screenshot provides an example minimal set of data:

The image displays three separate tables from a reporting application, likely Power BI, showing data for radio shows and their types.

**Radio Show Type**

Code	Description
CALL-IN	Talk and Listener Interview
MUSIC	Music and Misc
NEWS	In-depth Stories
TALK	Mostly Talk

**Radio Show**

No.	Radio Show Type	Name	Run Time	Host Code	Host Name
RS001	TALK	CeCe and Friends	2 hours	CECE	CeCe Grace
RS002	MUSIC	Alec Rocks and Bobs	2 hours	ALEC	Alec Benito
RS003	CALL-IN	Ask Cole!	2 hours	COLE	Cole Henry
RS004	CALL-IN	What do you think?	1 hour	WESLEY	Wesley Knudsen
RS005	MUSIC	Quiet Times	3 hours	SASKIA	Saskia Panko
RS006	NEWS	World News	1 hour	DAAN	Daan White
RS007	ROCK	Rock Classics	2 hours	JOSEPH	Josephine Perisic

**Playlist Header**

No.	Radio Show No.	Description	Broadcast Date	Duration	Start Time	End Time
221	RS001	CeCe and Friends	5/22/2018	2 hours	6:00:00 AM	8:00:00 AM
222	RS003	Ask Cole!	5/22/2018	2 hours	8:00:00 AM	10:00:00 AM
223	RS002	Alec Rocks and Bops	5/22/2018	2 hours	10:00:00 AM	12:00:00 PM
224	RS006	World News	5/22/2018	1 hour	12:00:00 PM	1:00:00 PM
225	RS005	Quiet Times	5/22/2018	3 hours	1:00:00 PM	4:00:00 PM

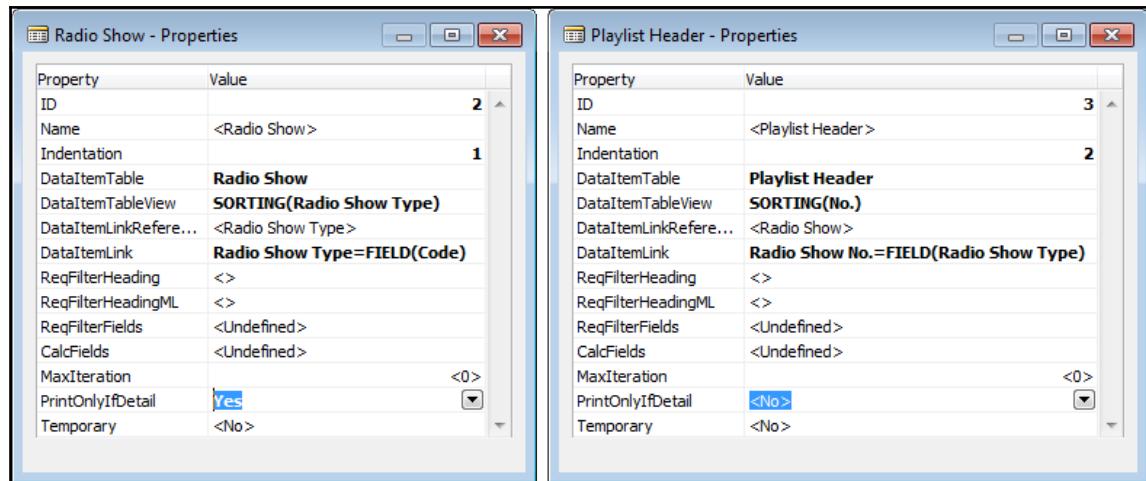
Since the C/SIDE part of our report design is relatively simple, we can do it as part of our Phase 1 effort. It's simple because we aren't building any processing logic, and we don't have any complex relationships to address. We just want to create a nice, neat, nested list of data.

Our next step is to define the data fields we want to be available for processing and outputting by Visual Studio. By clicking on the **Include Caption** column, we can cause the Caption value for each field to be available for use in Visual Studio. At this point, we should do that for all the data fields. If some are not needed in our layout design, we can later return to this screen and remove the check marks where unneeded. Please note that the **Name** column value will use the Visual Studio dataset Field Name to describe the data and its source as shown in the following screenshot:

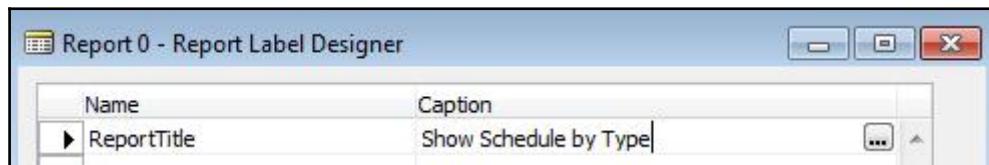
The screenshot shows the 'Report 50001 Shows by Type - Report Dataset Designer' window. It displays a table of data items with columns for Data Type, Data Source, Name, and Include Caption. The 'Include Caption' column contains checked checkboxes for most items. The table is organized into three main sections: 'Radio Show Type', 'Radio Show', and 'Playlist Header'. The 'Radio Show Type' section has three columns under 'Data Source': 'UserComment', 'Code\_RadioShowType', and 'Description\_RadioShowType'. The 'Radio Show' section has three columns under 'Data Source': 'No\_RadioShow', 'Name\_RadioShow', and 'RunTime\_RadioShow'. The 'Playlist Header' section has two columns under 'Data Source': 'BroadcastDate\_PlaylistHeader' and 'StartTime\_PlaylistHeader'.

E.. Data Type	Data Source	Name	Include Caption
<b>DataItem</b>	<b>Radio Show Type</b>	<Radio Show Type>	
Column	UserComment	UserComment	
Column	"Radio Show Type".Code	Code_RadioShowType	<input checked="" type="checkbox"/>
Column	"Radio Show Type".Description	Description_RadioShowType	<input checked="" type="checkbox"/>
<b>DataItem</b>	<b>Radio Show</b>	<Radio Show>	
Column	"Radio Show"."No."	No_RadioShow	<input checked="" type="checkbox"/>
Column	"Radio Show".Name	Name_RadioShow	<input checked="" type="checkbox"/>
Column	"Radio Show"."Run Time"	RunTime_RadioShow	<input checked="" type="checkbox"/>
<b>DataItem</b>	<b>Playlist Header</b>	<Playlist Header>	
Column	"Playlist Header"."Broadcast...	BroadcastDate_PlaylistHeader	<input checked="" type="checkbox"/>
Column	"Playlist Header"."Start Time"	StartTime_PlaylistHeader	<input checked="" type="checkbox"/>

Each of the subordinate nested DataItems must be properly linked to its parent DataItem. The **Playlist Header** DataItem is joined to the **Radio Show** DataItem by the **Playlist Header** "Radio ShowNo." field and **Radio Show** "No." field. The **Radio Show** DataItem is joined to the **Radio Show Type** DataItem by the **Radio Show** "Radio ShowType" field and the **Radio Show Type** "Code" field. The Radio Show portion of the dataset returned is limited by setting the **PrintOnlyIfDetail** value to Yes as shown in the following screenshot. This choice will cause the **Radio Show** record not be sent to output for reporting if no subordinate **Playlist Header** records are associated with that **Radio Show**:



The other data that we can pass from the Report Designer to Visual Studio are the labels. Labels will be used later as captions in the report and are enabled for multi-language support. Let's create a title label to hand over the fence to Visual Studio. Go to the **View** menu, choose **Labels** to open **Report Label Designer**, and enter the following label definition:

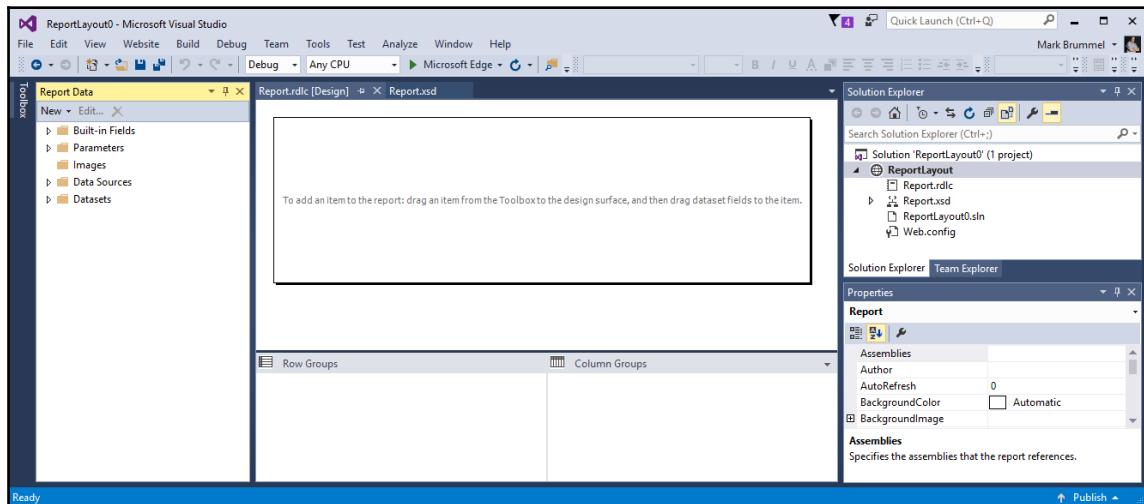


Now that we have our C/SIDE dataset definition completed, we should save and compile our work before doing anything else. Then, before we begin our Visual Studio work, it's a good idea to check that we don't have some hidden error that will get in our way later. The easiest way to do that is just to **Run** what we have now. What we expect to see is a basic Request Page display allowing us to run a report with no layout defined.

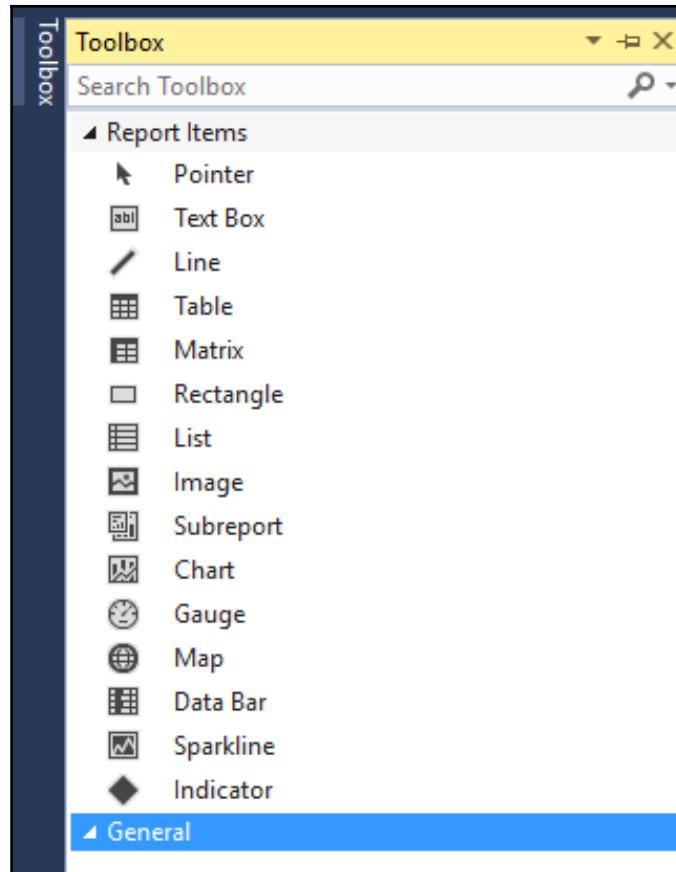
## Report building - Phase 2

As mentioned earlier, there are several choices of tools to use for NAV report layout development. The specific screen appearance depends somewhat on which tool is being used.

To begin our report development work in Visual Studio, we must have our C/SIDE dataset definition open in Design mode. Then navigate to **View | Layout** to open up Visual Studio. If we have previously done Visual Studio development work on this report and saved it, that work will be displayed in Visual Studio, ready for our next effort. In this case, since we are just starting, we will see the following screenshot:



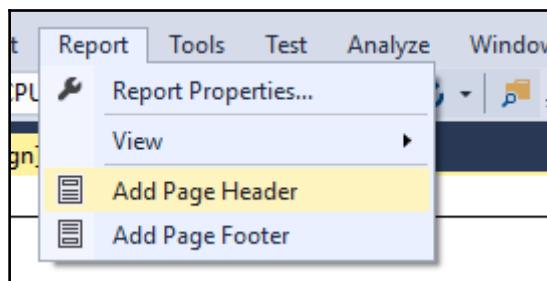
On the left of the screen is a tab labeled **Toolbox**. When we click on that tab, the window shown in the following screenshot will be displayed:



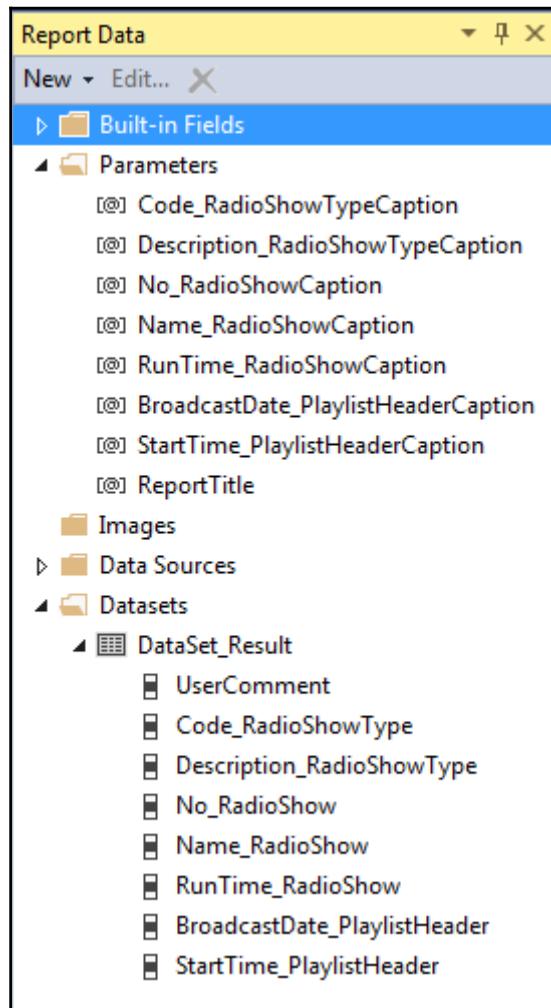


Before we start on our sample report Visual Studio work here, study some of the information on report design in **NAV Developer and IT Pro Help**. In particular, you should review *Walkthrough: Designing a Customer List Report* (<https://msdn.microsoft.com/en-us/dynamics-nav/walkthrough--designing-a-customer-list-report>) and *Walkthrough: Designing a Report from Multiple Tables* (<https://msdn.microsoft.com/en-us/dynamics-nav/walkthrough--designing-a-report-from-multiple-tables>). These Walkthrough scripts will provide additional helpful information.

Since we want a page header, let's start by adding that to our layout. Right-click on the **Report** menu, then click on **Add Page Header** as shown in the following screenshot. Note that if we want the Report Header to appear on each page, we must use the SetData and GetData functions in association with hidden text boxes (see *How to: Print Report Header Information on Multiple Pages*- <https://msdn.microsoft.com/en-us/dynamics-nav/how-to--print-report-header-information-on-multiple-pages>):



Next, we will add some fields to the Page Header. First, we will expand two of the categories in the **Report Data** panel that is on the left side of our layout screen. The two categories we want to expand are parameters, which contains our captions we checked, plus any defined Labels, and DataSets, which contains the data elements passed from our Report Designer, all seen in the following screenshot:



When we expand the DataSet Result, we can see the importance of our data field Names being self-documenting. Having done so makes it much easier to remember what we defined in our DataSet and with which DataItems they are associated. This habit will be especially important when we have multiple fields with the same name from different DataItems, such as Code, Description, or Amount.

The DataSet represents a record format in which all defined fields are present for all DataItems. This is a classic **flat file** format where the hierarchical data structure is flattened out to make it easier to pass from one environment (NAV) to another; in this case, Visual Studio.

Now we will add the report title that we defined as a label earlier in the Report Designer. Drag the **Report Title** field over to the Page Header workspace and position it where we want it to be. Now, expand the **Built-in Fields** section in the **Report Data** panel. We will add some of these fields, such as User ID, Execution Time, and Page Number to our Page Header. We will position these fields wherever we think appropriate.

At this point, it's a good time to save our work and test to see what we have done so far by performing the following steps:

- Click on the top-left **File** menu option, or the disc icon, or *Ctrl + S* to save the design as RDLC.
- Click on the **File** menu option, then on **Exit** or the **X** box at the top right of the screen (or *Alt + F4*) to return to the Report Designer.
- Exit the Report Dataset Designer.
- Respond **Yes** to the **The layout of report id 50001 has changed...Do you want to load the changes?** question.
- Then we'll see a familiar window asking us **Do you want to save Report 50001 Shows by Type?**. Save and compile the report.

Now **Run** the report. The **Preview** output should look something like in the following screenshot:

The screenshot shows a 'Print Preview' window. At the top left is a 'Print Preview' button. Below it is a dropdown arrow. The main area has a title 'Shows by Type'. Underneath is a toolbar with navigation icons (back, forward, search, etc.), a date/time '10-12-2016 17:34:48', a 'Show Schedule by Type' button, and a '100%' zoom level. Below the toolbar is a section labeled 'DESKTOP-AHMVIT0\MARKB'. A table follows with columns 'Code' and 'Description'. The single row contains 'CALL-IN' and 'Talk and Listener Interview'.

Code	Description
CALL-IN	Talk and Listener Interview

Not especially impressive, but not bad if this is our first try at creating a NAV 2017 report.

The wrapped report fields show us that we need to make those text boxes wider. This would be a good point to do some experimenting with positioning or adding other heading information, such as Page, in front of the page number. When we highlight a field, the properties of that field are displayed, and are available for modification in the **Properties** window. A few simple changes and our Report Heading could look like the following screenshot:

The screenshot shows a 'Print Preview' window similar to the previous one, but with a few changes. It includes a 'Page' indicator showing 'Page 1' at the bottom right. The rest of the interface and data are identical to the first screenshot.

We could even experiment with various properties of the heading fields, choosing different fonts, bolding, colors, and so on. As we only have a small number of simple fields to display (and could recreate our report if we have to do so), this is a good time to learn more about some of the report appearance capabilities that Visual Studio provides.

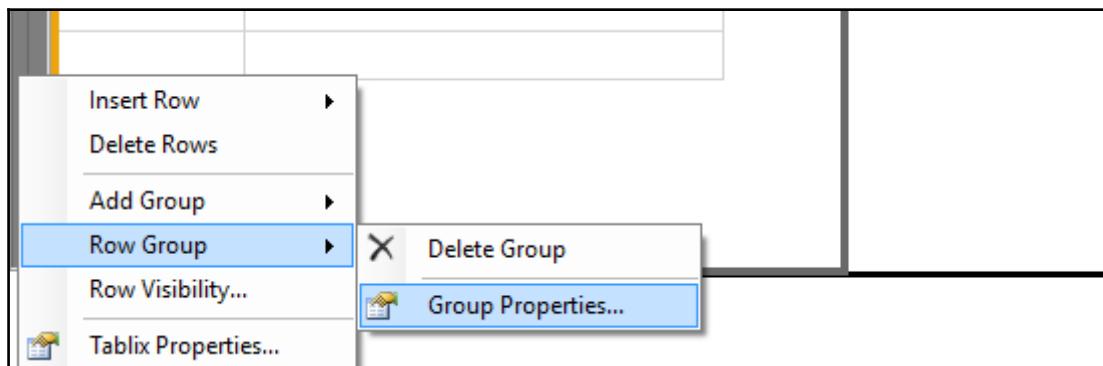
## Report building - Phase 3

Finally, we are ready to layout the data display portion of our **Radio Shows by Type** report. The first step of this phase is to layout the fields of our controlling DataItem data in such a way that we can properly group the subordinate DataItems' data.

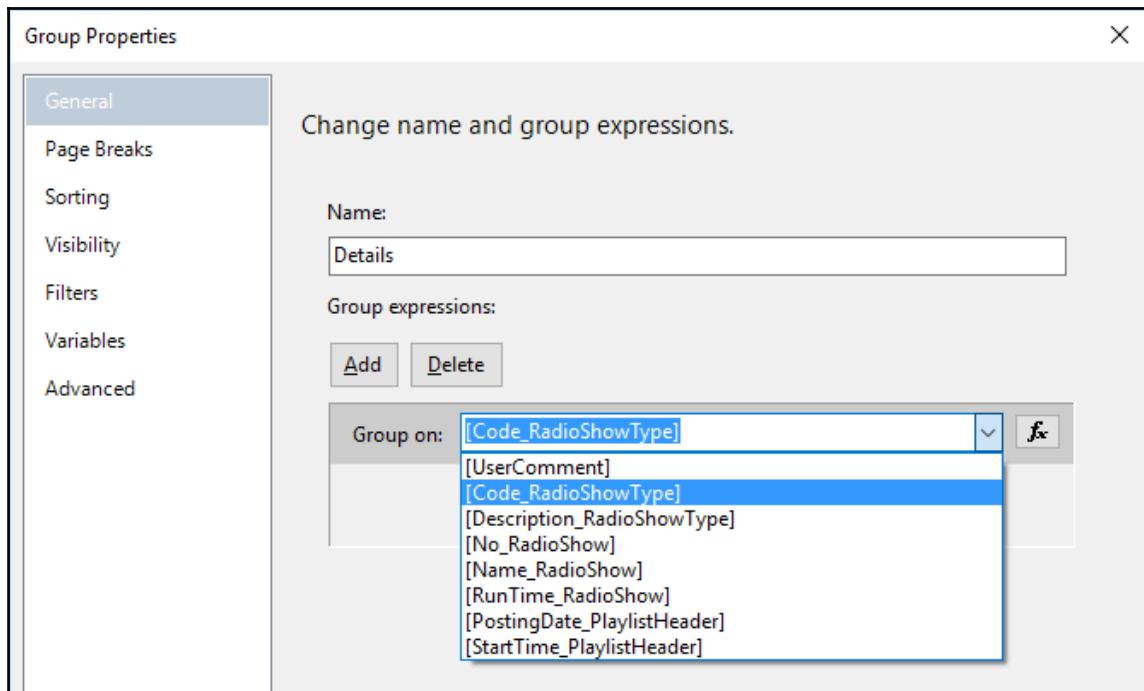
Once again, Design the report and navigate to **View | Layout** so that the Visual Studio Report layout screen is displayed with the **Insert** ribbon visible. We are now done with the Page Heading and, from here on, all our work will be done in the body of the report design surface.

Click on the **List** icon in the **Insert** ribbon and drop a List control in the body of the report design surface. Position the control at the top left of the layout body. Because we'll define six layout lines to hold the necessary data and header controls, we may want to expand the List control and work area (though this can be done later, as needed). Follow these steps:

1. Click on the List control so that a shaded area appears to the top and left of the control.
2. Right click on the shaded area and select **Tablix Properties**.
3. Right click on the drop-down arrow in the **Dataset Name** box to set the Dataset name value to **DataSet\_Result**. This causes the List control to reference our incoming NAV data.
4. Click on **OK** to save the new **Tablix** property setting.
5. Right click again in the List control's shaded area and select **Row Group**, then **Group Properties** as shown in the following screenshot:

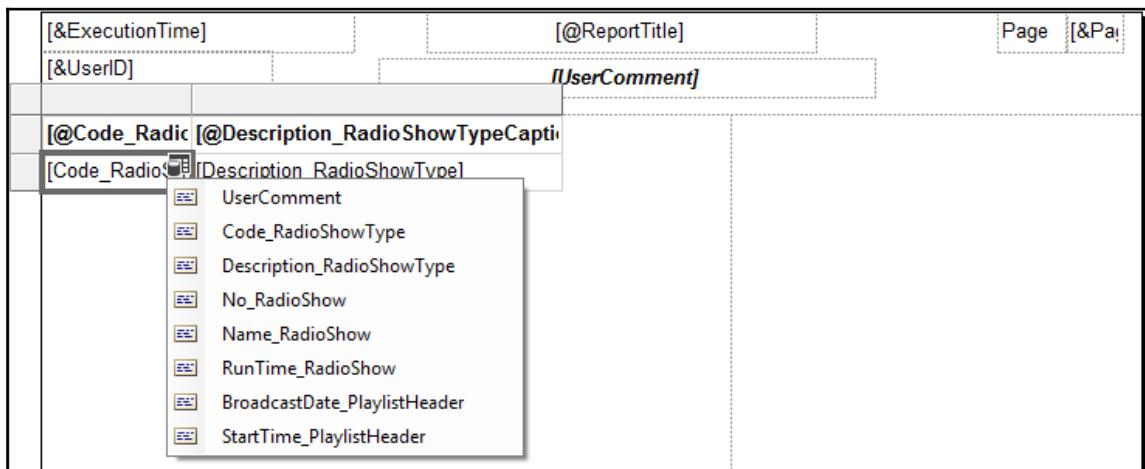


6. In the **Group Properties** screen, select the **General** tab and in the **Group expressions** area click on **Add**, then in **Group on:** select **[Code\_RadioShowType]**, because we want our data output grouped by the Type of Radio Show as shown:



7. Click on **OK** to return to the report layout design surface. Click on the **Table** control icon, choose **Insert Table** and click on the List control to place the new Table. This will be the container for our **Radio Show Type** DataItem fields.
8. The empty control will start with two rows and three columns. As we'll only display two data fields (code and description), we only need two columns. Highlight the rightmost column (click on the body so that a shaded area appears on the left and top of the control, then click on the shaded area above the column), right click on that shaded area and chose the **Delete Columns** option.
9. Because this section of our report acts as a heading to subordinate sections, we will delete the Data row from the Table control and replace it with a second **Header** row. Once again, click on the shaded area to highlight the Data row, select the **Delete Rows** option, respond **OK** to the confirmation message, highlight the **Header** row, and insert a second **Header** row. We will now have two rows and two columns of Text boxes (cells) ready for our data.

10. From the **Parameters** data list in the **Report Data** panel, grab the **Code\_RadioShowTypeCaption** and drop it into the top-left table cell. Drag the **Description\_RadioShowTypeCaption** and drop it into the top-right table cell. Stretch out the cells so the data is likely to fit without wrapping to a second line.
11. Our next step is to place some data fields in the Table control. Click on the lower-left cell of the Table control. The field list icon will display. Click on the icon and we will see all the fields in the **Dataset\_Result**:



12. Select **Code\_RadioShowType** for the bottom-left cell and **Description\_RadioShowType** for the bottom-right cell.

We are at another good point to save and check our work. If you are one of those people who like to do as much as possible from the keyboard, *Ctrl + S*, to save the RDLC; *Alt + F4*, to exit Visual Studio; *Esc*, to exit the Report Designer; *Enter*, to load the changed RDLC; *Enter*, to save and compile; and then *Alt + R*, to run the report. Now, you'll likely have to use your mouse to respond to the Request Page.

Don't worry about the vertical and horizontal layout of our output as long as our results make sense. We can fix the layout alignments later. When we run our report, we should get as many instances of the Radio Show type printed from our test data as we have Types of entries selected from the **Playlist Header** table based on any filters we applied.

Assuming our output looks pretty much like we expected (a simple list of Show Types with column headings), we can move on to the next layer of definition. This time, we will define how the Radio Show data fields will be shown, including the fact that this set of data is grouped subordinate to the **Type** field.

Insert another Table control in the report design surface inside the list control. Position this control below the last Table control in a position that will show its relation to the Type data records. Usually, this will involve indenting.

Click on the control to cause the shaded outer area to display. Highlight the **Header** row and delete it. Highlight the **Data** row, right click on it, and choose **Insert Row - Inside Group** (Above or Below doesn't matter this time). Now, to confirm if our List control is associated with the **DataSet\_Result**, perform the following steps:

1. Right click on the List control so that a shaded area appears at the top and left of the control.
2. Right click on the shaded area and select **Tablix Properties**.
3. The Dataset name box should contain the value **DataSet\_Result**.
4. Click on **OK** to save the **Tablix** property setting.

In the bottom row of our second Table control, cause the field list icon to display, then choose a field for each of the three cells (`No.`, `Show_Description`, and `Run_Time`). In the top row, we could choose **Parameter** captions as we did earlier; but we may want captions that are different than those that came across from Report Designer. We could have used Labels here if we had defined them in Report Designer or we can just do the simple thing--type in the column headings we want (`Show No.`, `Name`, `Run Time`), overwriting the default headings (doing this will not yield multi-language compliant captions). Highlight the middle column (`Name`) and stretch it out so a long name will display on one line rather than wrapping. Now, once again, it's time to save and test.

Our third (and final) set of data for this report will hold data from the **Playlist HeaderDataItem**. Right click on the grey area of the top row of the second Table control. Select **Row Group | Group Properties**. In the **Group Properties** screen, select the **General** tab; in the **Group expressions** area, click on **Add**; then, in **Group on:**, select **[Code-RadioShow]** because we want the next set of data output grouped by the **Radio Show Code**. When we return to the layout screen, we will see that the grey area next to the rows for this table will have changed from the data icon to a group brace icon that includes both of what were previously marked as data rows.

Right click on the grey area for the bottom row, choose **Insert Row | Inside Group - Below**. Repeat so that there are now two empty rows at the bottom of the second table area. Fill the rightmost two columns of these two rows. In the top of these two rows, insert captions from the **Parameters** list for **PostingDate** and **StartTime**. In the bottom row, use the field lists to insert data elements for **PostingDate** and **StartTime**. The final layout should look similar to the following screenshot:

The screenshot shows the Visual Studio RDLC report designer. At the top, there are three text boxes: '[&ExecutionTime]', '[@ReportTitle]', and '[Page]'. Below them is another text box: '[&UserID]'. A large table structure is centered. It has a header row with four cells containing '[@Code\_Radio]' and '[@Description\_RadioShowTypeCaption]'. Below this is a row with two cells: '[Code\_RadioSh]' and '[Description\_RadioShowType]'. The main body of the table is a 3x2 grid. The first column contains '[@No\_RadioSt]', '[@Name\_RadioShowCa]', and '[@BroadcastDate\_Playli]'. The second column contains '[@RunTime\_Radi]' and '[RunTime\_Radi]'. The third column contains '[@StartTime\_F]' and '[StartTime\_Play]'. The fourth column contains '[BroadcastDate\_PlaylistH]' and '[StartTime\_Play]'. The entire table is enclosed in a black border.

Finally we will save our RDLC code (*Ctrl + S*), exit Visual Studio (*Alt + F4*), exit CSIDE RD (*Esc, Enter*), save and compile (*Enter*). Let's **Run** our report and see what we've got. While this report is not beautiful, it is serviceable, especially for a first try. One thing that we should improve before we show it to very many people is to make the formatting more attractive.

A couple of very simple changes would be to make the heading rows bold (to stand out) and format the date and time fields so they would show properly (not showing as SQL DateTime fields). Easy ways to do this in the layout screen are as follows:

1. Click on the table so that the grey outline displays. Highlight the heading row. Click on the bold icon on the top ribbon. The text for the row should now show bolded.
2. Right click on the cell for the date field. Choose **Textbox Properties**. From the list of property categories, choose **Number**, then choose **Date** and the preferred Date format. Do essentially the same thing for the time field.

The result after some minor formatting should look similar to the following screenshot:

The screenshot shows a software application window titled "Shows by Type". The top bar includes a "Print Preview" button, a dropdown menu, and standard window controls. The main area displays a schedule for January 25, 2018, with the following details:

17-12-2016 15:29:24      Show Schedule by Type      Page 1

DESKTOP-AHMVIT0\MARKB

**CALL-IN**

Code	Description
CALL-IN	Talk and Listener Interview

No.	Name	Run Time
RS003	Ask Cole!	2 hours
Broadcast Date		Start Time
	donderdag, 25 januari, 2018	10:00

No.	Name	Run Time
RS004	What Do You Think!	1 hour
Broadcast Date		Start Time
	donderdag, 25 januari, 2018	12:00

**MUSIC**

Code	Description
MUSIC	Music and Misc

No.	Name	Run Time
RS002	Alec Rocks and Bobs	2 hours
Broadcast Date		Start Time
	donderdag, 25 januari, 2018	08:00

## **Modifying an existing report with Report Designer or Word**

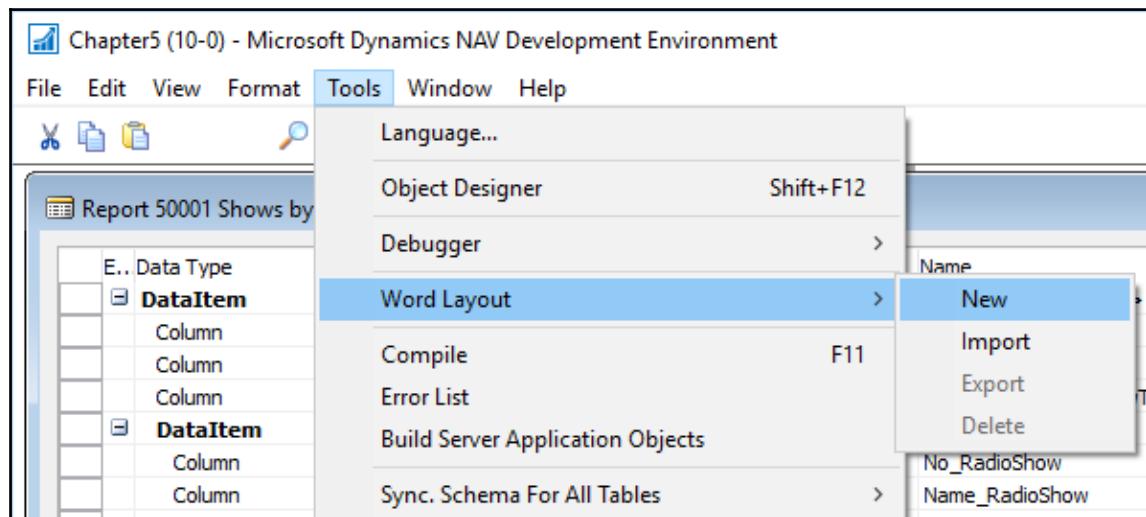
The basic process we must follow to modify an existing report is the same whether the report is one of the standard reports that comes with NAV 2017 or a custom report that we are enhancing in some way. An important discipline to follow in all cases where we are modifying a report that has been in production is *NOT* to work on the original, but on a copy. In fact, if this is a standard report we are customizing, we should leave the original copy alone and do all our work on a copy. Not only is this safer because we will eliminate the possibility of creating problems in the original version, but it will make upgrading easier later on. Even when working on a new custom report, it is good practice to save intermediate copies for backup. This allows returning to a previous working step should the next development step not go as planned.

While it is certainly possible in NAV 2017 to add a new layout to an existing dataset without disturbing the original material, the potential for a mistake creating a production problem is such that best practice dictates working on a copy, not the original.

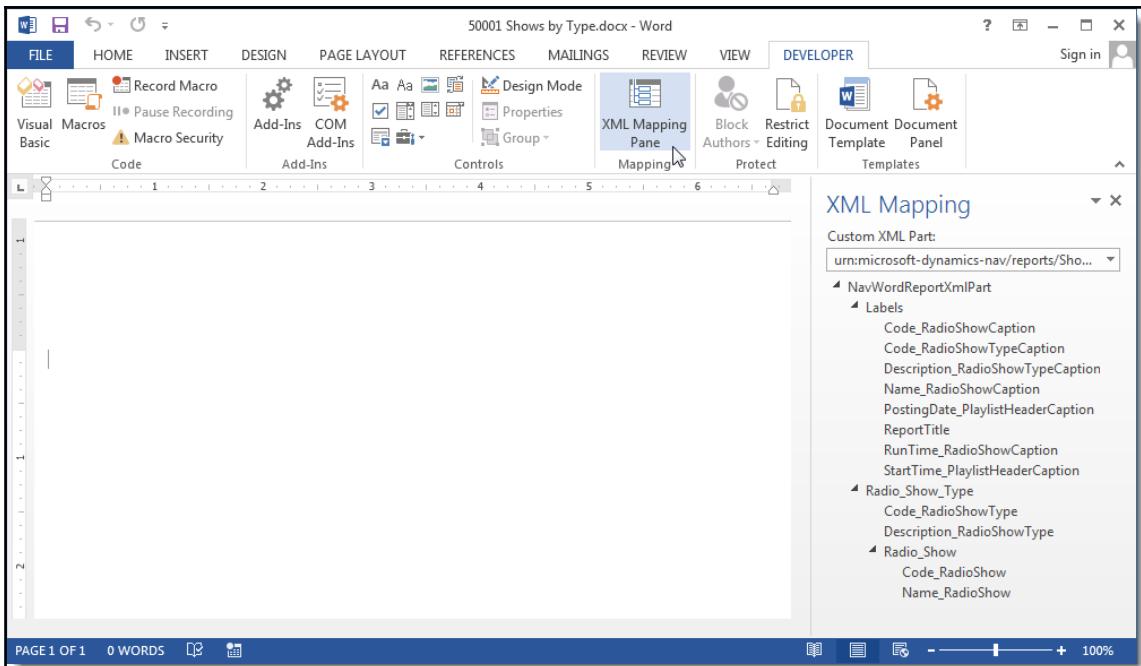
Just like report construction, report modification requires use of two toolsets. Any modification that is done to the processing logic or the definition of the data available for report output must be done using the C/SIDE Report Designer. Modification to the layout of a report can be done using Visual Studio, just like we've been doing, or the SQL Server Report Builder or, when a Word layout is available, using Microsoft Word. Each report can have either or both an RDLC and a Word layout, but for those reports unlikely to need modification by a non-programmer, a Word layout will probably not be worth the effort required to create it.

All NAV 2017 report layouts can be modified by a developer using Visual Studio because all standard reports have been developed with RDLC layouts. A small number of standard reports also have Word layouts available in the initial distribution of NAV 2017. These include Reports 1304, 1305, 1306, and 1307. It is likely that future releases of NAV will have additional report layouts available in Word format. In the meantime, if we want other reports, standard or custom, to have Word layout options available, we will have to create those ourselves. The primary advantage of having Word layout options for reports is to allow modifications of the layouts by a trained user/developer using only Word. As the modifications must still conform to good (and correct) report layout practices, appropriate training, careful work, and considerable common sense are needed to make such modifications, even though the tool is Microsoft Word.

If we decided we want to have a Word version of the layout for our report 50001 – Shows by Type, we would proceed along the following lines. First, we would create a Word layout in the form of a Word document. We can either start in Word, then import the resulting template document into our report, or we can start in the C/SIDE Report Designer, then create a blank Word document where we will build our Word format. This is done by opening the Report Designer as before, then clicking on **Tools | Word Layout | New** as shown in the following screenshot. This will immediately create an empty Word layout inside our report object:



The next step is to export that **Word Layout** so we can work on it in Word. Exporting is done in the same place as creating the new layout, this time clicking on **Tools | Word Layout | Export** as shown in the following screenshot. After saving the exported Word document, we proceed to Word and open it. We must have the **Developer** tab enabled on the Word command ribbon. After opening the **Word Layout** document, click on the **Developer** tab, then the **XML Mapping Pane** icon. As we might expect, this will open up the XML Mapping showing the XML data structure available from the report. When we expand the XML groups, we can see the same type of data list we earlier saw in the Visual Studio report layout. From this point, we can use standard Word capabilities combined with our report's Caption, Label, and DataSet fields to create a report layout:



Additional information on report layout capabilities and management are available in both the system Help and online in YouTube videos. Refer to *Designing RDLC Report Layouts* (accessed from the Microsoft Dynamics NAV Development Environment, for example <http://msdn.microsoft.com/en-us/dynamics-nav/designing-rdlc-report-layouts> and <https://msdn.microsoft.com/en-us/dynamics-nav/designing-word-report-layouts>). For applicable YouTube videos, search using combinations of keywords such as How do I:, NAV, Word, Report, and so on. An add-on tool to create NAV Report Word layouts, Jet Express for Word, is also available.

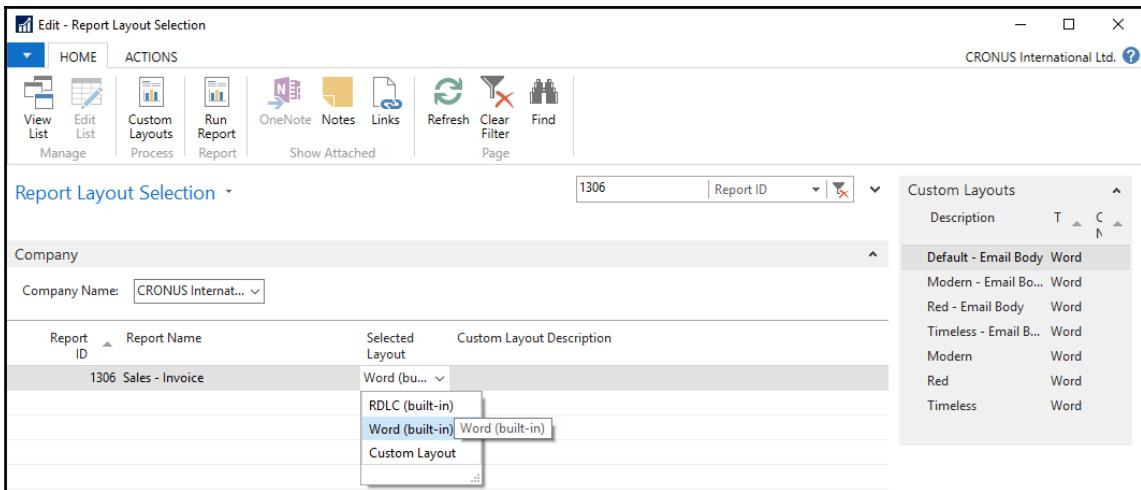
When our work on the report layout is complete, we will save the Word document in the normal fashion. At this point, we will return to the Report Designer and click on **Tools | Word Layout | Import** to import the Word layout template we just created/modified. The new layout can be tested from the RoleTailored Client. First, we must add the layout to the list of the available **Custom Report Layouts**. From the RoleTailored Client, we will use the Search box in the upper right-hand corner; search on **Layout** and select **Custom Report Layouts**. The following screen will display:

The screenshot shows the Microsoft Dynamics NAV interface for managing custom report layouts. The title bar reads "Edit - Custom Report Layouts - 50001 Shows by Type". The ribbon has tabs for HOME, ACTIONS, and REPORT. The ACTIONS tab is selected, showing icons for New, Copy, View List, Edit List, Delete, Import Layout, Export Layout, Edit Layout, Run Report, Show as List, Show as Chart, OneNote, Notes, Links, Show Attached, Find, and Refresh. A search bar at the top right contains the text "50001". The main area displays a table of report layouts:

Report ID	Report Name	Description	Company Name	Type
50001	Shows by Type	Copy of Built-in layout		RDLC
50001	Shows by Type	Copy of Built-in layout		Word

By following the guidance provided in *Managing Report and Document Layouts*-<https://docs.microsoft.com/en-us/dynamics365/financials/ui-manage-report-layouts> (accessed from the Microsoft Dynamics NAV client), we can maintain a list of the available custom report layouts, and add our new layout to the list.

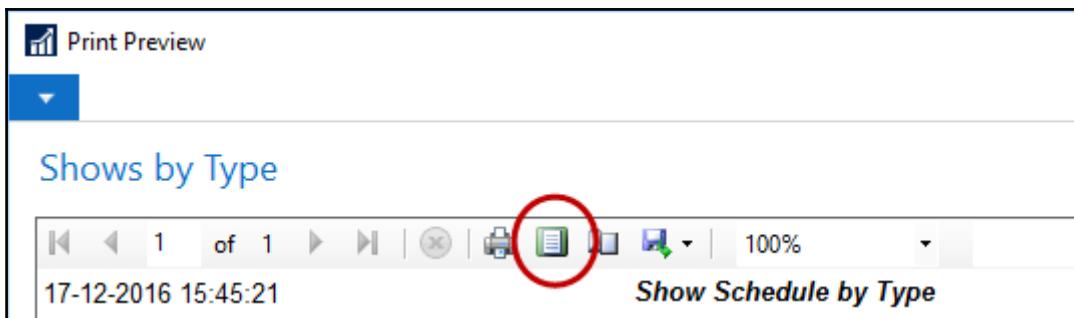
Finally, we can do our testing by choosing a report layout and running the report from the **Report Layout Selection** screen as shown in the following screenshot. This screen is also accessed by searching on Layout and selecting **Report Layout Selection**. We can choose either a standard (built-in) layout (RDLC or Word) or **Custom Layout**. The choice is stored in the NAV database and can be specific to individual companies and database tenants. This screen as well as the **Custom Report Layouts** screen are accessed through the RTC so that users with appropriate permissions can maintain and assign applicable report layouts:



## Runtime rendering

When NAV outputs a report to screen, hardcopy, or PDF, it will render using the printer driver for the currently assigned printer. If we change the target printer for a report, the output results may change depending on the attributes of the drivers.

When we preview a report, by default, it will be displayed in an interactive preview mode. This mode will allow us 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 we will get if we print it. If we click on the Print Layout icon button (circled in the following screenshot), then the printer layout version of the report will be displayed:



In most cases, the display on screen in the **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 some type of 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 it's limited to print-applicable properties. Properties, such as decimal formatting, are inherited, but as we saw with our date and time fields, not all formatting is inherited. Remember, if the property is explicitly defined in the table, it cannot be less restrictively defined elsewhere. This is one of the reasons why it's so important to focus on table design as the foundation of the system.

## **Interactive report capabilities**

NAV 2017 reports can have interactive features enabled. Of course, these features are only available when the report is displayed in preview mode; once the report is printed whether to a PDF, Word, Excel, or an output device, the interactive capabilities are no longer present.

## **Interactive sorting**

Among the useful interactive reporting features are interactive sorting and data expand/collapse. Two standard reports that are examples of the interactive sort feature are the Customer - Top 10 List (Report 111) and the Customer - Summary Aging (Report 105). We'll take a look at Report 111 to see how NAV does it.

Since we'll open the Report in the Designer, the possibility exists that, through an unlucky combination of keystrokes, we could accidentally change this production report; the first thing we want to do is make a copy for our inspection. Open Report 111 in the C/Side Report Designer, then click on **Save As** to Report object 50111 with a different name, such as *Customer - Top 10 ListTest*. Once that is done, we can safely do almost anything we want to Report 50111 because we can simply delete the object when we are done with it.

First, let's **Run** Report 50111 or 111, since they will look the same. At the top of four of the report columns, we will see an up/down arrowhead icon representing an interactive sort control, which is highlighted in the following screenshot:

**Customer - Top 10 List**

Period: CRONUS International Ltd.

17-12-2016  
Page 1  
DESKTOP-AHMVIT0\MARKB

Ranked according to Sales (LCY)

No.	Name	Sales (LCY)	Balance (LCY)
10000	The Cannon Group PLC	17.100,96	168.364,41
49858585	Hotel Pferdesee	14.395,75	14.395,75
43687129	Designstudio Gmunden	13.732,60	13.732,60
47563218	Klubben	11.772,20	11.772,20
20000	Selangorian Ltd.	6.510,64	96.049,99
30000	John Haddock Insurance Co.	6.142,90	349.615,40

Now open test report 50111 in the Report Designer, then **View | Layout** for Visual Studio review. Highlight the **BalanceLCY\_CustomerCaption** textbox, as shown in the following screenshot. Show the properties; choose the **Interactive Sorting** tab. As we can see here, interactive sort options are set for this column to sort the detail by the value of **BalanceLCY\_Customer**. If we look at the properties of the other three columns that have interactive sorting enabled, we will see similar setups to those shown in the following screenshot:

**CustomerCaption**

**General**

- CanGrow: True
- CanShrink: False
- Name: BalanceLCY\_CustomerCaption
- ToolTip:

**Interactive Sort**

**UserSort**

- SortExpression: =Fields!BalanceLCY\_Customer
- SortExpressionScope:
- SortTarget:
- ComponentMetadata:

**Lists**

- ListLevel: 0
- ListStyle: None

## Interactive Visible / Not Visible

As an experiment, we'll add a toggle to the rightmost column of our test report 50111 to make it visible or not visible at the user's option. In most cases, this feature would not be controlled by the user but by a parameter, perhaps one tied to the user's login. We can set the **ToggleItem** property in the **Visibility** tab as shown in the following screenshot to the variable we want to use as a toggle for visibility control of the data that will be visible/hidden. This time, we choose the **Customer No.** column and set the **Balance** column to initially be **Visible** when the report is first run:

The screenshot shows the Report Designer interface with a table structure. The last column of the table has its properties open, specifically the 'Visibility' tab. The 'ToggleItem' property is set to 'No\_CustomerCaption', which is highlighted with a red box. Other settings in the 'Visibility' tab include 'Hidden' (False), 'InitialToggleState' (True), and 'Fill', 'Font', etc.

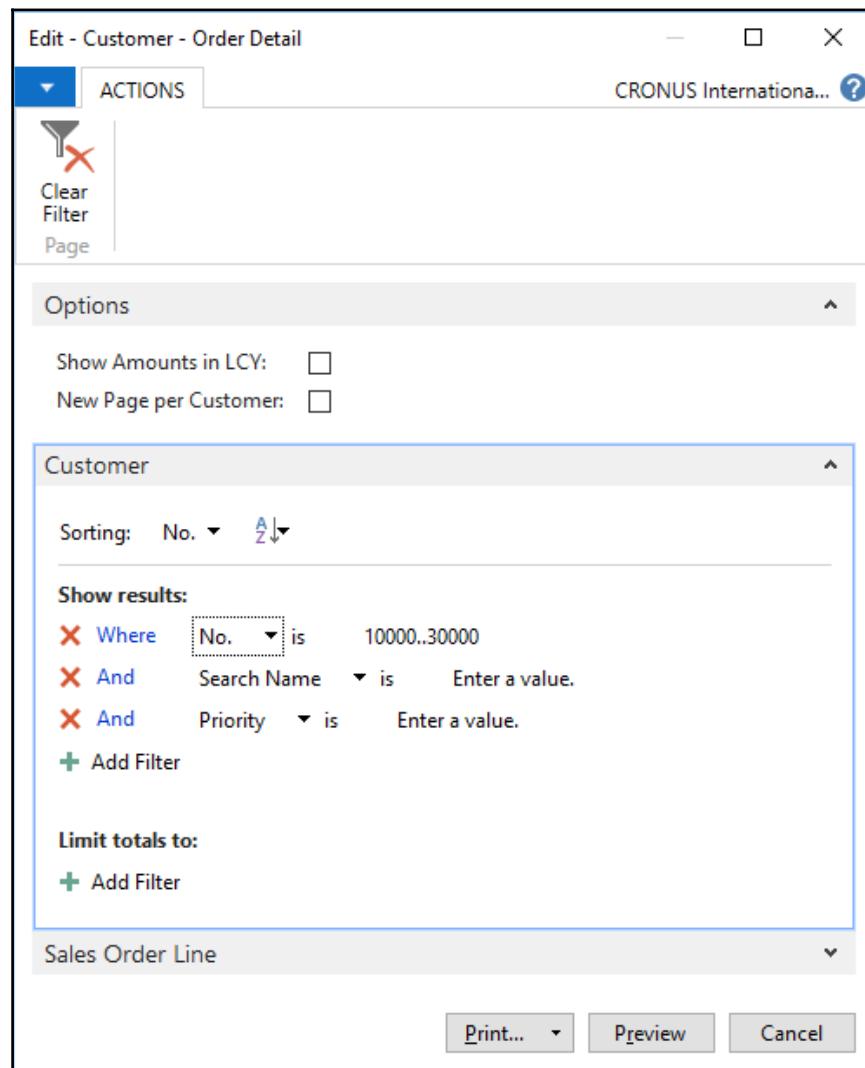
Save the RDLC layout and exit Visual Studio, then exit the Report Designer and save, compile, and **Run** the modified report. We will see the report display like that at the top of the following screenshot pair. If we click on the + (plus sign) icon located above the **No.** column caption, the + will change to - (minus sign) icon located next to the **No.** column caption, and the rightmost column becomes not visible, like that shown at the bottom of the following screenshot pair. If we click on the - (minus sign) icon, the **Balance (LCY)** column will be visible again as shown in the top screenshot of the pair:

The screenshot shows two versions of a report table. The top version has a visible 'Balance (LCY)' column with a plus sign icon above the 'No.' column caption. The bottom version has a hidden 'Balance (LCY)' column with a minus sign icon above the 'No.' column caption. The table data is identical in both cases.

No.	Name	Sales (LCY)	Balance (LCY)
10000	The Cannon Group PLC	17,100.96	168,364.41
43687129	Designstudio Gmunden	14,498.04	14,498.04
49858585	Hotel Pferdesee	14,450.00	14,450.00
47563218	Klubben	11,772.20	11,772.20

## 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, which is based on one of the standard system reports, the **Customer - Order Detail** report, Report 108:



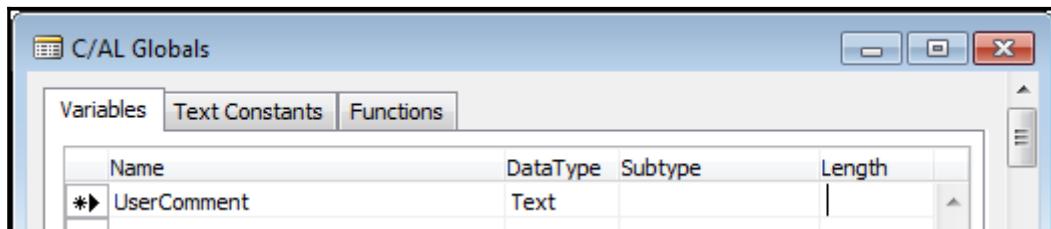
There are three FastTabs in this page. The **Customer** and **Sales Order Line** FastTabs are tied to the data tables associated with this report. These FastTabs allow the user to define both data filters and Flow Filters to control the report processing. The **Options** FastTab exists because the software developer wanted to allow some additional user options for this report.

## Add a Request Page option

Because we have defined the default sort sequences (**DataItemTableView**), except for the first DataItem, and we have not defined any Requested Filters (**ReqFilterFields**), the default Request Page for our report has only one DataItem FastTab. Since we have not defined any processing options that would require user input before the report is generated, we have no **Options** FastTab.

Our goal now is to allow the user to optionally input text to be printed at the top of the report. This could be a secondary report heading, instructions on interpreting the report, or some other communications to the report reader. To add this capability, perform the following steps:

1. Open **Report 50001** in the C/SIDE Report Designer.
2. Access the **C/AL Globals** screen through **View | C/AL Globals**.
3. Add a Global Variable named **UserComment** with a **DataType** of **Text**. We will not define the **Length** field as shown in the following screenshot; this will allow the user to enter a comment of any length:



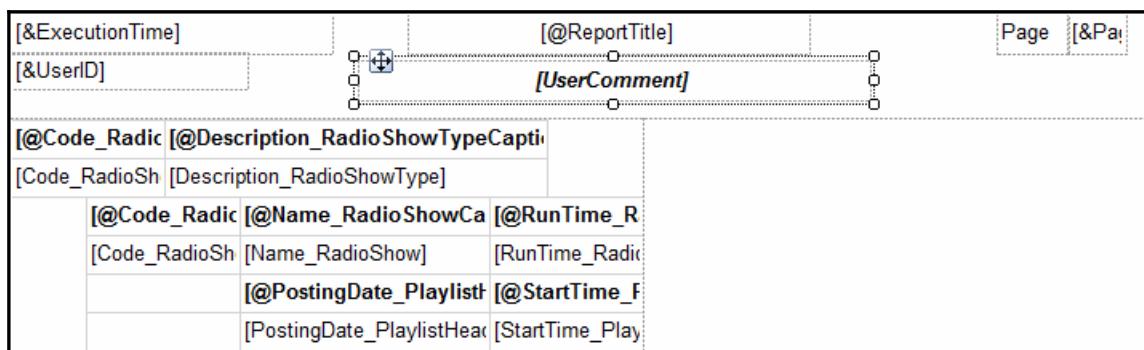
4. Add this variable as a data **Column** to be passed to Visual Studio. The **Column** must be subordinate to **DataItem**. We do not need a caption defined as we will use the variable name for this field in the report layout as shown in the following screenshot:

Report 50001 Shows by Type - Report Dataset Designer			
E.. Data Type	Data Source	Name	I...
DataItem	Radio Show Type	<Radio Show Type>	
Column	UserComment	UserComment	
Column	"Radio Show Type".Code	Code_RadioShowType	✓
Column	"Radio Show Type".Description	Description_RadioShowType	✓

5. Access the **Request Options Page Designer** through **View | Request Page**.
6. Enter three lines-- Container, Group, and Field--with the **SourceExpr** of User Comment.
7. Exit the Page Designer:

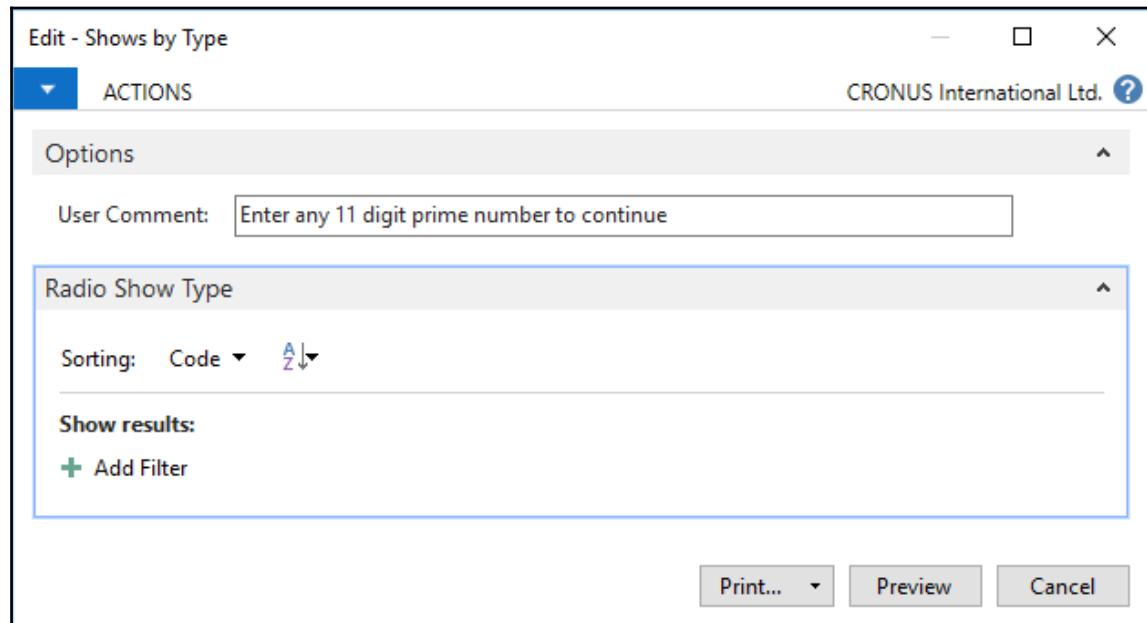
Report 50001 Shows by Type - Request Options Page Designer				
E.. Type	SubType	SourceExpr	Name	Caption
Container	ContentArea		RequestPage	<RequestPage>
Group	Group		Options	<Options>
Field		UserComment	UserComment	User Comment

8. Access Visual Studio through **View | Layout**.
9. Add a Text Box to the Layout design surface just below the Report Title, stretching the box out as far as the report layout allows.
10. Expand the **DataSet\_Result** in the **Report Data** panel.
11. Drag the **User Comment** field to the new text box as shown in the following screenshot:



12. Save the RDLC and exit Visual Studio; then save, compile, and exit the Report Designer.
13. Run **Report 50001**.

In the Request page, the user can enter their comment, as shown in the following screenshot:



The report heading then shows the comment in whatever font, color, or other display attribute the developer has defined:

17-12-2016 16:01:46	<b>Show Schedule by Type</b>	Page 1
DESKTOP-AHMVITO\MARKB	<i>Enter any 11 digit prime number to continue</i>	
<hr/>		
<hr/>		
Code	Description	
CALL-IN	Talk and Listener Interview	
No.	Name	Run Time

Because we did not specify a maximum length on our **User Comment** field, we can type in as much information as we want. Try it; type in an entire paragraph for a test.

## 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 dataset for displaying or printing, but will simply do the processing of the data we program it to do. The beauty of this capability is that we 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 us access to the Request Page to allow user input and guidance for the run. We could create the same functionality using code unit objects and programming all of the loops, filtering, user-interface Request Page, and so on, ourselves. However, with a **Processing-Only Report**, NAV gives us a lot of help, and makes it possible to create some powerful routines with minimal effort.

At the beginning of the run of a Processing-Only report, there is very little user interface variation compared to a normal printing report. The Processing-Only Request Page looks much as it would for a printing report, except that the choices to Print and Preview are not available. Everything else looks the same. Of course we have the big difference of no visible output at the end of processing.

## Creative report plagiarism and Patterns

In the same fashion as we discussed regarding pages in *Chapter 4, Pages - the Interactive Interface*, when we want to create a new report of a type that we 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. We should also check if there is a NAV Pattern defined for an applicable category of report. At the minimum, in both of these investigations, we will learn how the developers of NAV solved a data flow, totaling, or filtering challenge. In the best case, we will find a model that we can follow closely, respectfully plagiarizing (copying) a working solution, thus saving ourselves 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 reinventing the design for a report of a particular type when someone else has already invented a version of it. Not only that, but they have provided us with the plans and given us the C/AL code as well as 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, our search for a model may be based on any of the several key elements. We may be looking for a particular data flow approach and find that the NAV developers used the Integer table for some DataItems (as many reports do).

We may need a way to provide some creative filtering, similar to what is done in an area of the standard product. We might want to provide user options to print either detailed or a couple of different levels of totaling, with a layout that looks good no matter which choice the user makes. We may be dealing with all three of these design needs in the same report. In such a case, it is likely that we are using multiple NAV reports as our models, one for this feature, another for that feature, and so forth.

If we have a complicated, application-specific report to create, we may not be able to directly model our report on a model that already exists. However, often, we can still find ideas in standard reports that we can apply to our new design. We will almost always be better off using a model rather than inventing a totally new approach.

If our design concept is too big a leap from what was done previously, we should consider what we might change in our design so that we can build on the strengths of C/AL and existing NAV routines. Creating entirely new approaches may be very satisfying (when it works) but, too often, the extra costs exceed the incremental benefits.

For more NAV reporting information and ideas, please refer to Claus Lundstrom's blog at <https://clauslblog.wordpress.com/>.

## Review questions

1. The following are defined in the C/SIDE Report Designer. Choose three:
  - a) DataItems
  - b) Field display editing
  - c) Request Page
  - d) Database updating
2. Reports can be set to the ProcessingOnly status dynamically by C/AL code. True or False?
3. Reports are fixed displays of data extracted from the system, designed only for hardcopy output. True or False?

4. 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 Dataset Designer form?  
Choose one:
  - a) Nesting
  - b) Indentation
  - c) Linking
5. Queries can be designed to directly feed the Visual Studio Report Designer. True or False?
6. Union Joins are available using a special setup parameter. True or False?
7. A report that only does processing and generates no printed output can be defined. True or False?
8. The following are properties of Queries. Choose two:
  - a) TopNumberOfRows
  - b) FormatAs
  - c) OrderBy
  - d) FilterReq
9. NAV 2017 has four Report Designers. Reports can be created using any one of these by itself. True or False?
10. NAV 2017 Queries can directly generate OData and CSV files and are Cloud compatible. True or False?
11. The following are NAV 2017 Report Types. Choose three:
  - a) List
  - b) Document
  - c) Invoice
  - d) Posting

12. Queries cannot have multiple DataItems on the same indentation level. True or False?
13. Report formatting in Word has all the capabilities of report formatting in Visual Studio. True or False?
14. NAV 2017 reports can be run for testing directly from Visual Studio with an Alt + R. True or False?
15. Group properties are used to control the display of data in a parent - child relationship in Visual Studio layouts. True or False?
16. Queries are used to support what items? Choose two:
  - a) Charts
  - b) Pages
  - c) Cues
  - d) Data Sorting
17. Most reports can be initially created using the Report Wizard. True or False?
18. Interactive capabilities available after a report display include what? Choose two:
  - a) Font definition
  - b) Data Show/Hide
  - c) Sorting by columns
  - d) Data filtering
19. DataItem parent-child relationships defined in the C/Side Report Designer must also be considered in the SQL Server Report Builder in order to have data display properly in a parent-child format. True or False?
20. Users can create Word report layouts based on an existing dataset and put them into production without having access to a Developer's license. True or False?

## 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 WDTU application.

In the next chapter, we will begin exploring the key tools that pull the pieces of the C/SIDE development environment and the C/AL programming language together.

# 6

## Introduction to C/SIDE and C/AL

*"Programs must be written for people to read, and only incidentally for machines to execute."*

- Harold Abelson and Julie Sussman

*"The details are not the details. They make the design."*

- Charles Eames

So far, we have reviewed the basic objects of NAV 2017 such as tables, data fields, pages, queries, and reports. For each of these, we reviewed triggers in various areas, triggers whose purpose is to be containers for C/AL code. When triggers are fired (invoked), the C/AL code within is executed.

In this chapter, you'll start learning the C/AL programming language. Many things you already know from your experience programming in other languages. Some of the basic C/AL syntax and function definitions can be found in the NAV 2017 Help, as well as in the MSDN Library sections for Microsoft Dynamics NAV.

As with most of the programming languages, we have considerable flexibility to define our own model for our code structure. However, when we are inserting new code within existing code, it's always a good idea to utilize the model and follow the structure that exists in the original code. When we feel compelled to improve on the model of the existing code, we should do so in small increments, and we must take into account the effect of our changes on upgradability.

The goal of this chapter is to help us to productively use the C/SIDE development environment and be comfortable in C/AL. We'll focus on the tools and processes that we will use most often. We will also review concepts that we can apply in more complex tasks down the road. In this chapter, we will cover the following topics:

- C/SIDE Object Designers and their navigation
- C/AL Syntax, Operators, and Built-in Functions
- C/AL Naming conventions
- Input/Output functions
- Creating custom functions
- Basic Process Flow structures

## Understanding C/SIDE

With a few exceptions, all of the development for NAV 2017 applications takes place within the C/SIDE environment. Exceptions include use of Visual Studio (or SQL Server Report Builder) for reporting as we saw in [Chapter 5, Queries and Reports](#), plus the work we may do in a .NET language to create compatible add-ins. While it is possible, development using a text editor is only appropriate for special cases of modifications to existing objects by an advanced developer.

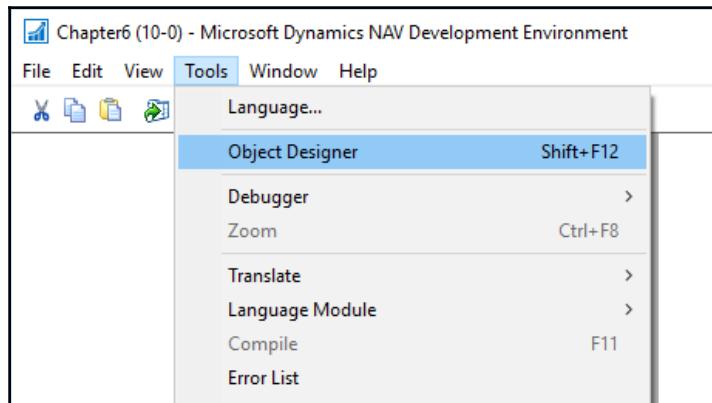
As an **Integrated Development Environment (IDE)**, C/SIDE provides us with a reasonably full set of tools for our C/AL development work. While C/SIDE is not nearly as fully featured as Microsoft's Visual Studio, it is not intended to be a general-purpose *one size fits all* development toolkit. Most importantly, C/SIDE and C/AL are designed for NAV-compatible business application software development with many features and functions specifically designed for business application work.

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.

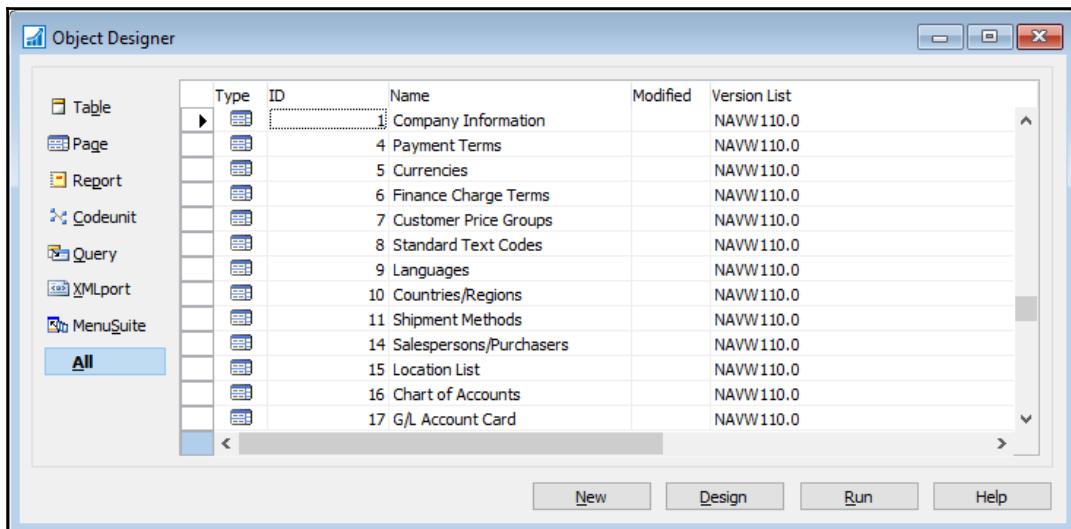
C/SIDE includes a smart editor (it knows C/AL, though sometimes not as much as we would like), the one and only C/AL compiler, integration with the application database, and tools to export and import objects both in compiled format and as formatted text files. We'll explore each of these C/SIDE areas in turn, starting with the Object Designer.

# Object Designer

All the NAV object development work starts from within the Microsoft Dynamics' Development Environment in the **C/SIDE Object Designer**. After we have invoked the Development Environment and connected to a NAV database, the **Object Designer** is accessed by selecting **Tools | Object Designer** or by pressing the *Shift + F12* keys, as shown in the following screenshot:



The type of object on which we'll work is chosen by clicking on one of the buttons on the left side of the **Object Designer** screen, as shown in the following screenshot:



The choices match the seven object types: **Table**, **Page**, **Report**, **Codeunit**, **Query**, **XMLport**, and **MenuSuite**. When we 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 the screen.

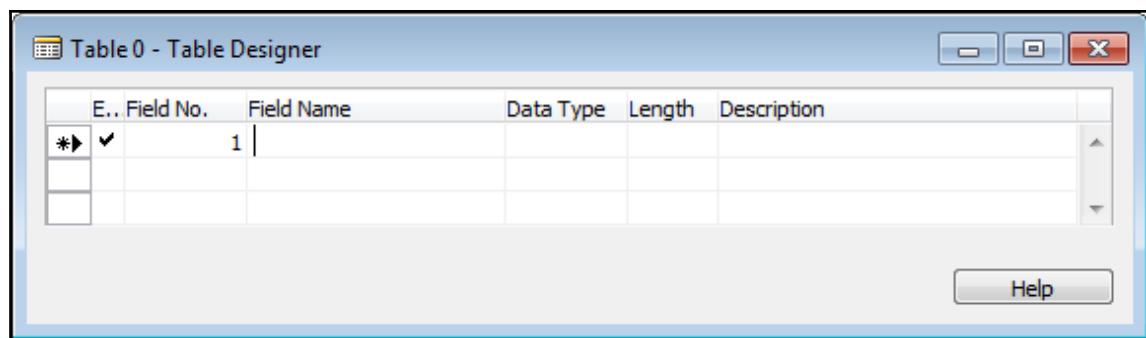
No matter which object type has been chosen, the same four buttons appear at the bottom of the screen: **New**, **Design**, **Run**, and **Help**. However, depending on which object type is chosen, the effect of selecting one of these options changes. When we select **Design**, we open the object that is currently highlighted, in a Designer specifically tailored to work on that object type. When we select **Run**, we are requesting the execution of the currently highlighted object. The results, of course, will depend on the internal design of that particular object. When we select **Help**, the C/SIDE Help screen will display, positioned at the general **Object Designer Help**.

## Starting a new object

When we select **New**, the screen we see will depend on what type of object has focus. In each case, we have the opportunity to create a new object in the Designer used for that object type.

### Accessing the Table Designer screen

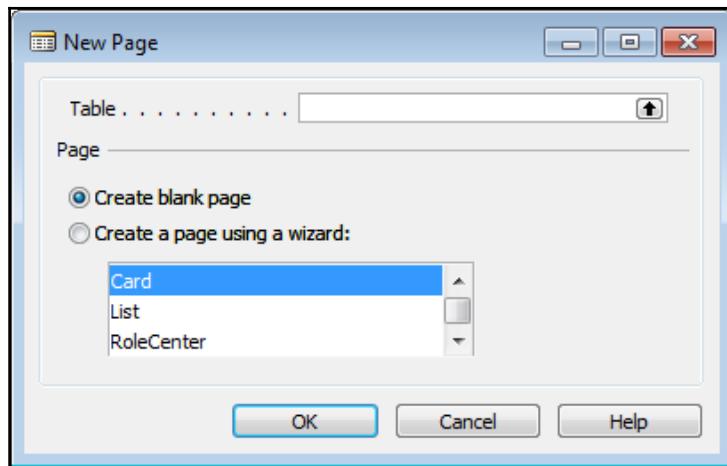
The **Table Designer** screen to start a new table is shown in the following screenshot:



The **Table Designer** invites us to begin defining data fields. All the associated C/AL code will be embedded in the underlying triggers and developer-defined functions.

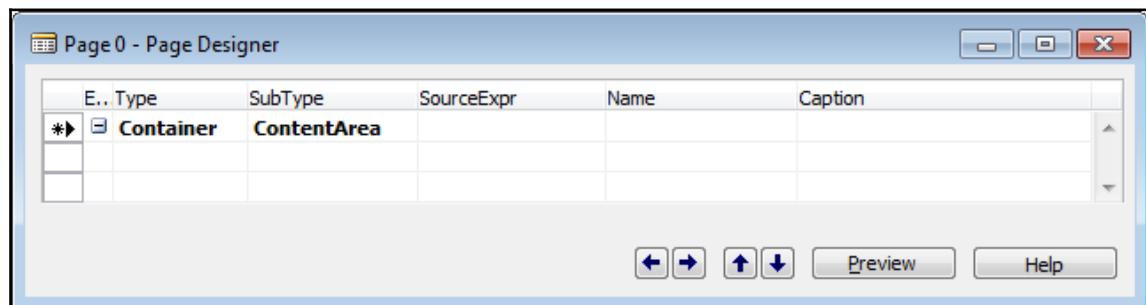
## Accessing the Page Designer

For the **Page Designer**, as shown in the following screenshot, the first screen for a new page allows us to choose between the wizard (for assistance) or the **Page Designer** (to work on our own):



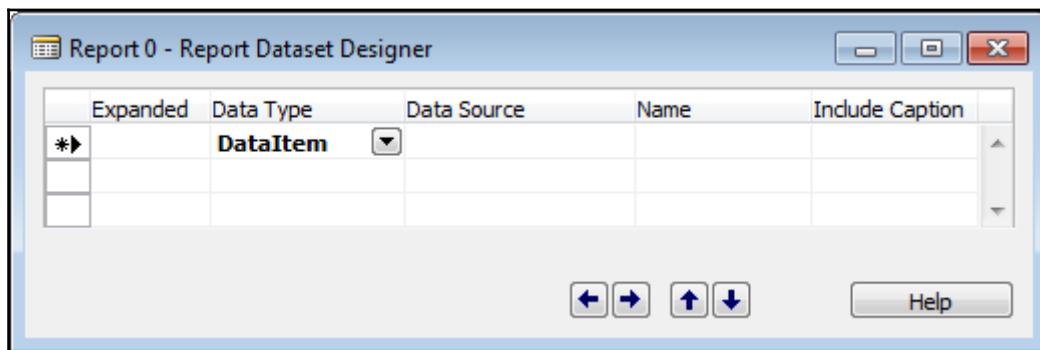
If we use the wizard, it will walk us through defining FastTabs (collapsible/expandable groups of fields) and assigning fields to those tabs, as we saw in [Chapter 4, Pages - the Interactive Interface](#). When we finish with the wizard, we will be dropped into the **Page Designer** screen with our page well on the way to completion.

If we choose not to use the wizard and want to begin designing our page totally on our own, we will select the **Create blank page** option. An empty **Page Designer** screen will be displayed. We will do all our own control and field definition. In either case, the C/AL code we create will be placed in triggers for the page or its controls. The following screenshot shows the **Page Designer** page:



## Accessing the Report Dataset Designer

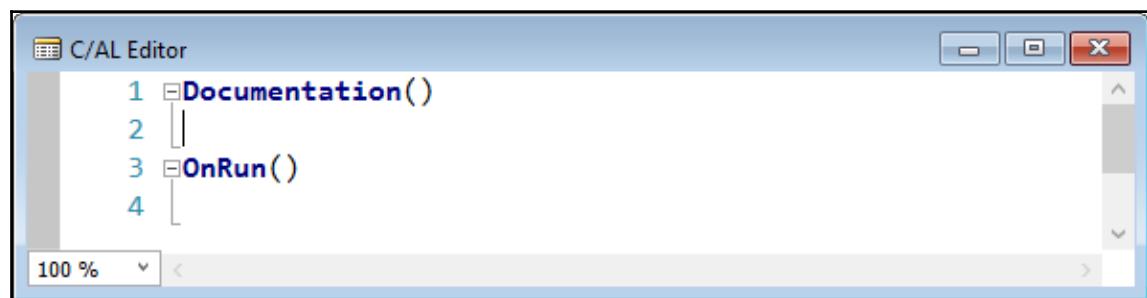
For a new report, the following **Report Dataset Designer** screen is initially displayed:



Since NAV 2017 does not have a Report wizard, we will begin report development by defining the primary DataItem for our report, and continuing from there as we did in Chapter 5, *Queries and Reports*. All C/AL code in a report is tied to the report triggers and controls.

## Accessing the Codeunit Designer

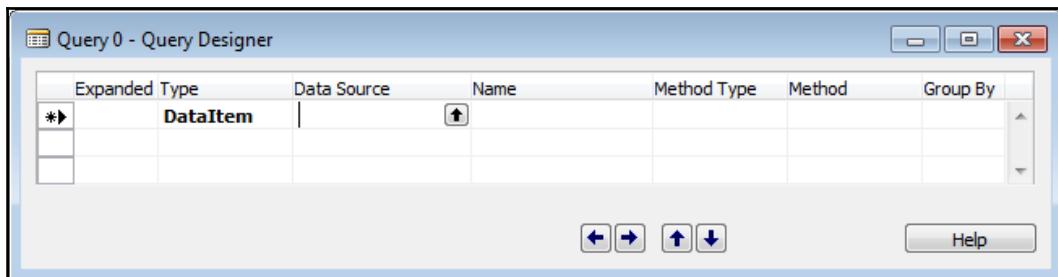
When we access the Codeunit Designer using the **New** button, a Codeunit structure is opened with the **C/AL Editor** active, as shown in the following screenshot:



Codeunits have no superstructure or surrounding framework around the single code `OnRun()` trigger. Codeunits are primarily a shell in which we can place our own functions and code so that it can be called from other objects.

## Query Designer

For a new query, the following screen will be displayed:

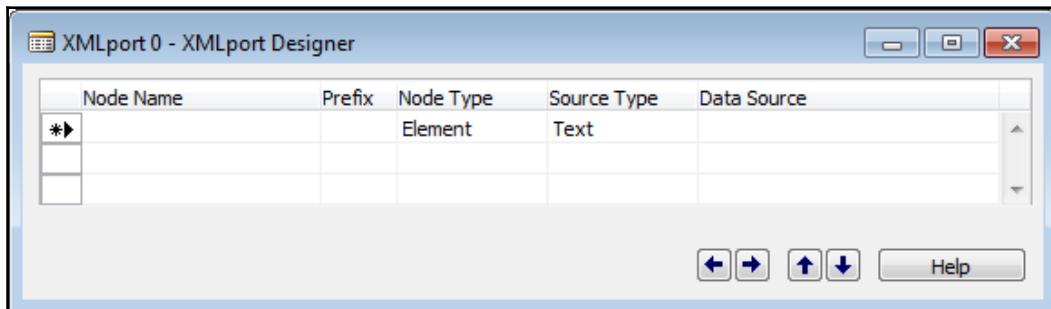


Much like a new Report, we begin Query development by defining the primary DataItem for our query and continuing from there as we did in [Chapter 5, Queries and Reports](#). All C/AL code in a query is tied to the `OnBeforeOpen` trigger of the query (this code is often used to apply filters to the Query DataItems using the **SETFILTER** function).

## XMLport Designer

XMLports are objects to define and process text-based data structures, including those which are defined in an XML format. XMLports are used to import and export both XML formatted files and text files (particularly, variations of the .csv format), but can handle many other text file formats in both delimited and fixed formats. XML is a set of somewhat standardized data formatting rules to aid dissimilar applications to exchange data. XML-structured files have become an essential component of business data systems.

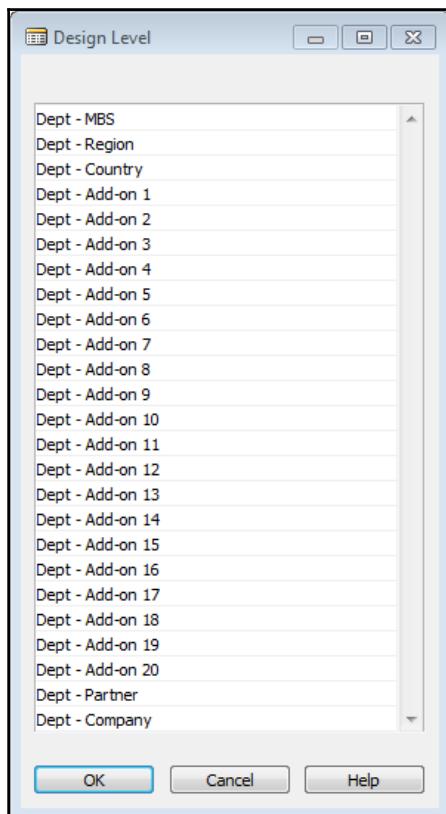
There is no wizard for XMLports. When we click on **New**, we proceed directly to the **XMLport Designer** screen as shown in the following screenshot:



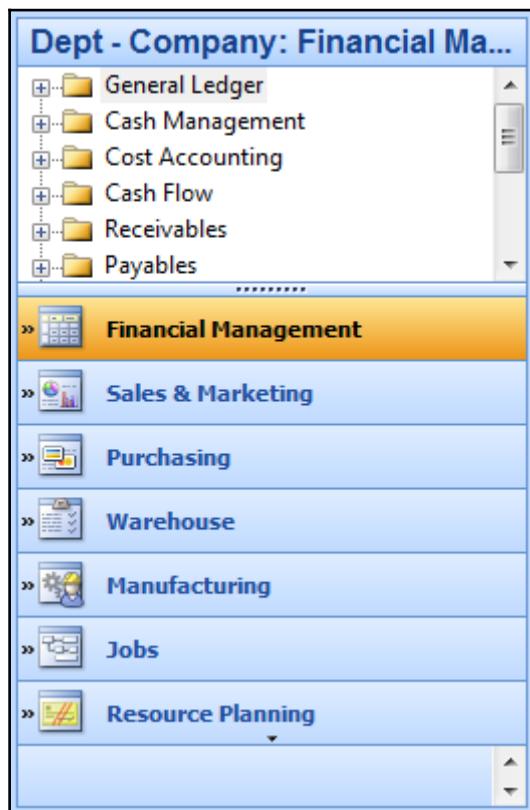
Once you become comfortable using C/SIDE and C/AL, you will learn more about XMLports for XML-formatted data and as well as other text file formats. XMLports can be run directly from menu entries as well as from within other objects. XMLport objects can also be passed as parameters to web services in a Codeunit function, thus supporting the easy passing of bulk information, such as a list of customers or inventory items.

## MenuSuite Designer

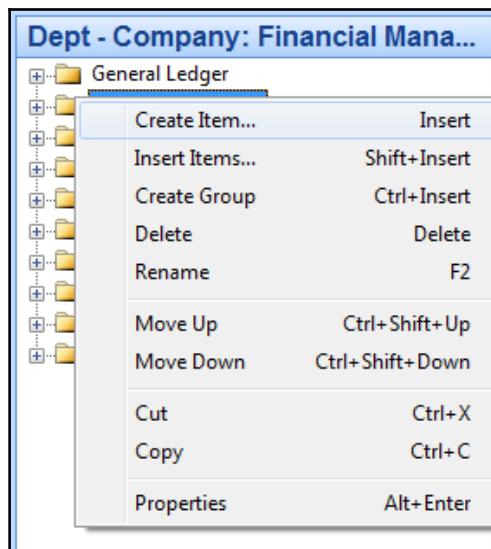
MenuSuites are used to define the menus that are available from the **Departments** button in the **Navigation** pane, and which also appear on the **Departments** page in the NAV Windows client. Items in the MenuSuite can also be found by the user via the Role Tailored Client Search function. The initial **MenuSuite Designer** screen that comes up when we ask for a new **MenuSuite** asks what MenuSuites **Design Level** we are preparing to create. The following screenshot shows all 25 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 that **New** is selected for the **MenuSuite Designer**. MenuSuites can only exist at the 25 levels shown, and only one instance of each level is supported. Once we have chosen a level to create, NAV shifts to the **MenuSuite Designer** mode. The following screenshot shows the navigation pane in Designer mode after selection of **Create | Dept - Company**:



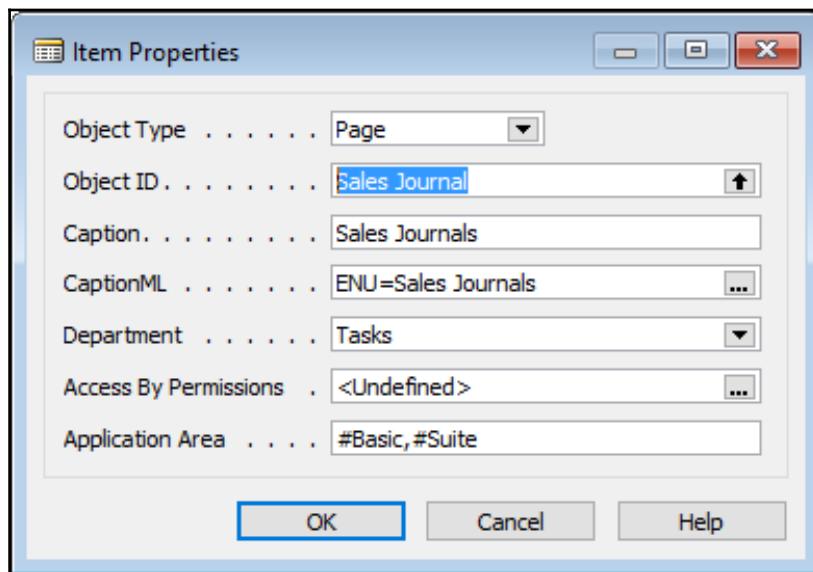
To add, change, or delete menu entries in the Navigation Pane Designer, highlight and right-click on an entry. That will display the following window. The action options visible in this **MenuSuite Designer** window are dependent on the entry that is highlighted and sometimes on the immediately previous action taken:



Descriptions of each of these menu-maintenance action options are as follows:

- **Create Item... (Insert):** This allows the creation of a new menu action entry (Item), utilizing the same window format for creation that is displayed when the entry **Properties** option is chosen
- **Insert Items...(Shift + Insert):** This allows the insertion of a new instance of a menu action entry, choosing from a list of all the existing entries
- **Create Group (Ctrl + Insert):** This allows the creation of a new group under which, menu action entries can be organized
- **Delete (Delete):** This is to delete either an individual entry or an entire group
- **Rename (F2):** This is to rename either an entry or a group
- **Move Up (Ctrl + Shift + Up) and Move Down (Ctrl + Shift + Down):** This allows moving an entry or group up or down one position in the menu structure
- **Cut (Ctrl + X), Copy (Ctrl + C), and Paste (Ctrl + V):** This provides the normal cut, copy, and paste functions for either entries or groups
- **Properties (Alt + Enter):** This displays the applicable property screen

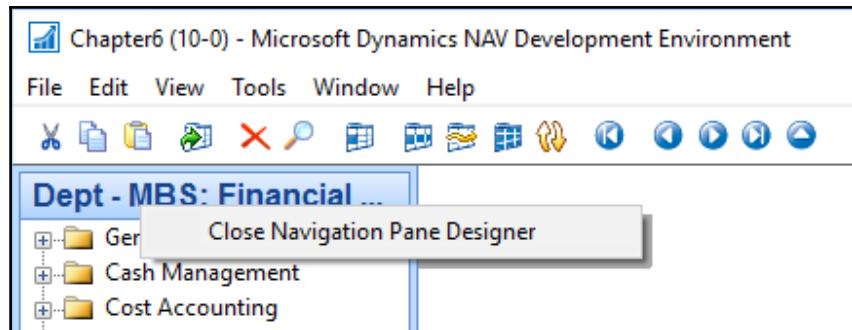
A **Group Properties** screen only contains **Caption** and **CaptionML** along with the **Department Page** checkmark field and **Application Area**. The **Item Properties** screen looks like the following screenshot:



The **Object Type** field can be any of **Report**, **Codeunit**, **XMLport**, **Page**, or **Query**. The **Department** field can be **Lists**, **Tasks**, **Reports and Analysis**, **Documents**, **History**, or **Administration**, all of which are groups in the **Departments** menu.

There are several basic differences between the MenuSuite Designer and the other object designers, including a very limited property set. One major difference is the fact that no C/AL code can be embedded within a MenuSuite entry.

To exit the Navigation Pane Designer, press the *Esc* key with focus on the Navigation Pane or right-click on the Navigation Pane Designer heading and select the **Close Navigation Pane Designer** option, as shown in the following screenshot:



We will then be asked if we want to save our changes. We should answer **Yes**, **No**, or **Cancel**, depending on what result we want.

## Object Designer Navigation

In many places in the various designers within the **Object Designer**, there are standard NAV keyboard shortcuts available, for example:

- *F3* to create a new empty entry.
- *F4* to delete the highlighted entry.
- *F5* to access the C/AL Symbol Menu, which shows us a symbol table for the object on which we are working. This isn't just any old symbol table, it is a programmer's assistant. More on this later in this chapter.
- *F9* to access underlying C/AL code.
- *F11* to do an on-the-fly compile (very useful for error checking as we go).
- *Shift + F4* to access properties.
- *Ctrl + X*, *Ctrl + C*, and *Ctrl + V* in normal Windows mode for deletion (or cut), copy, and paste, respectively.



We 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) much as if we were using a text editor. The source and target objects don't need to be of the same type.

When we are in a list of items that cannot be modified, for example, the C/AL Symbol Menu, we can focus on a column, key a letter, and jump to the next field in the column starting with that letter. This works in a number of places where search is not supported, so it acts as a very limited search substitute, applying only to an entry's first letter.

### The easiest way to copy a complete object to create a new version

Open the object in **Design** mode. 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 by pressing *Alt + F*, then the *A* key - continuously holding down the *Alt* key while pressing first *F* and then *A*.

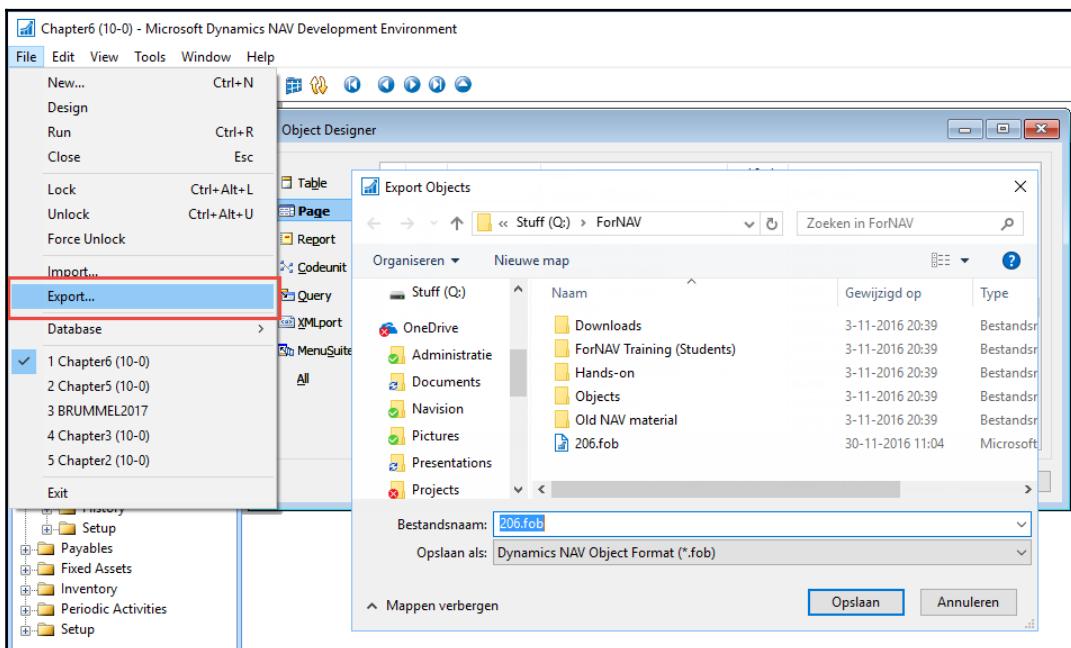


Never delete an object or a field numbered in a range where your license doesn't allow creation of an object. If there isn't a compiled (.fob) backup copy of the deleted object available for import, the deleted object will be irretrievably lost.

If we must use an object or field number in the NAV reserved number range for a different purpose other than the standard system assignment (not a good idea), we must make the change in place. Don't try a **Delete** followed by an **Add**; it won't work.

## Exporting objects

Object Export from the **Object Designer** can be accessed for backup or distribution purposes through **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) or .txt, as shown in the following screenshot:



The safer, more general purpose format to export is as a compiled object, created with a file extension of .fob. However, someone with a developer's license can export an object as a text file with a file extension of .txt. An exported text file is the only way to use a tool such as a text editor to do before and after comparisons of objects or to search all parts of objects for the occurrences of strings, such as finding all the places a variable name is used. An object text file can be used with a source-control tool such as Microsoft Visual Team Services (<https://www.visualstudio.com/vso/>) or GitHub (<https://www.github.com>).

A compiled object can be shipped to another system as a patch to be installed with little fear that it will be corrupted midstream. The system administrator at the other system simply has to import the new object following directions from the developer. Exported compiled objects also make excellent fractional backups (of code, not data). Before changing or importing any working production objects, it's always a good idea to export a copy of the before changes object images into a .fob file. These should be labeled so that they can easily be retrieved. If we want to check what objects are included in an .fob file, we can open the file in a text editor--the objects contained will be listed at the beginning. Any number of objects can be exported into a single .fob file. We can later selectively import any one or several of the individual objects from that group .fob.

## 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 decisions to be made. When we import a compiled version of an object, the **Object Designer** allows decisions about importing and provides some information to help us make those decisions.

When we import a text version of an object, the new version is brought in immediately, 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 until we do so in a separate action. By importing a text-formatted object, we could actually replace a perfectly good production object with something useless.

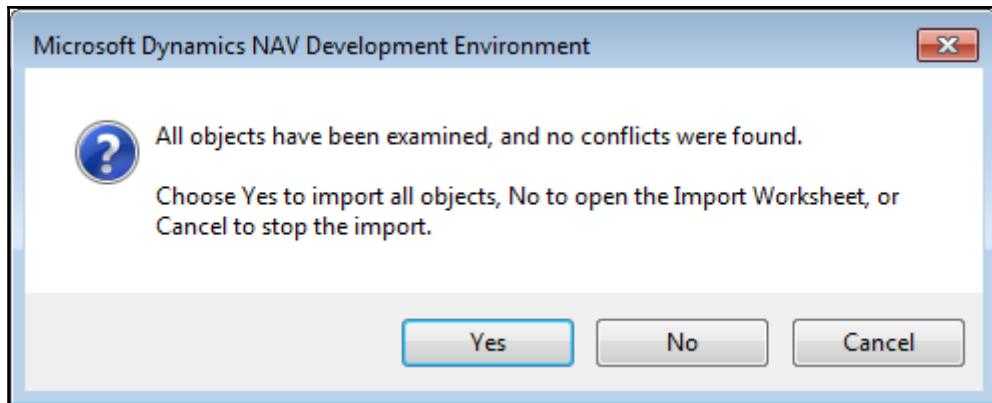


### Warning

Never import a text object until there is a current backup of all the objects that might be replaced.

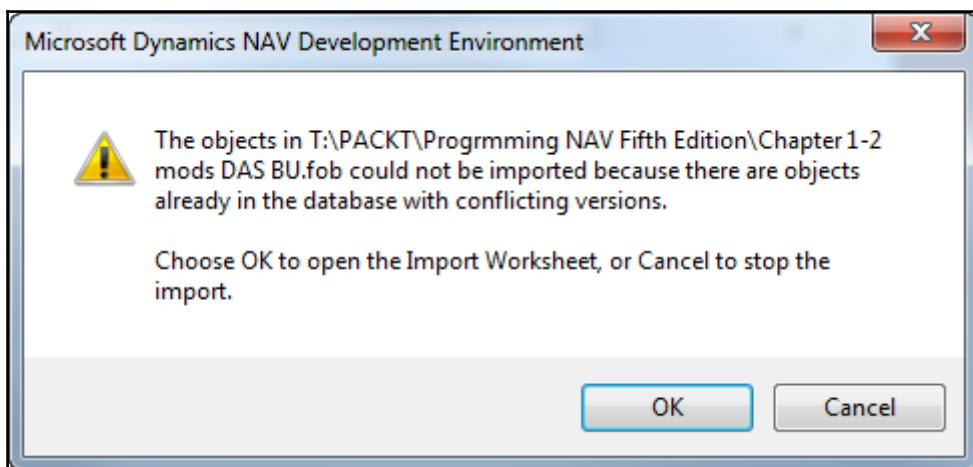
Never send text objects to an end user for installation in their system.

When we import a compiled object from a .fob file, we will get one of two decision message screens, depending on what the **Object Designer** Import finds when it checks existing objects. If there are no existing objects that the import logic identifies as matching and modified, then we will see the following dialog:

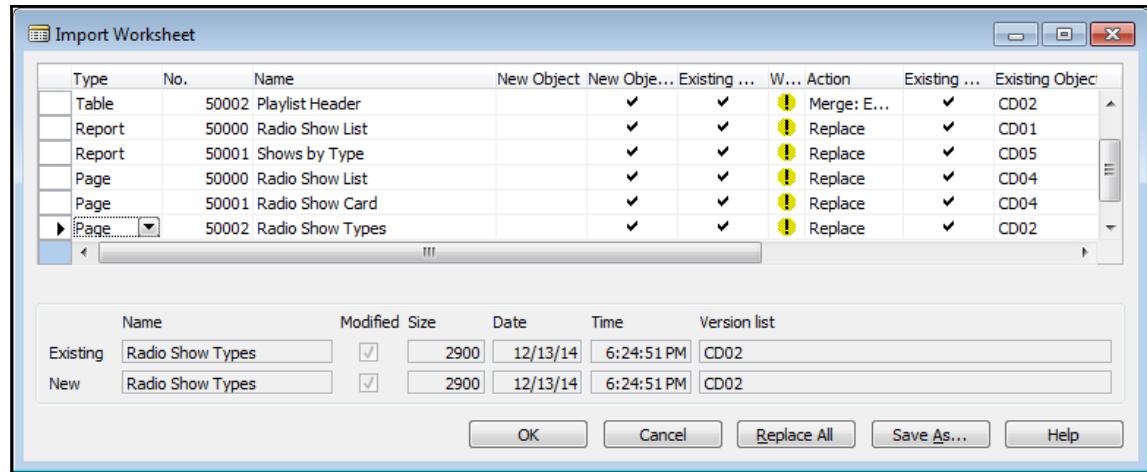


Even though you have the option to proceed without checking further, the safest thing to do is always open the Import Worksheet, in this case, by clicking on the **No** button. Examine the information displayed before proceeding with the import.

If the .fob file we are importing is found to have objects that could be in conflict with existing objects that have been previously modified, then we will see the following format of message:



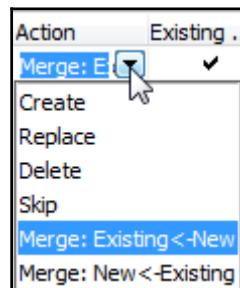
Of course, we can always click on **Cancel** and simply exit the operation. Normally, we will click on **OK** to open the **Import Worksheet** screen and examine the contents, as shown here:



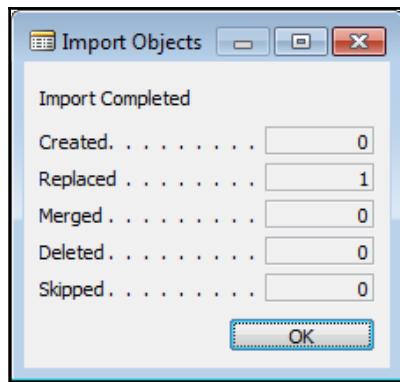
While all of the information presented is useful at one time or another, usually, we can focus on just a few fields. The basic question, on an object-by-object basis, is "Do I want to replace the old version of this object with this new one?"

At the bottom of the preceding screenshot, we can see the comparison of the **Existing** object and the **New** object information. Use this information to decide whether or not to take an action of **Create**, **Replace**, or **Skip**. More information on using the **Import Worksheet** screen and the meaning of various warnings and actions can be found in the NAV Developer and IT Pro Help under **Import Worksheet**.

Although Import allows us to merge the incoming and existing table versions, only very sophisticated developers should attempt to use this feature. The rest of us should always choose the Import Action **Replace** or **Skip**, or **Create**, if it is a new object by clicking on the drop-down button as shown in the following screenshot:

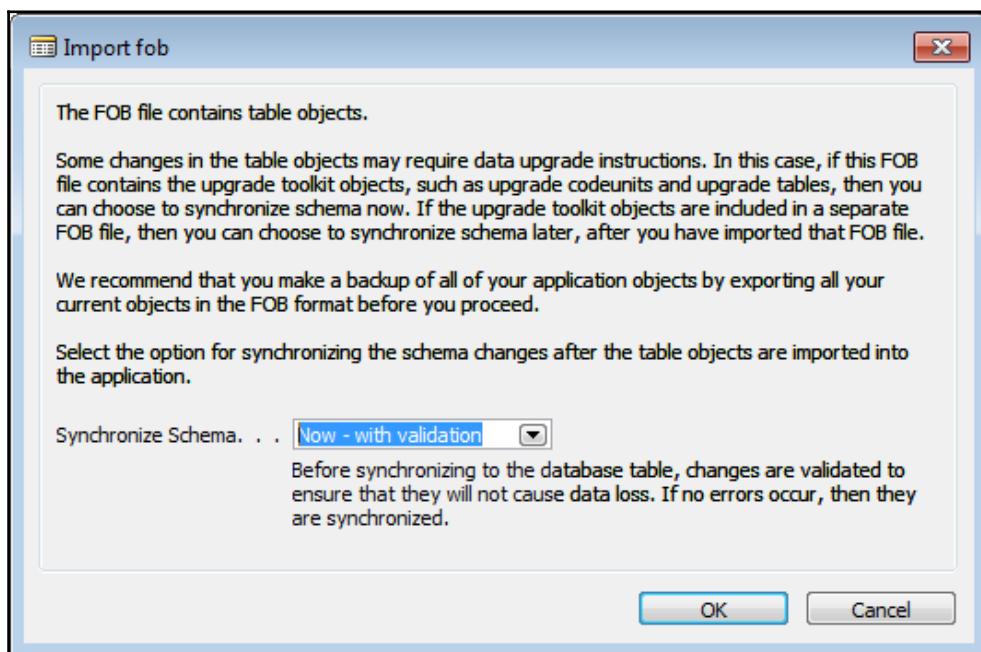


When a .fob import completes, the system tells us the result:

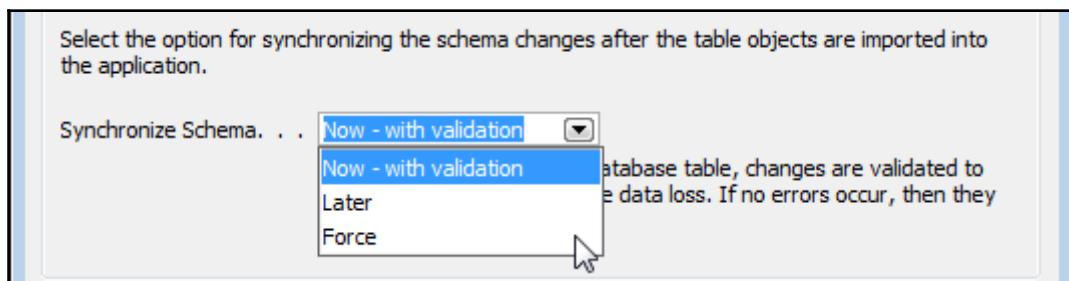


## Import Table object changes

When an existing table is changed as a result of a .fob import, the new table definition is compared against the existing schema defined in the SQL Server database. The following message will be displayed (new in NAV 2017):



The options available for the **Synchronize Schema** choice are shown in this screenshot:



If we choose the first option, we will receive another stern warning message allowing us one last opportunity to cancel the import. If we tell the system to proceed, it will do so and, at the end of its processing, inform us of the results. More information on this process is available in *Synchronizing Table Schemas* (<https://msdn.microsoft.com/en-us/dynamics-nav/synchronizing-table-schemas>).

**Warning**



Using the Force option may result in a corrupted database where the data structure is out of sync with the application software. This may not be recoverable, except by restoring a backup. Using the Force option is especially risky in a production environment.

## Text objects

A text version of an object is especially useful for only 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. As a result, it's not always easy to figure out just how an existing object is accomplishing its tasks.

An object exported to text has all its code and properties flattened out where we can use our 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 renumbering an object, can be done more easily in the text copy than within C/SIDE.

## Shipping changes as an extension

Since NAV 2015, it is possible to ship your solution as an extension. The benefit of using extensions compared to raw modifications using fob files is that the original objects delivered by Microsoft remain untouched. In theory, this should make it easier to upgrade as you don't have to merge your changes when Microsoft ships a new version of the core product.

Extensions are created using Windows PowerShell and have a list of restrictions and extra requirements.

We will create the WDTU application as an extension in Chapter 9, *Successful Conclusions*, and discuss the restrictions that come with this.



When programming for Dynamics 365 for Financials, extensions are the only way to deliver modifications for a Dynamics 365 system.

## Some useful practices

We should liberally make backups of objects on which we are working. Always make a backup of an object before changing it. Make intermediate backups regularly during development. This allows recovery back to the last working copy.

If our project involves several developers, we may want to utilize a source control system that tracks versioning and has a check-out and check-in facility for objects. Larger projects should take advantage of the testability framework that's now a part of C/AL (see *Testing the Application* in Help - <https://msdn.microsoft.com/en-us/dynamics-nav/testing-the-application>).

Compile frequently. We will find errors more easily this way. Not all errors will be discovered just by compiling. Thorough and frequent testing is always a requirement.

When we are developing pages or reports, we should do test runs (or previews) of the objects relatively frequently. Whenever we reach a stage where we have made a number of changes and again have a working copy, we should save it before making more changes.

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 sometimes tempting to do so, but that's a sure path to unhappiness. When creating a new Ledger for your application, design the process with a Journal table and a Posting process consistent with the NAV standard flow.



Follow the NAV standard approach to handle Registers, Posted Document tables, and other tables normally updated during Posting. Check out what Patterns have been defined to see what applies.

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 may continue working with the old version until they exit and re-enter, then be greeted with the new version. Production use of the obsolete object version may possibly cause confusion or even corruption of data.

Always test modifications in a reasonably current copy of the production system. Do the final testing using real data, or at least realistic data, and a copy of the customer's production license. As a rule, we 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.

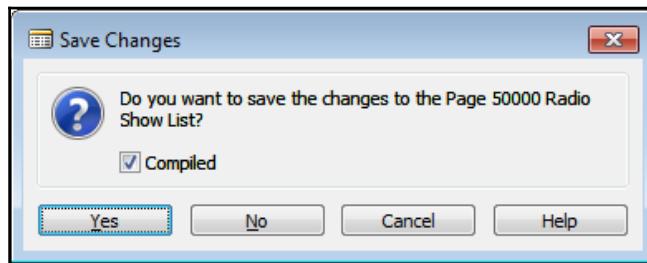
If we wish to check that changes to a production system are compatible with the rest of the system, we should import the changes into our test copy of the system and then recompile all of the objects in the system. We may uncover serious problems left by a previous developer with bad habits, so be prepared.

## **Changing data definitions**

The integration of the Development Environment with the application database is particularly handy when we are making changes to an application that is already in production use. C/SIDE is good about not letting us make changes that are inconsistent with existing data. For example, let's presume we 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 we attempt to change the definition of that field to 20 characters long, we will get a warning message when we try to save and compile the table object. We should not force the change until we adjust either the data in the database, or we adjust the change so that it is compatible with all the existing data.

## Saving and compiling

Whenever we exit the Designer for an object in which we have made a change, NAV wants to save and compile the object on which we were working. We will see a dialog similar to the following screenshot:



We have to be careful not to be working on two copies of the same object at once, as we may lose the first set of changes when the second copy is saved. If we want to save the changed material under a new object number while retaining the original object, we must **Cancel** the **Save Changes** and instead use the **File | Save As** option to rename and renumber the new copy.

If the object under development is at one of those in-between stages where it won't compile, we can deselect the **Compiled** checkbox and save it by clicking on the **Save** button without compiling it.



We should not complete a development session without getting an error-free compilation. Even if making big changes, make them in small increments.

On occasion, we may make changes that we think will affect other objects. In that case, from the **Object Designer** screen, we can select a group of objects to be compiled by Marking them. 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 on the left screen column for that object's row. After marking each of the objects to be compiled, use the **View | Marked Only** function to select just the marked objects.

We can then compile the Marked objects as a group. Select all the entries using the *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, we will get an **Error List** window indicating which objects had compilation errors of what types.

After we respond to that message, only the objects with errors will remain marked. The **Marked Only** filter will still be on so that just those objects that need attention will be shown on the screen. In fact, anytime we do a group compilation of object; those with errors will be marked so that we can use the **Marked Only** filter to select the objects needing attention.

## Some C/AL naming conventions

In 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 readability. Use meaningful names. These make the system more intuitive to the users and more self-documenting.

When we name variables, we must try to keep the names as self-documenting as possible. We should differentiate between similar, but different, variable meanings, 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 we find some violations of this in the base product). If we want to use special characters for the benefit of the user, we should put them in the caption, not in the name. If possible, we will stick to letters and numbers in our variable names. We should always avoid Hungarian naming styles (see [https://msdn.microsoft.com/en-us/library/aa260976\(v=vs.60\).aspx](https://msdn.microsoft.com/en-us/library/aa260976(v=vs.60).aspx)); keep names simple and descriptive.

There are a number of reasons to keep variable names simple. Other software products with which we may interface may have limitations on variable names. Some special characters have special meaning to other software or in another human language. In NAV, symbols such as ? and \* are wildcards and must be avoided in variable names. The \$ symbol 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 us but to the software). The same can be said for the names constructed by the internal RDLC generator, which replaces spaces and periods with underscores.

When we are defining multiple instances of a table, we should either differentiate clearly by name (for example, Item and NewItem) or by a descriptive suffix (for example, Item, ItemForVariant, ItemForLocation). 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 sometimes not warn us that we have used a reserved word, and we may find our logic and the automatic logic working at very mysterious cross-purposes.

Do not start variables with a prefix `x`, which is used in some automatically created variables, such as `xRec`. We should make sure that we can clearly differentiate between working storage variable names and the field names originating in tables. Sometimes, C/SIDE will allow us to have a global name, local name, and/or record variable name, all with the same literal name. If we do this, we are practically guaranteeing a variable misidentification bug where the compiler uses a different variable than what we 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, we should be reasonably descriptive. Don't name two functions located in different objects with the 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 are obvious, but for clarity's sake, we'll summarize here. In Chapter 3, *Data Types and Fields*, we reviewed various data types for variables defined within objects (referred to 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, *Data Types and Fields*, are those that can be defined in either the C/AL Global Variables or C/AL Local Variables tabs. Variables can also be defined in several other places in an NAV object.

## C/AL Globals

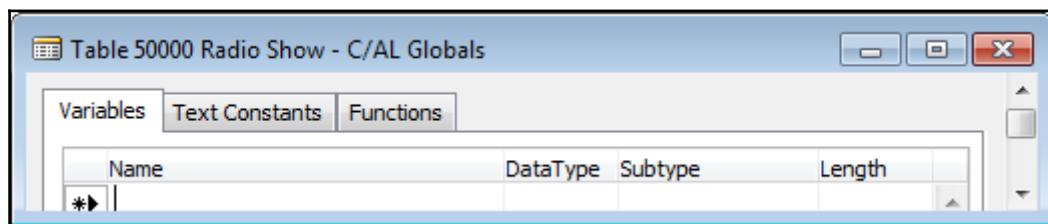
Global variables are defined on the **C/AL Globals** form in the **Variables** tab.



Global variables should be avoided. Dynamics NAV legacy code has inherited many of them from the MS-DOS era of the product when local variables were not available.

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 *NAV Developer and IT Pro Help* on *How to: Add a Text Constant to a Codeunit* (<https://msdn.microsoft.com/en-us/dynamics-nav/how-to--add-a-text-constant-to-a-codeunit>). The information applies generally.

Global Functions are defined on the **Functions** tab of the **C/AL Globals** form. The following screenshot shows the **C/AL Globals** form:



## C/AL Locals

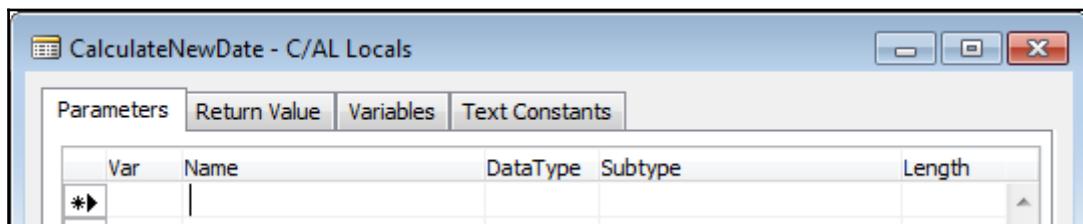
Local identifiers only exist defined within the range of a trigger. This is true whether the trigger is a developer-defined function, one of the default system triggers, or standard application-supplied functions. In NAV 2017, when a new function is defined, it is set as a local function by default. This means that if we want the new function to be accessible from other objects, we must set the Local property of the function to No.

### Function local identifiers

Function local identifiers are defined on one or another of the tabs on the **C/AL Locals** form that we use to define 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 of the same names. The tabs of the **C/AL Locals** form can be seen in the following screenshot:



## Other local identifiers

Trigger local variables (variables that are local to the scope of a trigger) 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 only the **Variables** and **Text Constants** tabs exist for trigger Local Variables. The use of the **Variables** and **Text Constants** tabs are exactly the same for triggers as for functions. Whether we are working within a trigger or a defined function, we can access the local variables through the menu option, **View | C/AL Locals**.

## Special working storage variables

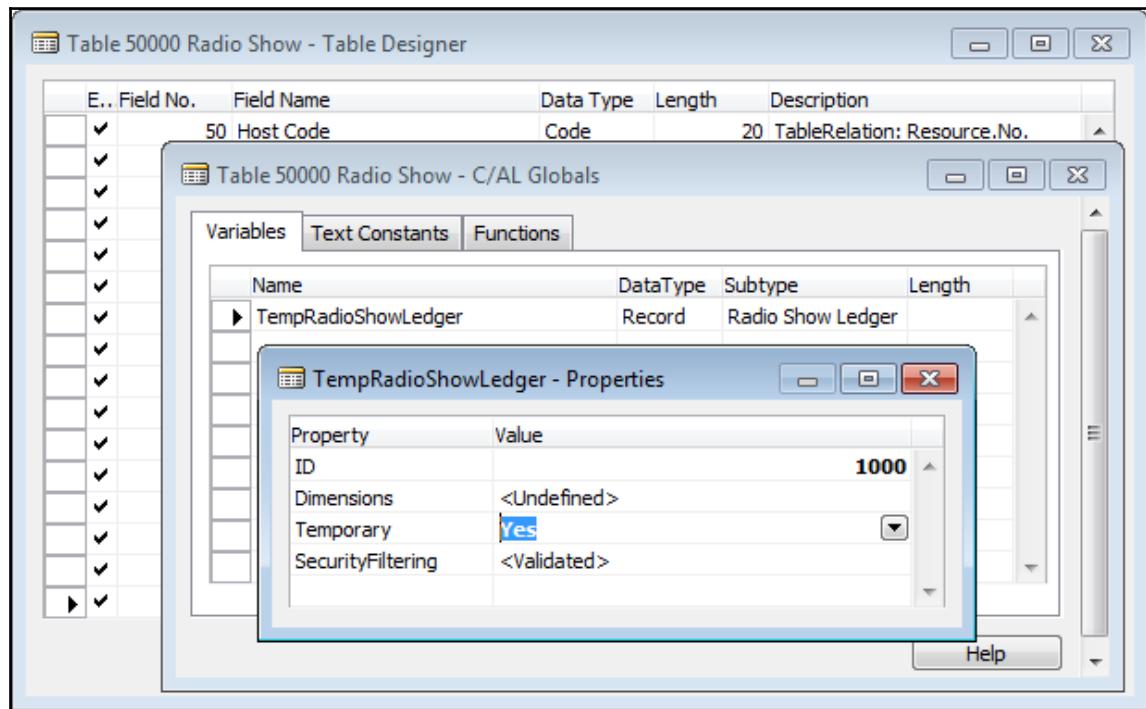
Some working storage variables have additional attributes to be considered.

### Temporary tables

Temporary tables were discussed in [Chapter 2, Tables](#). 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, perform the following steps:

1. Select **View | C/AL Globals**.
2. Enter a variable name and data type of Record.
3. Choose the table whose definition is to be replicated for this temporary table as the Subtype.
4. With focus on the new Record variable, click on the **Properties** icon or press the **Shift + F4** keys.
5. Set the **Temporary** property to **Yes**.

That's it. We've defined a temporary table like the one in the following screenshot:



We can use a temporary table just as though it were a permanent table with some specific differences:

- The table contains only the data we add to it during this instance of the object in which it resides.
- We 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 client system in a user-specific instance of the business logic. It is, therefore, inherently single user.
- A properly utilized temporary table reduces network traffic and eliminates any locking issues for that table. It is often much faster than processing the same data in a permanent database-resident table because both data transmission and physical storage I/O are significantly reduced.

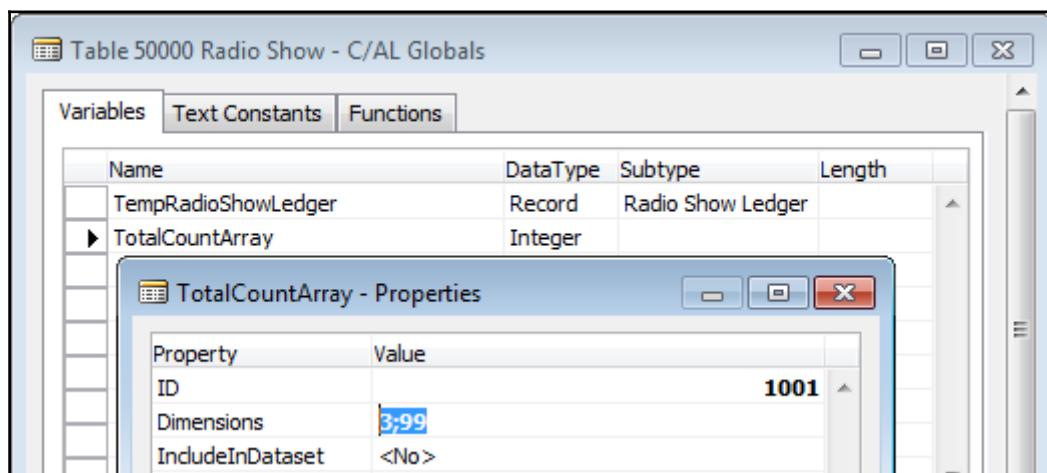


In some cases, it's a good idea to copy the database table data into a temporary table for repetitive processing within an object. This can give us a significant speed advantage for a particular task by updating data in the temporary table, then copying it back out to the database table at the end of processing.

When using temporary tables, we need to be very careful that references from the C/AL code in the temporary table, such as data validations, don't inappropriately modify permanent data elsewhere in the database. We also must remember that if we forget to properly mark the table as temporary, we will likely corrupt production data with our processing.

## 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 which defines an integer array of 3 rows, with 99 elements in each row:



The semicolon separates the dimensions of the array. The numbers indicate the maximum number of elements of each of the dimensions. An array variable such as TotalCountArray is referred to in C/AL as follows:

- The 15th entry in the first row is TotalCountArray[1, 15]
- The last entry in the last row is TotalCountArray[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, we must make sure the code is especially thoroughly tested so that we aren't surprised by unexpected results. NAV 2013 added the capability to also use arrays from the .NET Framework. For more information, see the Help titled *Using Arrays* (<https://msdn.microsoft.com/en-us/dynamics-nav/using-arrays>).

## Initialization

When an object is initiated, the variables in that object are 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 individual components of complex variables are appropriately initialized. The system also automatically initializes all system-defined variables.

Of course, once the object is active, through our code and property settings, we can do whatever additional initialization we wish. If we wish to initialize variables at intermediate points during processing, we can use any of the several approaches. First, we will reset a Record variable (for example, the `TempRadioShowLedger` temporary table defined in the preceding example) with the `RESET` function, then initialize it with the `INIT` function in statements in the form, as follows:

```
TempRadioShowLedger.RESET;  
TempRadioShowLedger.INIT;
```

The `RESET` function makes sure all previously set filters on this table are cleared. The `INIT` function makes sure all the fields, except those in the **Primary Key**, are set either to their `InitValue`, property value, or to their data type default value. The **Primary Key** fields must be explicitly set by the C/AL code.

For all types of data, including complex data types, we 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 an Integer data type, the element would be set to Integer zero 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 us with some variables automatically, such as **Rec**, **xRec**, **CurrPage**, **CurrReport**, and **CurrXMLport**. What variables are provided is dependent on the object in which we are operating. Descriptions of some of these can be found in the Help titled *System-Defined Variables* (<https://msdn.microsoft.com/en-us/dynamics-nav/system-defined-variables>).

# C/SIDE programming

Many of the things that we do during development in C/SIDE might not be called programming by some people as it doesn't involve writing C/AL code statements. However, 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 DataItem levels, creating Request pages in Reports, defining Controls and their properties, defining Report data structures and their properties, creating Source Expressions, defining Functions, and, of course, writing C/AL statements in all the places where we can put C/AL. We will focus on 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 C/SIDE Report Designer and not within the RDLC created by Visual Studio.



Because no coding can be done within MenuSuites, we will omit those objects from the programming part of our discussions.

NAV objects are generally consistent in structure. Most have some properties and triggers. Pages and Reports have controls, though the tools that define the controls in each are specific to the individual object type. Reports have a built-in DataItem looping logic. XMLports also have DataItem looping logic, but those are structured differently from reports; for example, Reports can have multiple DataItems at 0 level and XMLports can only have one Node at 0 level. All the object types that we are considering can contain C/AL code in one or more places. All of these can contain function definitions that can be called either internally or externally (if not marked as Local). Remember, good design practice says any functions designed as "library" or reusable functions that are called from a variety of objects should be placed in a Codeunit or, in some circumstances, in the primary table.



Don't forget that our fundamental coding work should focus on tables and function libraries as much as possible, as these are the foundation of the NAV system.

## Non-modifiable 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 prewritten functions that are available to us to perform a wide variety of different tasks. The underlying logic for some of these functions is hidden and not modifiable. These non-modifiable functions are supplied as part of the C/AL programming language. Some simple examples of non-modifiable functions follow:

- DATE2DMY: Supply a date to this function and, depending on a calling parameter, it will return the integer value of the day, month, or year of that date
- STRPOS: Supply a string variable and a string constant to the function and it 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 to the function and it will read the record in the table with a **Primary Key** value equal to the supplied value, if a matching record exists
- INSERT: This function will add a record to a table
- MESSAGE: We supply a string and optional variables and 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 basic language components.

## Modifiable functions

In addition to the prewritten language component functions, there are a large number of prewritten application component functions. The difference between the two types is that the code implementing the latter is visible and modifiable, though we 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's Shipping Address to eliminate empty lines and standardize the layout based on user-defined setup parameters. Such a 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

theFormat Address Codeunit. We can explore the following Codeunits for some functions we 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 we may find useful in a future development project, either to be used directly or as templates to design our own similar function. Many library Codeunits have the words Management or Mgt in their name:

Object number	Name
1	ApplicationManagement
356	DateComprMgt
358	DateFilter-Calc
359	PeriodFormManagement
365	Format Address
397	Mail
5052	AttachmentManagement
5054	WordManagement
6224	XML DOM Management

The prewritten application functions, generally, have been provided to address the needs of the NAV developers working at Microsoft. However, we can use them too. Our 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. A feature has been added to the C/AL Editor to help us in our review of such functions. If we highlight the function reference in the inline code and right click, we are given the option **Go to definition**. This allows us to easily find the function's code and review it.

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 we decide to change one of these functions, we should make sure our change is compatible with all existing uses of that function.



A useful trick to find all the calls of a function is to add a dummy calling parameter to the function (temporarily), and then compile all objects in a copy of the application system. Errors will be displayed for all objects that call the changed function (we mustn't forget to remove the dummy calling parameter and recompile when we're done testing). This technique not only works for functions created by Microsoft, but also for functions created as part of a customization or an add-on.

Rather than changing an existing function, it is much better to clone it into our own library Codeunit, creating a new version, and making any modifications to the new version while leaving the original untouched.



If you want to ship your solution as an Extension, you cannot modify functions in the base product. You should subscribe to events instead. We will discuss events later in the recipe, *Creating an extension*, in Chapter 9, *Successful Conclusions*.

## Custom functions

We can also create our own custom functions to meet any need. The most common reason to create a new function is to provide a single, standardized instance of logic to perform a specific task. When we need to use the same logic in more than one place, we should consider creating a callable function.

We should also create a new function when we're modifying standard NAV processes. Whenever more than three or four lines of code are needed for the modification, we should consider creating the modification as a function. If we do that, the modification to the standard process can be limited to a call to the new function. It's usually not a good idea to embed a new function into an existing standard function. It's better to clone the existing function and make the modifications in-line in our copy.

Although using a function to insert new code into the flow is a great concept, occasionally, it may be difficult to implement in practice. For example, if we 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. In such a case, we may just settle for creating an in-line modification and doing a good job of commenting the modification. This is most reasonable, when the modification code is only required in one place and does not also need to be referenced elsewhere.

If a new function will be used in several objects, it should be housed in our library codeunit (we may choose to have multiple library codeunits for the purpose of clarity or project management). If a new function is only for use in a single object, then it can be resident in that object. This latter option also has the advantage of allowing the new function direct access to the global variables within the object being modified, if necessary.



When working with Extensions, it is not allowed to add new functions to existing objects created by Microsoft. You should always create them in your own library Codeunit objects.

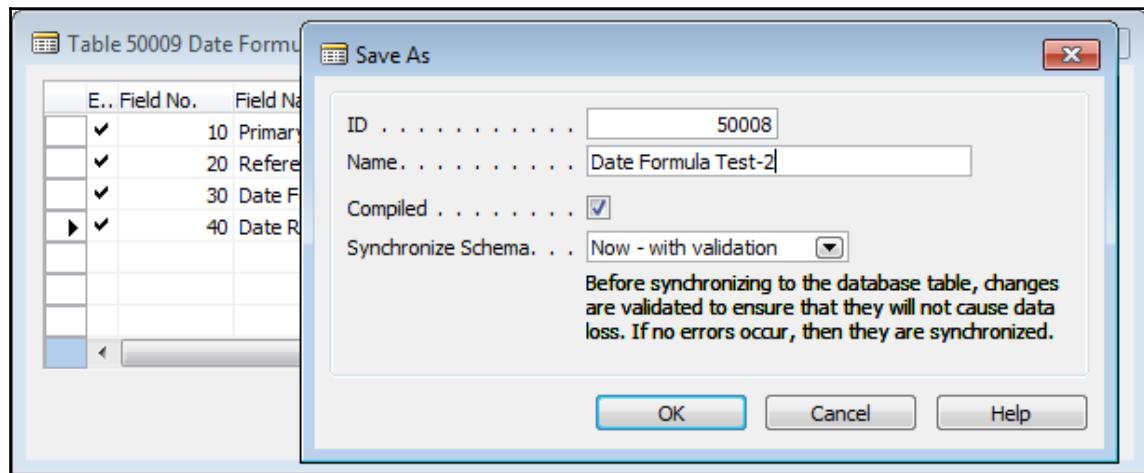
## Creating a function

Let's take a quick look at how a function can be created. We'll 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 WDTU application, we will simply call it `Radio Show Management`. In that Codeunit, we'll 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](#), to illustrate how a `DateFormula` data type works. This time, our focus will be on the creation of a function.



When working on projects with larger teams, creating more codeunits is advised to have a better chance of each developer working on separate objects.

Our first step is to copy Table 50009, which we created for testing, and then save it as Table 50008. As a reminder, we do that by opening Table 50009 in the **Table Designer**, then selecting **File | Save As**, changing the object number to 50008 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. We used CDM and 03 for the original table Version in Chapter 3, *Data Types and Fields*. Now we'll add ,06 to make the new table Version read CDM 03,06.

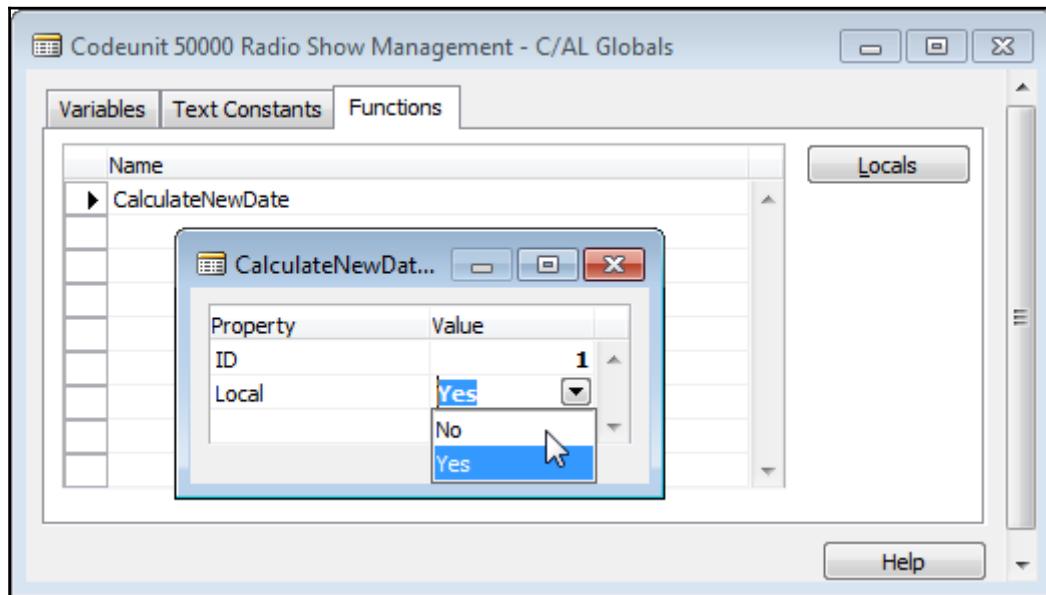
We will create our new Codeunit by simply clicking on the **Codeunit** button on the left of the **Object Designer** screen, then clicking on the **New** button, choosing **File | Save As**, and entering the **Object ID** of 50000 and **Name** as Radio Show Management.

Now comes the important part - designing and coding our new function. When we had the function operating as a local function inside the table where it was called, we didn't worry 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 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)
```

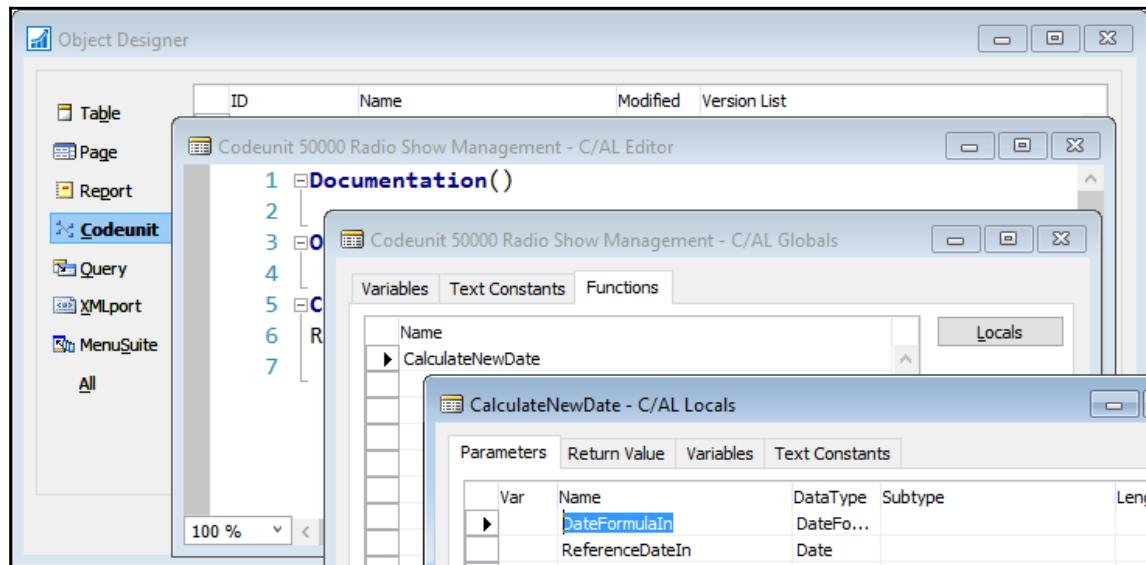
In other words, we need to feed two values into our new callable function and accept a Return value back on completion of the function's processing.

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, such as `CalculateNewDate`, as shown in the following screenshot, then keeping the function name in focus (highlighted), display the Properties of the function by either clicking on the **Properties** icon, pressing **Shift + F4**, or navigating through **View | Properties** :



Set the **Local** property to **No** so that we will be able to call the function from other objects. 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 practices, we will define two input parameters, as shown in the following screenshot:



#### Re Var column in the leftmost column of the Parameters tab form:

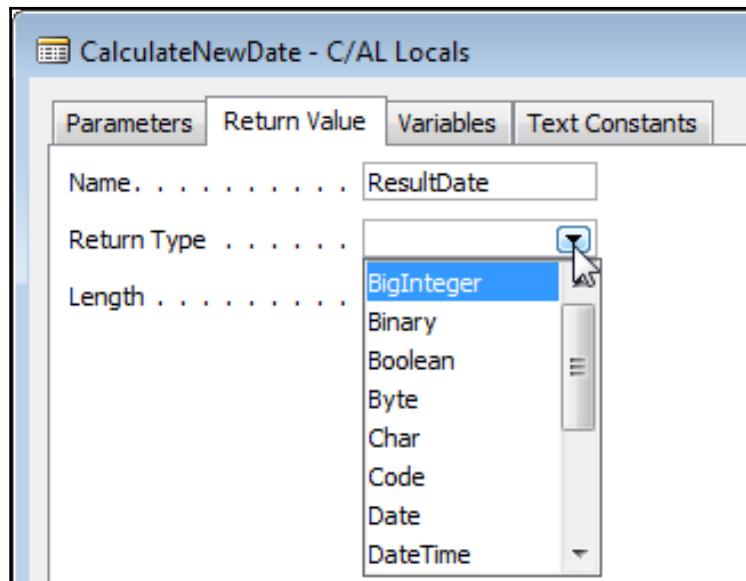
If we checkmark the **Var** column, the parameter is passed by reference to the original calling routine's copy of that variable. If the parameter is passed by reference, when the called function changes the value of an input parameter, it directly changes the original variable value in the calling object.



Because we've specified the input parameter passing here with the **Var** column unchecked, changes in the value of that input parameter will be passed by value. That makes the parameter local to this function and any changes to its value will not directly affect variable in the calling routine. Checking the **Var** column on one or more parameters is a way to effectively have multiple results passed back to the calling routine.

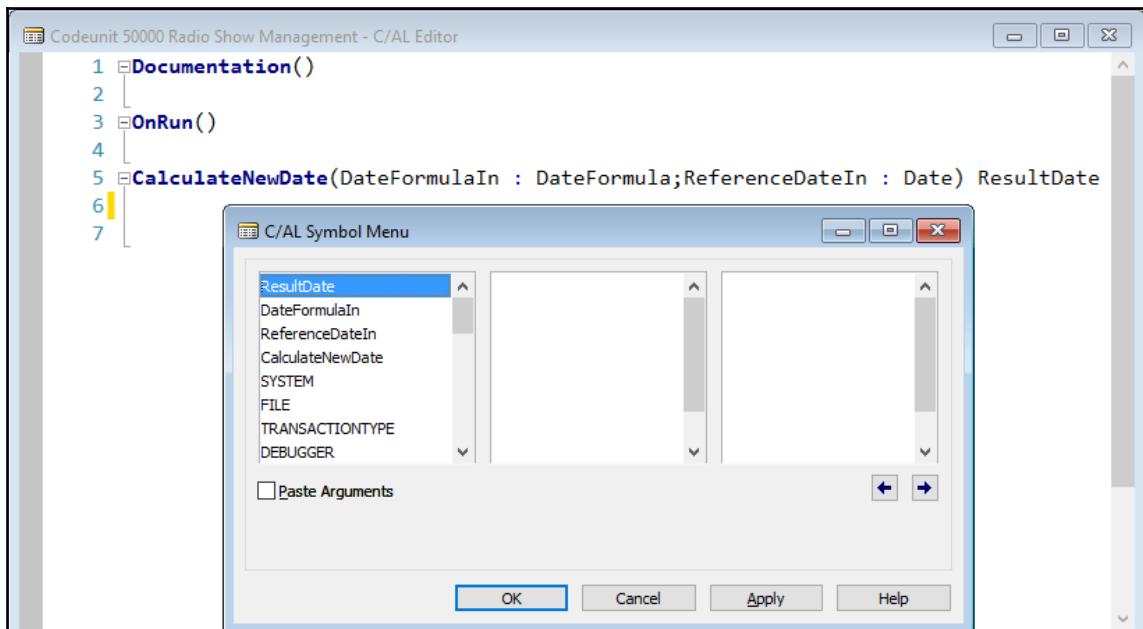
Parameter passing with the **Var** column checked (passing by reference) is also faster than passing by value, especially when passing complex data types (for example, records).

Select the **Return Value** tab and define our output variable, as shown in the following screenshot:



A **Name** value is not required for the **Return Value** option if the return terminates processing with an `EXIT([ReturnValue])` instruction. Choose the **Date** option for the **Return Type**. Exit by using the Esc key and the result 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 our new Codeunit 50000 in view and our cursor placed in the code area for our new function, we will click on **View | C/AL SymbolMenu** (or just press *F5*) and see the following screenshot:



We see in the **C/AL Editor** that our `CalculateNewDate` function has been defined with two parameters and a result. Now, press *Esc* or select **OK**, move the cursor to the `OnRun` trigger code area and again press *F5* to view the **C/AL Symbol Menu**. We don't see the two parameters and result variables.

Why? Because the **Parameters** and **Return Value** 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, because it is a very valuable C/AL development tool. However, right now, we need to finish our new function and integrate it with our test Table 50008.

Move the cursor back to the code area for our new function. Click on the menu item **Window | Object Designer | Table button**, then on **Table 50008 | Design**, and press **F9**. That will take us to the C/AL Code screen for Table 50008. 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 50008. We should see the following screen displayed:

```
Codeunit 50000 Radio Show Management - C/AL Editor
1 Documentation()
2
3 OnRun()
4
5 CalculateNewDate(DateFormulaIn : DateFormula; ReferenceDateIn : Date) ResultDate : Date
6 "Result Date" := CALCDATE("Date Formula to Te","Reference Date for Calculation");
7
```

Edit the line of code just pasted into the codeunit so the variable names match those shown in our function trigger in the preceding screenshot. This will result in the following screenshot:

```
Codeunit 50000 Radio Show Management - C/AL Editor
1 Documentation()
2
3 OnRun()
4
5 CalculateNewDate(DateFormulaIn : DateFormula; ReferenceDateIn : Date) ResultDate : Date
6 ResultDate := CALCDATE(DateFormulaIn,ReferenceDateIn);
7
```

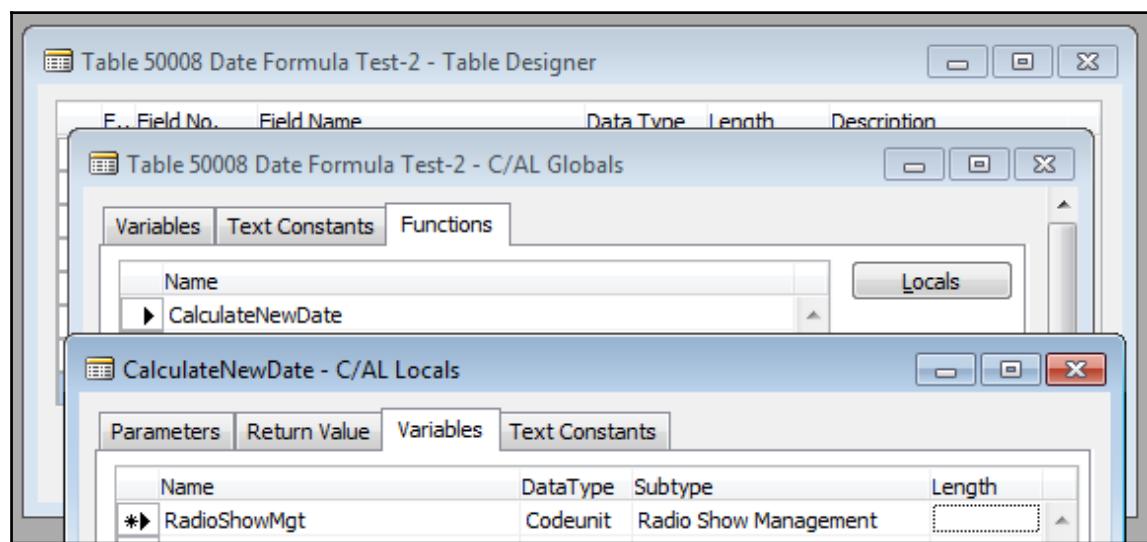
Press **F11** to check if we have a clean compile. If we get an error, we must do the traditional programmer thing. Find it, fix it, recompile. Repeat until we get a clean compile. Then exit and save our modified Codeunit 50000.

Finally, we will return to our test Table 50008 to complete the changes necessary to use the external function rather than the internal function. We have two obvious choices for doing this. One is to replace the internal formula in our existing internal function with a call to our external function. This approach results in fewer object changes.

The other choice is to replace each of our internal function calls with a call to the external function. This approach may be more efficient at runtime because, when we need the external function, we will invoke it in one step rather than two. We will walk through the first option here and then you should try the second option on your own.

Which option is the best? It depends on our criteria. Such a decision comes down to a matter of identifying the best criteria on which to judge the design options, then applying those criteria. Remember, whenever feasible, simple is good.

For the first approach (calling our new Codeunit resident function), we must add our new Codeunit 50000 Radio Show Management to Table 50008 as a variable. After designing the table, navigate to **View | Globals**, click on the **Functions** tab, highlight the CalculateNewDate function, click on the **Locals** button, and click on the **Variables** tab. Add the Local variable as shown in the following screenshot (it's good practice to define variables as local unless global access is required):



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 as follows:

```
Global/LocalVariable :=  
Global/LocalObjectName.FunctionName(Parameter1, Parameter2, ...).
```

Based on that, the new line of code should be as follows:

```
"Date Result" := RadioShowMgt.CalculateNewDate("Date Formula to  
Test","Reference Date for Calculation");
```

If all has gone well, we should be able to save and compile this modified table. When that step works successfully, we can run the table and experiment with different Reference Dates and Date Formulas, just as we did back in Chapter 3, *Data Types and Fields*. We should get the same results for the same entries as we saw before.

When you try out the other approach of replacing each of the calls to the internal function by directly calling the external function, you will want to do the following:

- Either define the Radio Show Management Codeunit as a Global variable or as a Local variable for each of the triggers where you are calling the external function
- Go to the **View | Globals | Functions** tab and delete the now unused internal CalculateNewDate function

We should now have a better understanding of the basics of constructing both internal and external functions and some of the optional design features available to us for building functions.

## 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 character and syntax practices as Pascal.

## 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, to the right of the assignment symbol, is assigned to the variable on the left side, as shown in the following line of code:

```
"Phone No." := '312-555-1212';
```

All statements are terminated with a semicolon. Multiple statements can be placed on a single program line, but that makes the code hard for others 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 `Phone No.`). The same structure 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 phone number string in the preceding code line).

Double quotes are used to surround an identifier (for example, a variable or a function name) that contains any character other than numerals or uppercase and lowercase 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 language-- to indicate sets of expressions to be interpreted according to their parenthetical groupings. The expressions are interpreted in sequence, first the innermost parenthetical group, then the next level, and so forth. The expression  $(A / (B + (C * (D + E))))$  would be evaluated as follows:

1. Summing  $D + E$  into Result1.
2. Multiplying Result1 by  $C$ , yielding Result2.
3. Adding Result2 to  $B$  yielding Result3.
4. Dividing  $A$  by Result3.

Brackets [ ] are used to indicate the presence of subscripts for indexing of array variables. A text string can be treated as an array of characters, and we can use subscripts with the string name to access individual character positions within the string, but not beyond the terminating character of the string. For example, `Address[1]` represents the leftmost character in the `Address` text variable contents.

Brackets are also used for IN (In range) expressions, such as the following code snippet:

```
Boolean := SearchValue IN[SearchTarget]
```

In this line, `SearchValue` and `SearchTarget` are text variables.

Statements can be continued on multiple lines without any special punctuation; although, we can't split a variable or literal across two lines. Because 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';
```

## Expressions

Expressions in C/AL are made up of four elements: constants, variables, operators, and functions. We could include a fifth element, expressions, because an expression may include a subordinate expression within it. As we become more experienced in coding C/AL, we 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.

We can create complex statements that will conditionally perform important control actions and operate in much the way that a person would think about the task. We can also create complex statements that are very difficult for a person to understand. These are tough to debug and sometimes almost impossible to deal with in a modification.

One of our responsibilities is to learn to tell the difference so we can write code that makes sense in operation, but is also easy to read and understand.

According to the NAV *Developer and IT Pro Help*, a *C/AL Expression* (<https://msdn.microsoft.com/en-us/dynamics-nav/statements-and-expressions>) is a group of characters (data values, variables, arrays, operators, and functions) that can be evaluated with the result having an associated data type. The following are two code statements that accomplish the same result in slightly different ways. They each assign a literal string to a text data field. In the first one, the right side is a literal data value. In the second, the right side of the := assignment symbol is an expression:

```
ClientRec."Phone No." := '312-555-1212';
ClientRec."Phone No." := '312' + ' -' + '555' + ' -' + '1212';
```

## Operators

Now, we'll review C/AL operators grouped by category. Depending on the data types we are using with a particular operator, we may need to know the type conversion rules defining the allowed combinations of operator and data types for an expression. The NAV *Developer and IT Pro Help* provides good information on type conversion rules. Search for the phrase NAV 2017 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
@	Case-insensitive

Explanations regarding uses of the set of operator symbols in the preceding table follows:

- 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`), a control is a member of a page (`PageName.ControlName`) or report (`ReportName.ControlName`), or a function is a member of an object (`Objectname.FunctionName`).
- Parentheses ( ) and brackets [ ] can 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 using brackets, as well as with the `IN` function earlier. Parentheses are also used to enclose the parameters in a function call as shown in the following code snippet:

```
Objectname.FunctionName (Param1, Param2, Param3);
```

- The Scope operator is a two-character sequence consisting of 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 table Radio Show , we have an **Option** field defined, called **Frequency**, with Option string values of (blank), Hourly, Daily, Weekly, and Monthly. Those values will be stored as integers 0, 1, 2, 3, or 4, but we can use the strings to refer to them in code, which makes our code more self-documenting. The Scope operator allows us to refer to Frequency::Hourly (rather than 1) and Frequency::Monthly (rather than 4). These constructs are translated by the compiler to 1 and 4, respectively. If we want to type fewer characters when entering code, we could enter just enough of the Option string value to be unique, letting the compiler automatically fill in the rest when we next save, compile, close, and reopen the object. In a similar fashion, we can refer to objects in the format (Object Type::"ObjectName") to be translated to the object number, as in the following code snippet:

```
PAGE.RUN(PAGE::"Bin List"); is equivalent to  
PAGE.RUN(7303);
```

- 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 lowercase 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 (lowercase) by a...a\*. Don't underestimate the power of the range operator. For more information on filtering syntax, refer to the *NAV Developer and IT Pro Help* section, *Entering Criteria in Filters* (<https://msdn.microsoft.com/en-us/dynamics-nav/entering-criteria-in-filters>).

## Arithmetic operators and functions

The Arithmetic operators include the following set of operators, as shown in the following table:

Arithmetic Operators		
Symbol	Action	Data Types
+	Addition	Numeric, Date, Time, Text and Code (concatenation),
-	Subtraction	Numeric, Date, Time
*	Multiplication	Numeric
/	Division	Numeric
<b>DIV</b>	Integer Division (provides only the integer portion of the quotient of a division calculation)	Numeric
<b>MOD</b>	Modulus (provides only the integer remainder of a division calculation)	Numeric

As we can see, in the **Data Types** column, these operators can be used on various data types. Numeric types include 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, will be 3:

```
MODIntegerValue := BigNumber MOD 60;
```

The contents of **MODIntegerValue**, after executing the preceding statement, will be 20.

The syntax for these **DIV** and **MOD** statements is as follows:

```
IntegerQuotient := IntegerDividend DIV IntegerDivisor;  
IntegerModulus := IntegerDividend MOD IntegerDivisor;
```

## Boolean operators

Boolean operators only operate on expressions that can be evaluated as Boolean. These are shown in the following table:

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)`. The following operators can be used:

Relational Operators	
Symbol	Evaluation
<	Less than
>	Greater than
<=	Less than or Equal to
>=	Greater than or Equal to
=	Equal to
$\diamond$	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 Valueset as a container of a defined set of individual values, expressions, or other Value sets. 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 ['5'..'9']  
  
"FA Posting Date" IN [01010001D..12312008D]
```

Here is another example of what the `IN` operator 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`. If the value of `TestString` were `z`, then this expression would evaluate to `FALSE`. Note that the Data Type of the search value must be the same as the Data Type of the Valueset.

## 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 *NAV Developer and IT Pro Help* section, *C/AL Operators* (<https://msdn.microsoft.com/en-us/dynamics-nav/c-al-operators>), but for convenience, the information is repeated here:

C/AL Operator Precedence Hierarchy		
Sequence	Symbols	Description
1	.	Member (Fields in Records, etc)
	[ ]	Indexing
	( )	Parenthetical Grouping
	::	Scope
	@	Case-insensitive
2	NOT + -	Unary instances of: Logical Not Positive value Negating value
3	*	Multiplication
	/	Division
	DIV	Integer division
	MOD	Modulus
	AND	Logical AND
	XOR	Logical Exclusive OR
4	+	Addition or Concatenation
	-	Subtraction
	OR	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

For complex expressions, we should always freely use parentheses to make sure the expressions are evaluated the way we intend.

## Frequently used C/AL functions

It's time to learn some more of the standard functions provided by C/SIDE. We will focus on the following short list of frequently used functions: MESSAGE, ERROR, CONFIRM, and STRMENU.

There is a group of functions in C/AL called Dialog functions. The purpose of these functions is to allow communications, that is, dialog, between the system and the user. In addition, the Dialog functions can be useful for quick and simple testing/debugging. In order to make it easier for us to proceed with our next level of C/AL development work, we'll take time now to learn about those four dialog functions. None of these functions will operate if the C/AL code is running on the NAV Application Server as it has no GUI available. To handle that situation in previous versions of NAV, the Dialog function statements had to be conditioned with the GUIALLOWED function to check whether or not the code is running in a GUI allowed environment. If the code was being used in a Web Service or NAS, it would not be GUIALLOWED. However, in NAV 2017, NAS and Web Services simply ignore 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 numbers 1 through 10, located within a message text string. This could look like the following code snippet, assuming the local currency was defined as USD:

```
MESSAGE ('A message + a data element to display = %1', "OrderAmount");
```

If the OrderAmount value was \$100.53, the output from the preceding code would be as follows:

```
A message + a data element to display = $100.53
```

We can have up to 10 substitution strings in one dialog function. The use of substitution strings and their associated display values is optional. We can use any one of the dialog functions simply to display a completely predefined text message with nothing that is variable. Use of **Text Constant** (accessed through **View | C/AL Globals** in the **Text Constants** tab) for the message is recommended as it makes maintenance and multilanguage enabling easier.

## MESSAGE function

MESSAGE is easy to use for the display of transient data and can be placed almost anywhere in our 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 we 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 elementary trace tool. We can program the display of messages to occur only under particular circumstances and use them to view either the flow of processing (by outputting simple identifying codes from different points in our logic) or to view the contents of particular data elements through multiple processing cycles.



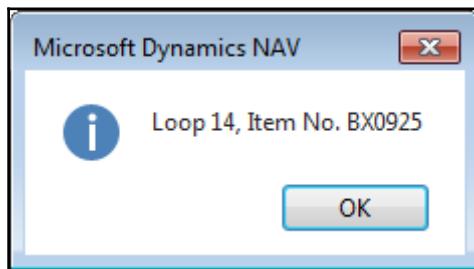
To display information to the user without interrupting them in their work, Notifications can be used. You can learn more about Notifications via this MSDN article: <https://msdn.microsoft.com/en-us/dynamics-nav/notifications-developing>.

MESSAGE has the following syntax: MESSAGE (String [, Value1] , ...), where there are as many ValueX entries as there are %X substitution strings (up to 10).

Here is a sample debugging message:

```
MESSAGE ('Loop %1, Item No. %2', LoopCounter, "Item No.");
```

The display would look as shown in the following screenshot (when the counter was 14 and the Item No. was BX0925):



When MESSAGE is used for debugging, make sure all messages are removed before releasing the object to production.

## ERROR function

When an ERROR function is invoked, the execution of the current process terminates, the message is immediately displayed, and the database returns to the status it had following the last (implicit or explicit) COMMIT function as though the process calling the ERROR function had not run at all.

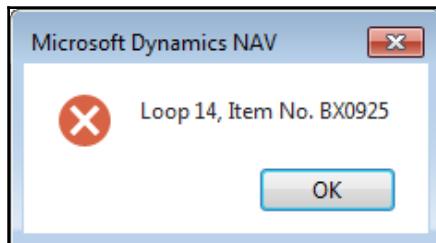


We can use the `ERROR` function in combination with the `MESSAGE` function to assist in repetitive testing. `MESSAGE` functions can be placed in code to show what is happening with an `ERROR` function placed just prior to where the process would normally complete. Because the `ERROR` function rolls back all database changes, this technique allows us to run through multiple tests against the same data without any time-consuming backup and restoration of our test data. The enhanced testing functionality built into NAV 2017 can accomplish the same things in a much more sophisticated fashion, but sometimes there's room for a temporary, simple approach.

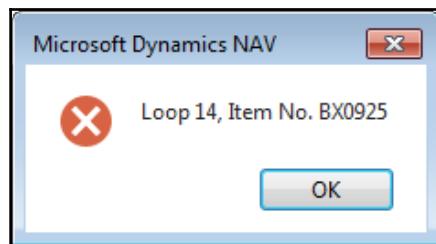
An `ERROR` function call is formatted almost exactly like a `MESSAGE` call. `ERROR` has the syntax `ERROR (String [, Value1] ,... ])`, where there are as many `ValueX` entries as there are %X substitution strings (up to 10). If the preceding `MESSAGE` was an `ERROR` function instead, the code line would be follows:

```
ERROR('Loop %1, Item No. %2',LoopCounter,"Item No.");
```

The display would look like the following screenshot:



The big X in a bold red circle tells us that this was an `ERROR` message, but some users might not immediately realize that. We can increase the ease of `ERROR` message recognition by including the word `ERROR` in our message, as shown in the following screenshot:



Even in the best of circumstances, it is difficult for a 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. For example, 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 our job, as developers, is to ensure that our systems communicate clearly and completely.

## CONFIRM function

A third dialog function is the `CONFIRM` function. A `CONFIRM` function call causes processing to stop until the user responds to the dialog. In `CONFIRM`, we will include a question in our text because the function provides **Yes** and **No** button options. The application logic can then be conditioned on the user's response.



We can also use `CONFIRM` as a simple 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. Execution of a `CONFIRM` function will also cause any pending `MESSAGE` function outputs to be displayed before the `CONFIRM` function displays. Combined with `MESSAGE` and `ERROR`, creative use of `CONFIRM` can add to our elementary debugging/diagnostic toolkit.

`CONFIRM` has the following syntax:

```
BooleanValue := CONFIRM(String [, Default] [, Value1] ,...)
```

When we do not specify a value for `Default`, the system will choose `FALSE` (which displays as **No**). We should almost always choose the `Default` option as it will do no damage if accepted inadvertently by an inattentive user. The `Default` choice is `FALSE`, which is often the safest choice (but `TRUE` may be specified by the programmer). There are as many `ValueX` entries as there are `%X` substitution strings (up to ten).

If we just code `OK := CONFIRM(String)`, the default choice will be `False`.

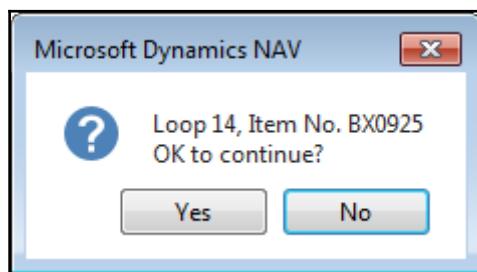


Note that `True` and `False` appear on screen as the active language equivalent of **Yes** and **No** (a feature that is consistent throughout NAV for C/AL Boolean values displayed from NAV code but not for RDLC report controls displayed by the report viewer - see *How to: Change the Printed Values of Boolean Variables*) at  
<https://msdn.microsoft.com/en-us/dynamics-nav/how-to--change-the-printed-values-of-boolean-variables>.

A CONFIRM function call with a similar content to the preceding examples might look like the following sample for the code and the display:

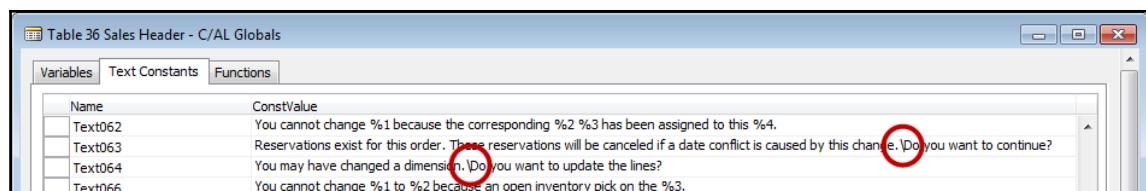
```
Answer := CONFIRM('Loop %1, Item No. %2OK to  
continue?', TRUE, LoopCounter, "Item No.");
```

The output screen is shown in the following screenshot:



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 for onscreen dialogs is the use of the backslash () embedded in the text. This forces a new line in the displayed message. This works throughout NAV screen display functions. The following are examples in **Text Constants** **Text063** and **Text064** in **Table 36 Sales Header**, which are shown in the following screenshot:



To display a backslash onscreen, we must put two of them in our message text string, (\ \).

## STRMENU function

A fourth dialog function is the STRMENU function. A STRMENU function call also causes processing to pause while the user responds to the dialog. The advantage of the STRMENU function is the ability to provide several choices, rather than just two (Yes or No). 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])
```

IntegerValue will contain the user's selection entry and OptionDefault is an integer representing which option will be selected by default when the menu displays. If we do not provide an OptionDefault value, the first option listed will be used as the default.

Instruction is a text string that will display above the list of options. If the user responds Cancel or presses the Esc key, the value returned by the function is 0.

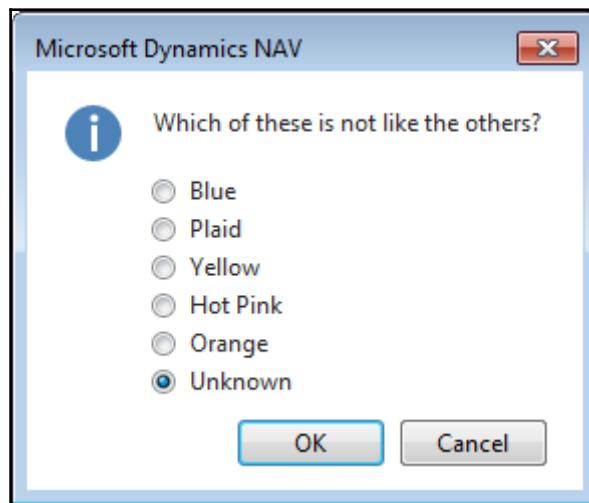
Use of the STRMENU function eliminates the need to use a Page object when asking the user to select from a limited set of options. It can also be utilized from within a report or Codeunit when calling a Page would restrict processing choices.

If we phrase our instruction as a question rather than simply an explanation, then we 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('Blue,Plaid,Yellow,Hot Pink,Orange,Unknown', 6,  
'Which of these is not like the others?');
```

The output screen is shown in the following screenshot:



Setting the default to 6 caused the sixth option (**Unknown**) to be the active selection when the menu is displayed.

## Record functions

Now, we will review some of the functions that we commonly use in Record processing.

### SETCURRENTKEY function

The syntax for SETCURRENTKEY is as follows:

```
[BooleanValue :=] Record.SETCURRENTKEY(FieldName1, [FieldName2], ... )
```

Because NAV 2017 is based on the SQL Server database, SETCURRENTKEY simply determines the order in which the data will be presented for processing. The actual choice of the index to be used for the query is made by the SQL Server Query Analyzer. For this reason, it is very important that the data and resources available to the SQL Server Query Analyzer are well maintained. This includes maintaining the statistics that are used by the Query Analyzer, as well as making sure efficient index options have been defined. Even though SQL Server picks the actual index, the developer's choice of the appropriate SETCURRENTKEY parameter can have a major effect on performance.



The fields used in the SETCURRENTKEY command do not have to match a key in the table definition. You can also use FlowFields as fields for SETCURRENTKEY, but expect slower performance than doing this.

The indexes that are defined in the SQL Server do not have to be the same as those defined in the C/AL table definition. For example, we can add additional indices in the SQL Server and not in C/AL, disable indices in the SQL Server but leave the matching keys enabled in C/AL, and so on. Any maintenance of the SQL Server indices should be done through the NAV Table Designer using the NAV keys and properties, not directly in SQL Server. Even though the system may operate without problem, any mismatch between the application system and the underlying database system makes maintenance and upgrades more difficult and error-prone.

NAV-defined keys are no longer required to support SIFT indexes because SQL Server can dynamically create required indices. However, depending on dynamic indices for larger datasets can lead to bad performance. Good design is still our responsibility as developers.

### SETRANGE function

The SETRANGE function provides the ability to set a simple range filter on a field. The 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 with only one value, that one value will act as both the From and To values. If SETRANGE is executed without any From or To values, it will clear the filters on the field. This is a common use of SETRANGE. Some examples of the SETRANGE function in code are as follows:

- Clear the filters on Item.No:

```
Item.SETRANGE ("No.");
```

- Filter to get only Items with a No. from 1300 through 1400:

```
Item.SETRANGE ("No.", '1300', '1400');
```

- Or with the variable values from LowVal through HiVal:

```
Item.SETRANGE ("No.", LowVal, HiVal);
```

In order to be effective in a Query, SETRANGE must be called before OPEN, SAVEASXML, and SAVEASCsv functions.

## SETFILTER function

SETFILTER is similar to, but much more flexible than, the SETRANGE function because it supports the application of any of the supported NAV filter functions to table fields.

SETFILTER syntax is as follows:

```
Record.SETFILTER(FieldName, FilterExpression [Value], ...);
```

The FilterExpression consists of a string (Text or Code) in standard NAV filter format, including any of the operators < > \* & | = in any legal combination. Replacement fields (%1, %2, ..., %9) are used to represent the values that will be inserted into the FilterExpression by the compiler to create an operating filter formatted as though it were entered from the keyboard. Just as with SETRANGE, prior to applying its filter, the SETFILTER function clears any filters that were previously set for the defined field, for example:

- Filter to get only Items with a No. from 1300 through 1400:

```
Item.SETFILTER ("No.", '%1..%2', '1300', '1400');
```

- Or with any of the variable values of `LowVal` or `MedVal` or `HiVal`:

```
Item.SETFILTER"No.",'%1|%2|%3',LowVal,MedVal,HiVal);
```

In order to be effective in a Query, `SETFILTER` must be called before the `OPEN`, `SAVEASXML`, and `SAVEASCsv` functions.

## 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 for the `GET` function is the primary key value (or all the 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 (no record found) and the statement is not part of an `IF` statement, the process will terminate with a runtime error. Typically, therefore, the `GET` function is encased in an `IF` statement structured like the following code snippet:

```
IF Customer.GET(NewCustNo) THEN ...
```



GET data retrieval is not constrained by filters, except for security filters (see *How to: Set Security Filters* at <https://msdn.microsoft.com/en-us/dynamics-nav/how-to--set-security-filters>). If there is a matching record in the table, GET will retrieve it.

## FIND Functions

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 through the Primary Key, looking for a single uniquely keyed entry. There are two forms of `FIND` functions in C/AL, one a remnant from a previous database structure and the other designed specifically to work efficiently with SQL Server. Both are supported, and we will find both in standard code.

The older version of the `FIND` function has the following syntax:

```
[BooleanValue :=] RecordName.FIND ( [Which] ).
```

The newer SQL Server-specific members of the `FIND` function family have slightly different syntax, as we will see shortly.

Just as with the `GET` function, assigning the `FIND` function result to a Boolean value is optional. However, in almost all of the cases, `FIND` is embedded in a condition that controls subsequent processing appropriately. Either way, it is important to structure our code to handle the instance where `FIND` is not successful.

`FIND` differs from `GET` in several important ways, some of which are as follows:

- `FIND` operates under the limits of whatever filters are applied on the subject field.
- `FIND` presents the data in the sequence of the key that is currently selected by default or by C/AL code.
- When `FIND` is used, the index used for the data reading is controlled by the SQL Server Query Analyzer.
- Different variations of the `FIND` function are designed specifically for use in different situations. This allows coding to be optimized for better SQL Server performance. All the `FIND` functions are described further in the Help section *C/AL Database Functions and Performance on SQL Server* at <https://msdn.microsoft.com/en-us/library/dd355237.aspx>.

The forms of `FIND` are as follows:

- `FIND ('-')`: This finds the first record in a table that satisfies the defined filter and current key.
- `FINDFIRST`: This finds the first record in a table that satisfies the defined filter and defined key choice. Conceptually, it is equivalent to the `FIND ('-')` for a single record read but better for SQL Server when a filter or range is applied.
- `FIND ('+')`: This finds the last record in a table that satisfies the defined filter and defined key choice. Often this is not an efficient option for SQL Server because it causes it to read 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`: This finds the last record in a table that satisfies the defined filter and current key. It is conceptually equivalent to `FIND ('+')` but, often, much better for SQL Server as it reads a single record, not a set of records.

- FINDSET: This is the most efficient way to read a set of records from SQL Server for sequential processing within a specified filter and range. FINDSET allows defining the standard size of the read record cache as a setup parameter but, normally, it defaults to reading 50 records (table rows) for the first server call. The syntax includes two optional True/False parameters, as follows:

```
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. FINDSET clears any FlowFields in the records that are read.

## FIND ([Which]) options and the SQL Server alternates

Let's review the FIND function options syntax variations:

```
[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 table lists the Which parameter options and prerequisites:

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

The second table lists the FIND options, which are specific to SQL Server, as follows:

FINDxxxx options	FINDxxx action	Search and primary key value prerequisite before FINDxxx
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 <code>Record.FINDSET([For Update] [,UpdateKey])</code> Set <code>ForUpdate = True</code> if data to be updated Set <code>ForUpdate = True</code> and <code>UpdateKey = True</code> if a key field is to be updated If no parameter specified, both default to False	All must be specified

For all FIND options, the results always respect the applied filters.

The FIND ('-') function is sometimes used as the first step of reading a set of data, such as reading all the sales invoices for a single customer. In such a case, the NEXT function is used to trigger all subsequent data reads after the sequence is initiated with FIND ('-').

Generally, FINDSET should be used rather than FIND ('-'); however, FINDSET only works to read forward, not in reverse. We should use FINDFIRST if only the first record in the specified range is of interest.

One form of 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 using the FINDSET function 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". For the FIND-NEXT read loop, the NEXT function provides both the definition of how the read loop will advance through the table and when the loop is to exit.

When `DataTable.NEXT = 0`, it means there are no more records to be read. We have reached the end of the available 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 forward (or backward) 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` is set to 2, NAV will move ahead two records at a time and the process will only be presented with every other record.

`Step` can also be negative, in which case, NAV moves backwards through the table. This would allow us to execute a `FIND('+')` function for the end of the table, then a `NEXT(-1)` function to read backwards through the data. This is very useful if, for example, we need to read a table sorted ascending by date and want to access the most recent entries first.

## Conditional statements

Conditional statements are the heart of process flow structure and control.

## BEGIN-END compound statement

In C/AL, there are instances where the syntax only allows the use of a single statement. However, a design may require the execution of several (or many) code statements.

C/AL provides at least two ways to address this need. One method is to have the single statement call a function that contains multiple statements.

However, inline coding is often more efficient to run and 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
```

The C/AL code contained within a BEGIN - END block should be indented two characters, as shown in the preceding pseudocode snippet, to make it obvious that it is a block of code.

## IF-THEN-ELSE statement

IF is the basic conditional statement of most programming languages. It operates in C/AL much the same as how it works in 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 as follows:

```
IF <Condition> THEN <Statement-1> [ ELSE <Statement-2> ]
```

Note that the statements within an IF do not have terminating semicolons, unless they are contained in a BEGIN - END framework. IF statements can be nested so that conditionals are dependent on the evaluation of other conditionals. Obviously, one needs to be careful with such constructs, because it is easy to end up with convoluted code structures that are difficult to debug and difficult for the developers following us to understand. In the next chapter, we will review the CASE statement that can make some complicated conditionals much easier to format and understand.

As we work with NAV C/AL code, we will see that often <Condition> is really an expression built around a standard C/AL function. This approach is frequently used when the standard syntax for the function is "Boolean value, function expression". Some examples follow:

- 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 the BEGIN-END compound statements and the 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 with the NAV 2017 C/AL code editor, 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 all the instances, but they don't cause any problem and can make the code easier to read.

Some examples are as follows:

```
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

Now we'll add some C/AL code to objects we've created for our WDTU application.

## Adding field validation to a table

In Chapter 4, *Pages - the Interactive Interface*, we created Table 50010 - Radio Show Fan. We've decided that we want to be able to use this list for promotional activities, such as having drawings for concert tickets. Of course, we want to send the tickets to the winners at their mailing addresses. We didn't include those fields originally in our table design, so we must add them now. To keep our design consistent with the standard product, we will model those fields after the equivalent ones in Table 18 - Customer. Our updated Table 50010 will look like the following screenshot:

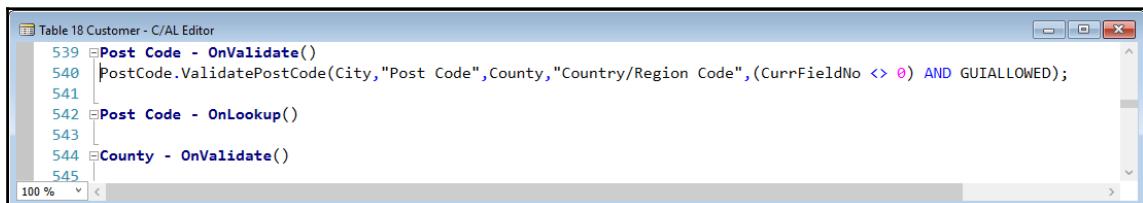
E..	Field No.	Field Name	Data Type	Length	Description
✓	10	No.	Code	20	
✓	20	Radio Show Code	Code	20	
✓	30	Name	Text	50	
✓	40	Email	Text	250	
✓	50	Last Contacted	Date		
✓	60	Address	Text	50	
✓	70	Address 2	Text	50	
✓	80	City	Text	30	
✓	90	County	Text	30	
✓	100	Country/Region Code	Code	10	
▶ ✓	110	Post Code	Code	20	

Part of modeling our Table 50010 - Radio Show Fan fields on those in Table 18 - Customer is faithfully copying the applicable properties. For example, the TableRelation property for the **Post Code** field in Table 18 contains the following lines of code, which we should include for the **Post Code** field in Table 50010:

```
IF (Country/Region Code=CONST()) "Post Code" ELSE IF (Country/Region Code
= FILTER(<>'') "Post Code" WHERE (Country/Region Code = FIELD
(Country/Region Code)) )
```

When a Radio Show Fan record is added or the **Post Code** field is changed, we would like to update the appropriate address information. Let's start with some code in a Validation trigger of our table.

As we model the address fields for our Fan record on the standard Customer table, let's look at the Customer table to see how **Post Code** validation is handled there. We can access that code through the Table Designer via **Object Designer** | **Table** | **Table 18 - Customer** | **Design** | **Field 91 - Post Code** and press F9. The following screen is displayed:

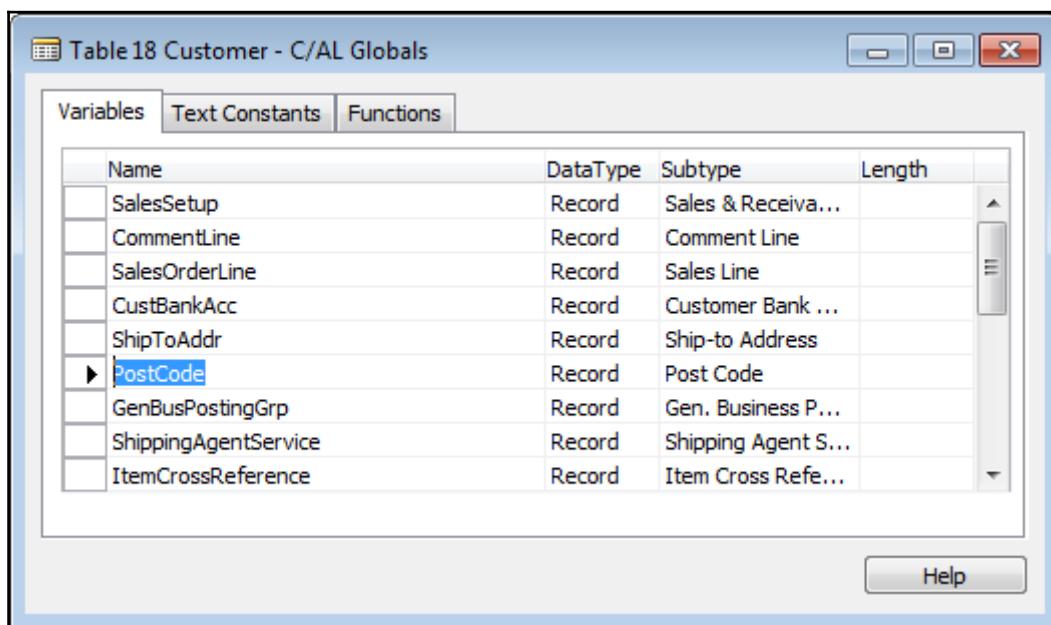


```

Table 18 Customer - C/AL Editor
539 [Post Code - OnValidate()]
540 |PostCode.ValidatePostCode(City,"Post Code",County,"Country/Region Code", (CurrFieldNo <> 0) AND GUIALLOWED);
541
542 [Post Code - OnLookup()]
543
544 [County - OnValidate()]
545
100 %

```

Looking at this C/AL code, we can see that the **OnValidate** trigger contains a call to a function in another object identified as **PostCode**. To find out what object **PostCode** actually is, we need to look in the **C/AL Globals**, which we have sometimes referred to in this book as part of Working Storage, as shown in the following screenshot:



Name	DataType	Subtype	Length
SalesSetup	Record	Sales & Receiva...	
CommentLine	Record	Comment Line	
SalesOrderLine	Record	Sales Line	
CustBankAcc	Record	Customer Bank ...	
ShipToAddr	Record	Ship-to Address	
► PostCode	Record	Post Code	
GenBusPostingGrp	Record	Gen. Business P...	
ShippingAgentService	Record	Shipping Agent S...	
ItemCrossReference	Record	Item Cross Refe...	

We will 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 will follow that clue to the next stop: the Post Code table and the `ValidatePostCode` function that is used in the Customer Post Code validation trigger. 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 function 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:

The screenshot shows a Microsoft Dynamics NAV Table Designer window titled "Table 225 Post Code - Table Designer". The window contains a grid of five fields with the following data:

E..	Field No.	Field Name	Data Type	Length	Description
▶	1	Code	Code	20	
▼	2	City	Text	30	
▼	3	Search City	Code	30	
▼	4	Country/Region Code	Code	10	
▼	5	County	Text	30	

A "Help" button is located in the bottom right corner of the window.

Next, the C/AL code for the ValidatePostCode function:

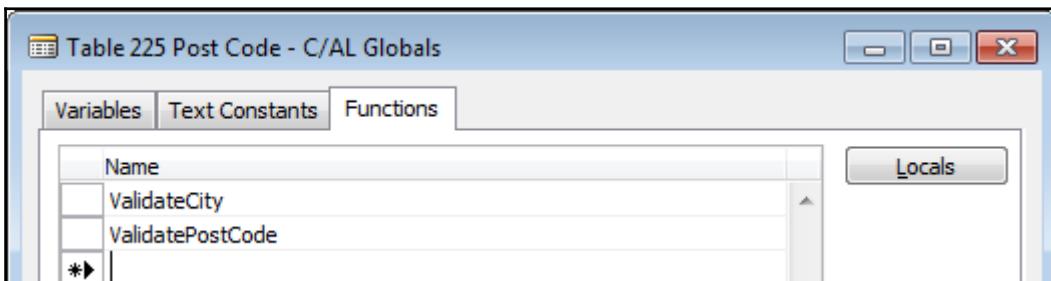


```
Table 225 Post Code - C/AL Editor
66 |ValidatePostCode(VAR City : Text[30];VAR PostCode : Code[20];VAR County : Text[30];VAR CountryCode
67 |IF PostCode <> '' THEN BEGIN
68 |  IF STRPOS(PostCode,'*') = STRLEN(PostCode) THEN
69 |    PostCodeRec.SETFILTER(Code,PostCode)
70 |  ELSE
71 |    PostCodeRec.SETRANGE(Code,PostCode);
72 |  IF NOT PostCodeRec.FINDFIRST THEN
73 |    EXIT;
74 |  PostCodeRec2.COPY(PostCodeRec);
75 |  IF UseDialog AND (PostCodeRec2.NEXT = 1) AND GUIALLOWED THEN
76 |    IF PAGE.RUNMODAL(PAGE::"Post Codes",PostCodeRec,PostCodeRec.Code) <> ACTION::LookupOK THEN
77 |      EXIT;
78 |  PostCode := PostCodeRec.Code;
79 |  City := PostCodeRec.City;
80 |  CountryCode := PostCodeRec."Country/Region Code";
81 |  County := PostCodeRec.County;
82 |END;
83 |
```

Using **Go to Definition** or pressing **Ctrl + F12** takes you directly from the code in the Customer table to the function in the Post Code table.

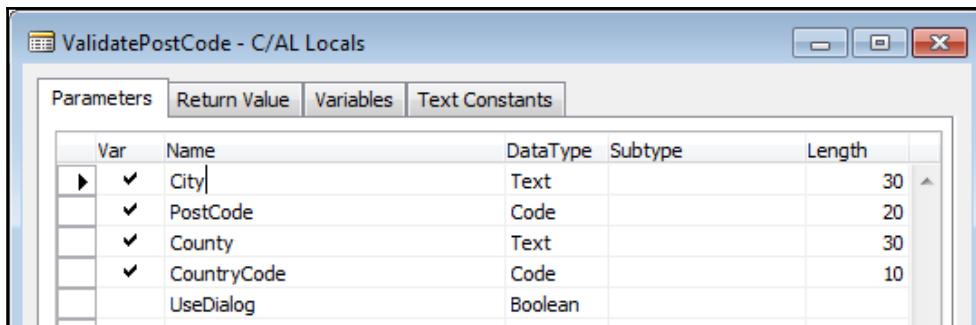


Now the list of callable functions available within the Post Code table (this isn't critical information, but helps us better understand the whole picture of the structure):

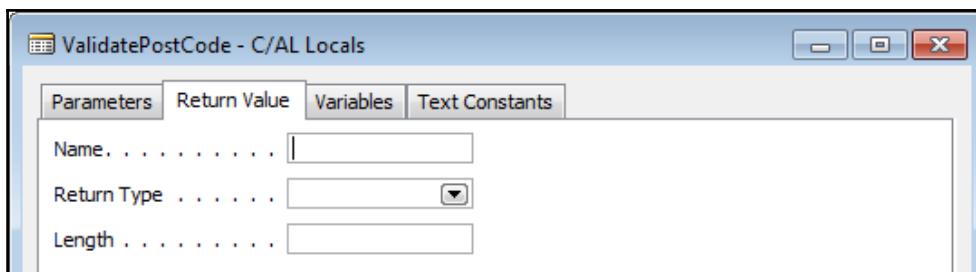


The screenshot shows the 'Table 225 Post Code - C/AL Globals' window. The 'Functions' tab is selected. A list box displays two entries: 'ValidateCity' and 'ValidatePostCode'. There is also a 'Locals' button on the right side of the window.

Finally, a look at the calling **Parameters** for the ValidatePostCode function:

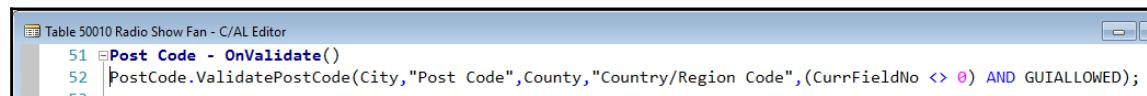


And at **Return Value** for the ValidatePostCode function:

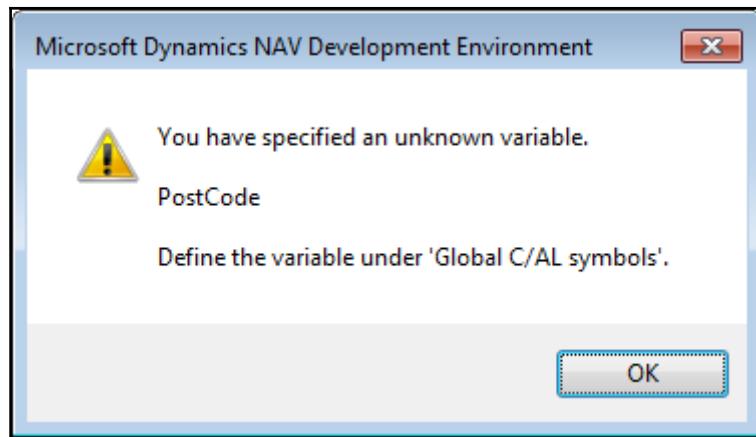


Doing some analysis of what we have dissected, we can see that the ValidatePostCode function call uses five calling **Parameters**. There is no **Return Value**. The function avoids the need for a **Return Value** by passing four of the parameters "by Reference" (not "by Value"), as we can tell by the checkmark in the **Var** column on the **Parameters** tab. The function code updates the parameters that reference data elements in the calling object. This interpretation is reinforced by studying the ValidatePostCode function C/AL code as well.

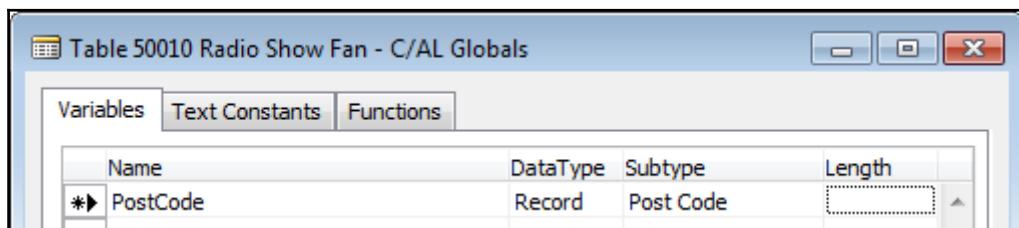
We conclude that we can just copy the code from the Post Code OnValidate trigger in the Customer table into the equivalent trigger in our Fan table. This will give us the Post Code maintenance we want. The result looks like the following screenshot (the variable CurrFieldNo is a system-defined Variable left over from previous versions retained for compatibility reasons):



If we press *F11* at this point, we will get an error message indicating that the variable **PostCode** is not defined, as shown in the following screenshot:



Obviously, we will need to attend to this. The answer is shown in the next screenshot in the form of the **PostCode** Global Variable definition:



After we save this change (by simply moving focus from the new line of code to another line on the form or closing the form), press *F11* again. We should get no reaction other than a brief cursor blink when the object is compiled.

Because we haven't created a page for maintenance of Table 50010, we will test our work by running the table. All we need to do is move to the **Post Code** field, click on it, and choose an entry from the displayed list of codes. The result should be the population of the **Post Code** field, the **Country/Region Code** field, and the **City** field. If we fill in the new data fields for some Fan records, our **Radio Show Fan** table could look like the following screenshot:

Radio Show Fan										Type to filter (F3)	No.	▼ ↗	▼ ↘
▲ Radio Show Code	Name	Email	Last Contacted	Address	Address 2	City	County	Country/Re... Code	Post Code	No filters applied			
RS003	Michael Jones	m.jones@gmail.com	10/3/2016	6542 Oak Hill Lane	3rd Floor	Chicago		US	US-IL 61236				
RS002	Maryann Smith	smith925@tigerfire.com	9/25/2016	1123 E Birminham		Sydney, NSW	NSW	AU	AU-2000				
RS003	Thomas Nielsen	navitop10@dankinder.com	4/6/2016	653-2 Mertel	Apt 14a	Düsseldorf		DE	DE-40593				
RS002	Andrew Patrick	apatrick921@eastiu.edu	10/3/2016	79A Albert Way		Atlanta		US	US-GA 31772				

We've accomplished our goal. The way we've done it may seem disappointing. It didn't feel like we really designed a solution or wrote any code. What we did was find where in NAV the same problem was already solved, figured out how that solution worked, cloned it into our object, and we were done.

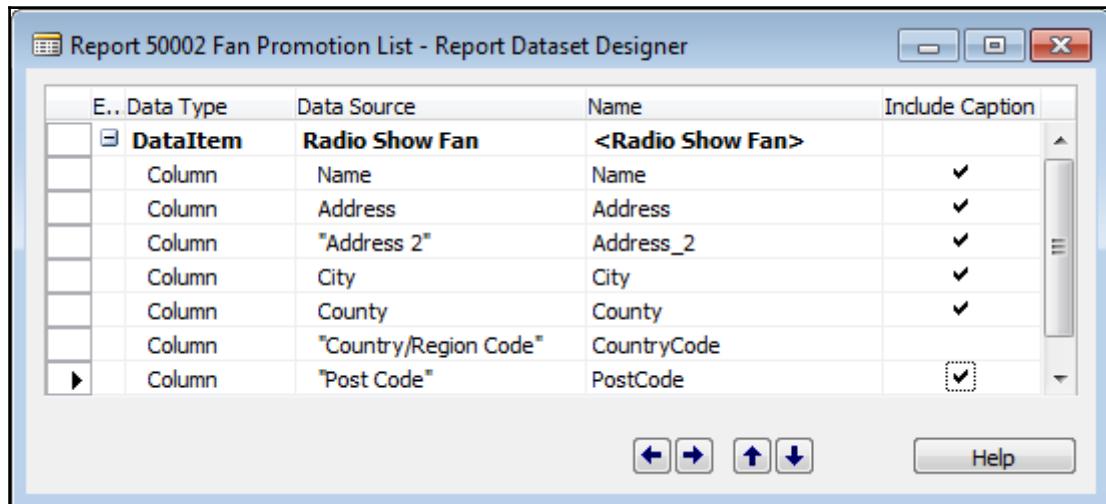
Each time we start this approach, we should look at the defined Patterns (<https://community.dynamics.com/nav/w/designpatterns/105.nav-design-patterns-repository.aspx>) to see if any Patterns fit our situation. The benefit of starting with a Pattern is that the general structural definition is defined for how this function should be done within NAV. Whether you find a matching Pattern or not, the next step is to find and study applicable C/AL code within NAV.

Obviously, this approach doesn't work every time. However, every time it does work is a small triumph of efficiency. This helps us keep the structure of our solution consistent with the standard product, reuse existing code constructs, and minimize the debugging effort and chances of production problems. In addition, our modifications are more likely to work even if the standard base application function changes in a future version.

## Adding code to a report

Most reports require some embedded logic to process user-selected options, calculate values, or access data in related tables. To illustrate some possibilities, we will extend our WDTU application to add a new report.

To support promotions giving away posters, concert tickets, and so on, we must further enhance the **Radio Show Fan** table and create a new report to generate mailing information from it. Our first step is to create a new report in the C/SIDE Report Designer, define the data fields we want to include for mailings (including a global variable of `CountryName`), then save and compile the result as **Report 50002 - Fan Promotion List**, as shown in the following screenshot:



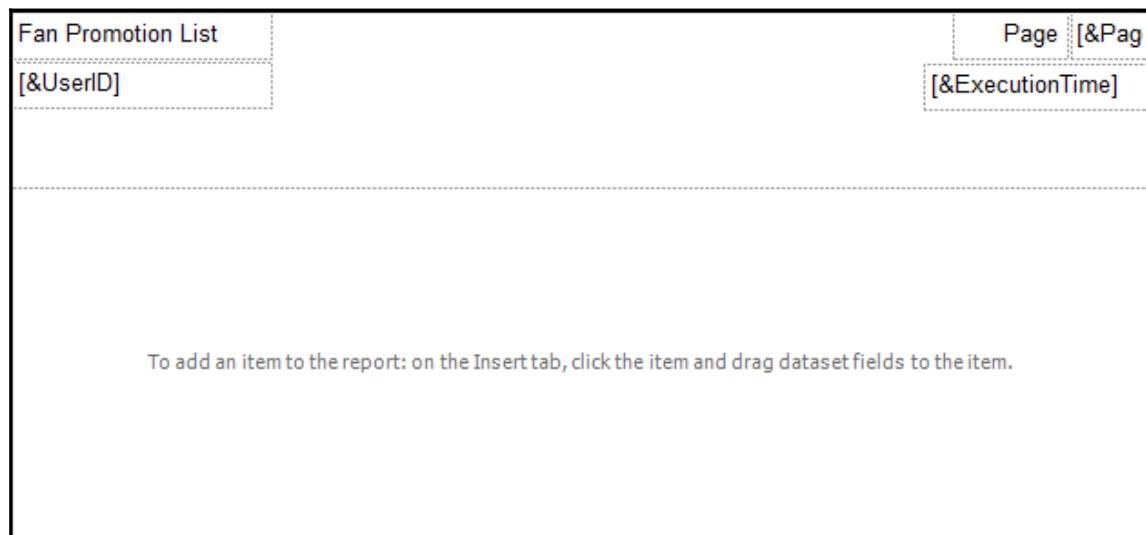
## Lay out the new Report Heading

Next, we will begin the design of the report layout in Visual Studio. From the C/SIDE Report Designer, we will click on **View | Layout** to open Visual Studio, ready to begin work on our layout. We'll begin by defining a report header.

Right-click on the layout work area (in the middle of the screen display), click on **Insert**, then select **Page Header**. On the left side of the menu, **ReportData** is displayed. Click on **Datasets** to display the **DataSetResult**. Depending on what information we want to appear in the header, we might use fields from various parts of the **ReportData**. If we had defined a **Label** to use for our Report Header, we could have done a drag and drop from the **Label** in the **Parameters** list. However, in our example, we dragged in a **Text Box** from the menu ribbon that we placed in the upper-left corner of the layout work area. We then typed a report name into that text box. Most of the other header fields were brought in from the **Report Data Built-in Fields** section. We used **Execution Time**, **Page Number**, and **User ID** fields. We added another **Text Box** for the word **Page** in front of the **Page Number**.

Entering text data directly into the Visual Studio layout for heading labels (as we did here with the Report Header and Page label) is only appropriate for beginners or for reports that are for very short-term use. Good report design practice requires that such values are defined in the C/SIDE Report Designer, where multi-language is supported and where such fields should be maintained. In the C/SIDE Report Designer, those values can be entered and maintained in the Report Label Designer that is accessed by navigating through **View | Labels**.

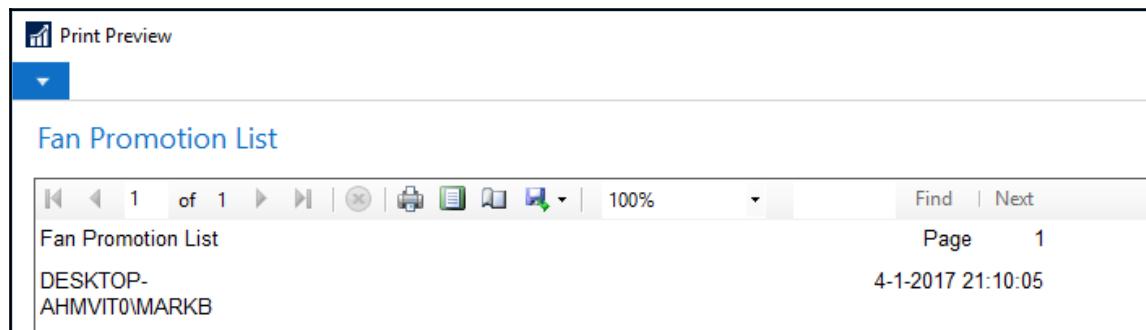
In the process of working on this sample report, we might want a different layout to use Labels or add other features. Feel free to experiment and design your header to suit your own preferences. You will learn by the results of your experiments. A sample result is shown in the following screenshot:



## Saving and testing

At this point, it's time to save and test what we've done so far. Exit from the Report Builder. Save the report layout changes, exit the C/SIDE Report Designer, and save and compile the report object.

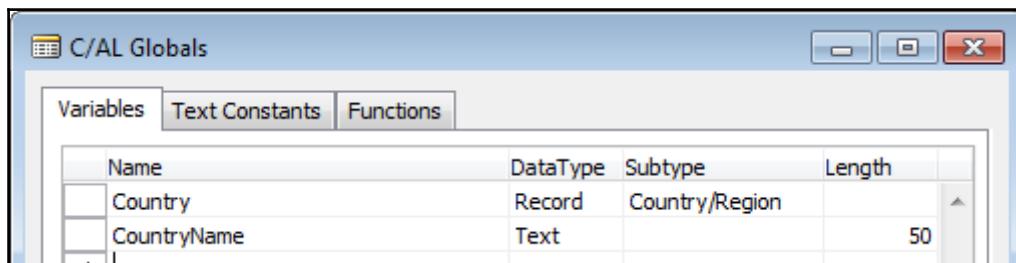
This first test is very simple (assuming it works). Run Report 50002. The **Report Request Page** will appear in the RoleTailored client. Click on **Preview** to see the Report display onscreen. The layout shown in the preceding screenshot will result in the following report page (or something similar):



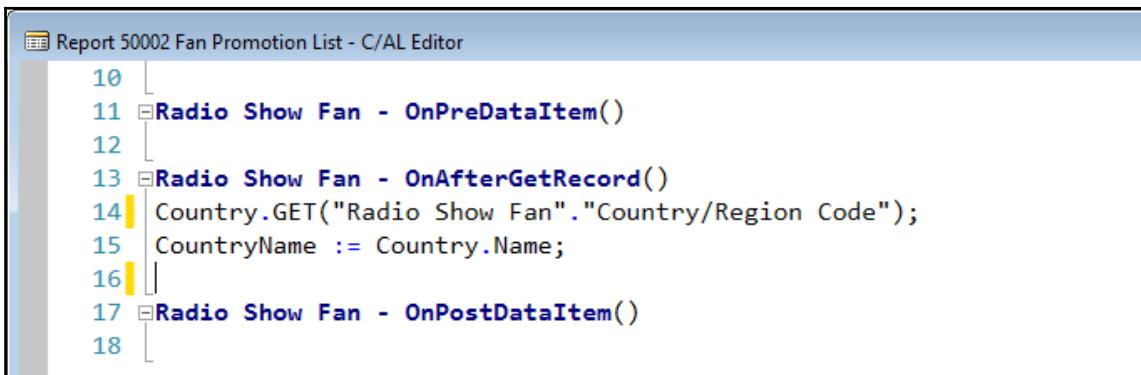
## Lookup related table data

Once we have a successful test of the report (heading only), we'll move on to laying out the body of the report. As we think through the data, we will want to include a mailing address, such as Name, Address, Address 2, City, County (State), Country Name, and Post Code, and realize that our table data includes Country Code, not Country Name. So, we will look up the Country Name from the Country/Region table (Table 9). Let's take care of that now.

First, we'll add a couple of Global Variables to our report. One of them will allow us to access the Country/Region table, and the other will act as a holding place for the Country Name data we get from that table, as shown in the following screenshot:



Each time we read a Radio Show Fan record, we'll look up the Country Name for that fan and store it in **CountryName** by entering the following code snippet:



The screenshot shows the C/AL Editor interface with a title bar "Report 50002 Fan Promotion List - C/AL Editor". The code area contains the following C/AL code:

```
10
11 Radio Show Fan - OnPreDataItem()
12
13 Radio Show Fan - OnAfterGetRecord()
14 Country.GET("Radio Show Fan"."Country/Region Code");
15 CountryName := Country.Name;
16
17 Radio Show Fan - OnPostDataItem()
18
```

Now, we can add the `CountryName` variable to the list of data elements attached to the DataItem Radio Show Fan so it will be included in the data passed to the Report Builder and, when the report is run, to the Report Viewer.

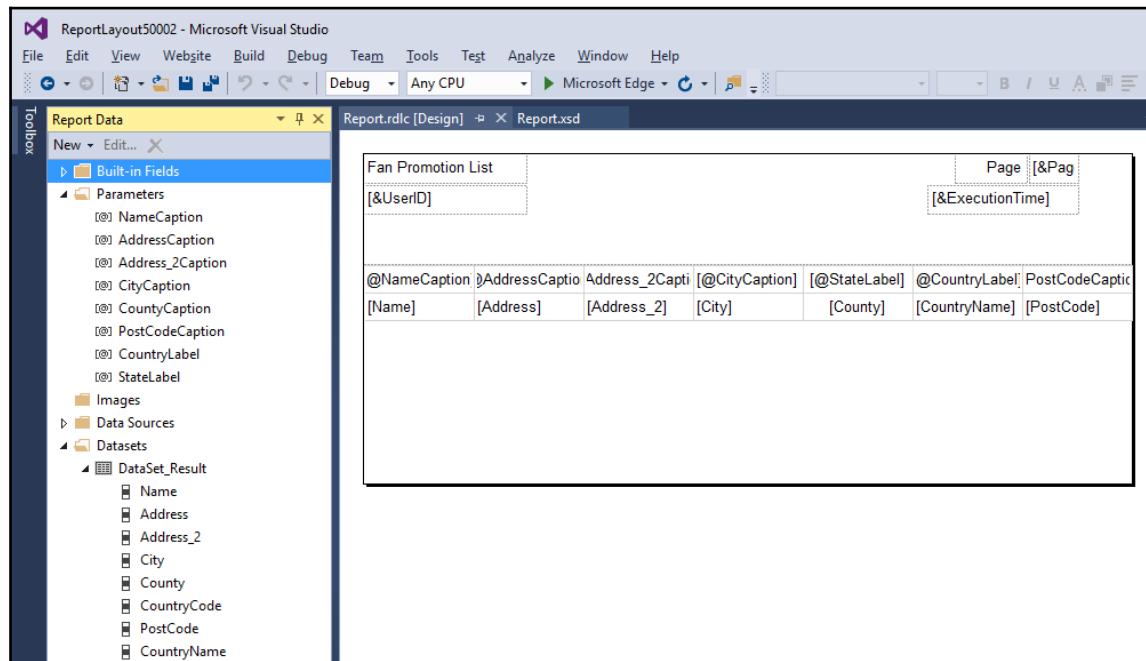
While what we've done will probably work most of the time, how could it be made better? For one thing, shouldn't we handle the situation where there is no `Country/Region Code` in the fan record? Do we really need to move the country name to a Global variable instead of simply reporting it directly from the `Country/Region` record?

Both of these issues could be handled better. Look up the `GET` function in Help to see what should be done in terms of error handling. Additionally, after we work through the report as we're doing it here, enhance it by eliminating use of the `CountryName` Global variable. For now, let's just move on to completing an initial version of our report by creating the rest of our report layout in Visual Studio.

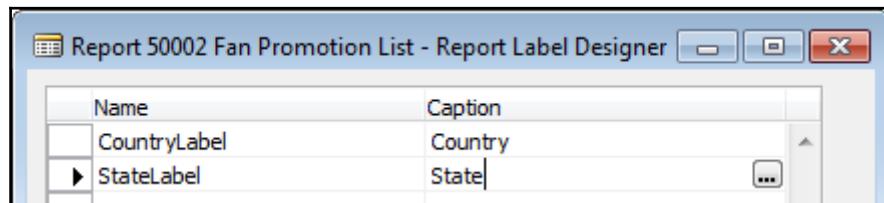
## Laying out the new report Body

Open the report layout in Visual Studio. From the Ribbon, we'll grab a Table and drag it into the layout work area for the report body. The Table starts with only three columns. After positioning the Table to the top left of the body, we will add four more columns to accommodate the seven data fields that we want to include for each mailing address.

We will drag a data field from the DataSet\_Result into each of the Data Row Text Boxes (the bottom row). In the top row, captions will appear. Where we want the displayed captions to be different than what fills in automatically, we'll either type in what we want (not very sophisticated) or delete the default captions and drag in captions from the **Parameters** list, as shown in the following screenshot:



Among our caption options are the **CountryLabel** and **StateLabel**, which we see in the preceding and following screenshots. These are the results of defining Labels in the C/SIDE RD Report Label Designer:



## Saving and testing

After we lay out, Save and Exit, Update, Save and Compile, it's time to do another test Run of our report in process. If we simply Preview without doing any filtering, we should see all of our test data address information (complete with Country Name), as shown in the following screenshot:

The screenshot shows a Microsoft Dynamics NAV report titled "Fan Promotion List". At the top, there are navigation buttons for Print Preview, a dropdown menu, and a toolbar with various icons. Below the toolbar, the report title "Fan Promotion List" is displayed, along with the date and time "4-1-2017 21:14:05". The report content area contains two rows of data. The first row shows "Maryann Smith" with "Wien" in the City column, "Austria" in the Country column, and "AT-1100" in the Post Code column. The second row shows "Andrew Good" with empty values in the Address, Address 2, City, State, and Post Code columns, and "Austria" in the Country column.

Name	Address	Address 2	City	State	Country	Post Code
Maryann Smith			Wien		Austria	AT-1100
Andrew Good					Austria	

## Handling user entered report options

Part of our report design includes allowing the user to choose fans based on some simple demographic data based on age and gender. We'll need to add two more fields to our **Radio Show Fan** table definition, one for **Gender** and the other for **Birth Date**, from which we can calculate the fan's age, as shown in the following screenshot:

Table 50010 Radio Show Fan - Table Designer

E..	Field No.	Field Name	Data Type	Length	Description
✓	10	No.	Code	20	
✓	20	Radio Show No.	Code	20	
✓	30	Name	Text	50	
✓	40	Email	Text	250	
✓	50	Last Contacted	Date		
✓	60	Address	Text	50	CDM06
✓	70	Address 2	Text	50	CDM06
✓	80	City	Text	30	CDM06
✓	90	County	Text	30	CDM06
✓	100	Country/Region Code	Code	10	CDM06
✓	110	Post Code	Code	20	CDM06
✓	120	Gender	Option		,Female, Male CDM06
▶ ✓	130	Birth Date	Date		CDM06

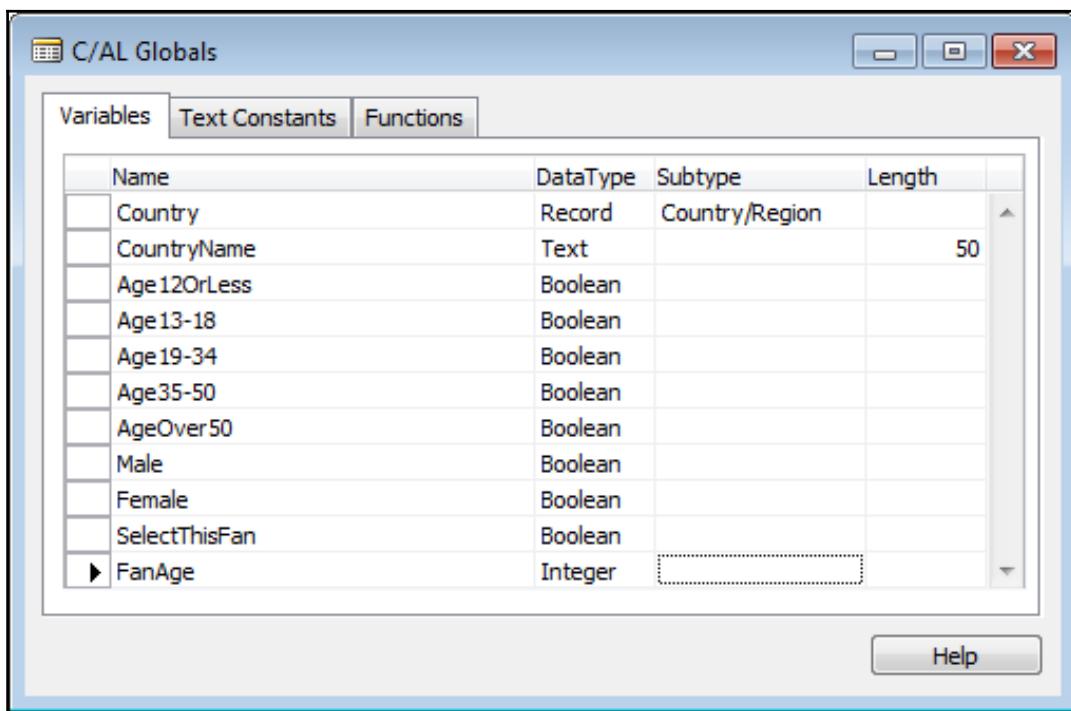
Help

This back and forth process of updating first one object, then a different one, then yet another, is typical of the NAV development process much of the time. Exceptions are those cases where either the task is so simple we think of everything the first time through, or the cases where we create a completely documented, full-featured design before any development starts (but nobody thinks of everything, there are always changes; our challenge is to keep the changes under control).



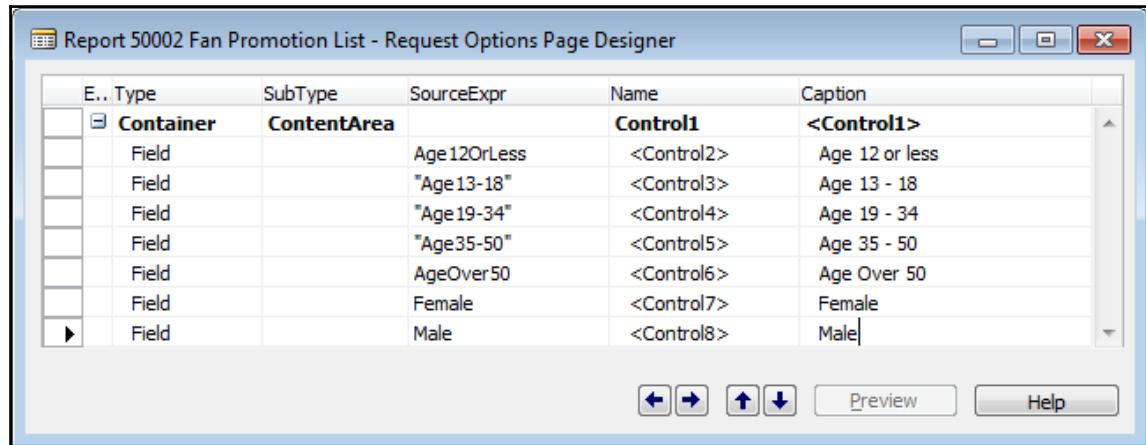
An advantage to the more flexible approach we are following is that it allows us to view (and share with others) intermediate versions of the application as it is developed. Design issues can be addressed as they come up and overlooked features that can be considered midstream. Two downsides are the very real possibility of scope creep (the project growing uncontrollably) and poorly organized code. Scope creep can be controlled by good project management. If the first pass through results in poorly organized code, then a thoughtful refactoring is appropriate, cleaning up the code while retaining the design.

In order for the user to choose which Fan demographics will be used to filter the Fan data for a particular promotion, we will have to create a Request Page for entry of the desired criteria. This, in turn, requires the definition of a set of Global Variables in our Report object to support the Request Page data entry fields and as working variables for the age calculation and Fan selection. We've decided that if a Fan fits any of the individual criteria, we will include them. This makes our logic simpler. Our final Global Variable list in Report 50002 looks like the following screenshot:



## Defining the Request Page

Now, let's define the Request Page. Click on **View | Request Page** and make the entries necessary to describe the page contents, as shown in the following screenshot:



## Finishing the processing code

Next, we will create the C/AL code to calculate a Fan's age (in years) based on their Birth Date and the current WORKDATE. The logic is simple--subtract the Birth Date from the WORKDATE. This gives a number of days. So, we will divide by 365 (not worrying about Leap Years) and round down to integer years (if someone is 25 years, 10 months, and 2 days old, we will just consider them 25). In the following code, we did the division as though the result were a decimal field. However, because our math is integer, we could have used the simpler expression:

```
FanAge := ((WORKDATE - "Birth Date") DIV 365);
```

Finally, we'll write the code to check each Fan record data against our selection criteria, determining if we want to include that fan in our output data (**SelectThisFan** set to **True**). This code will select each fan who fits any of the checked criteria; there is no combination logic here. The following is our commented C/AL code for Report 50002:

```

Report 50002 Fan Promotion List - C/AL Editor
11 11 Radio Show Fan - OnPreDataItem()
12 12
13 13 Radio Show Fan - OnAfterGetRecord()
14 14 //Look up the Country Name using the Country/Region Code
15 15 Country.GET("Radio Show Fan"."Country/Region Code");
16 16 CountryName := Country.Name;
17 17
18 18 //Calculate the fan's age
19 19 FanAge := ROUND(((WORKDATE - "Birth Date")/365),1.0,'<');
20 20
21 21 //Select Fans to receive promotional material
22 22 SelectThisFan := FALSE;
23 23 IF Age120rLess AND (FanAge <= 12) THEN
24 24   SelectThisFan := TRUE;
25 25 IF "Age13-18" AND (FanAge > 12) AND (FanAge < 19) THEN
26 26   SelectThisFan := TRUE;
27 27 IF "Age19-34" AND (FanAge > 18) AND (FanAge < 35) THEN
28 28   SelectThisFan := TRUE;
29 29 IF "Age35-50" AND (FanAge > 34) AND (FanAge < 51) THEN
30 30   SelectThisFan := TRUE;
31 31 IF AgeOver50 AND (FanAge > 50) THEN
32 32   SelectThisFan := TRUE;
33 33 IF Male AND (Gender = Gender::Male) THEN
34 34   SelectThisFan := TRUE;
35 35 IF Female AND (Gender = Gender::Female) THEN
36 36   SelectThisFan := TRUE;
37 37
38 38 //If this Fan not selected, skip this Fan record on report
39 39 IF SelectThisFan <> TRUE THEN
40 40   CurrReport.SKIP;

```

After this version of the report is successfully tested, enhance it. Make the report support choosing any of the options (as it is now) or, at user option, choose a combination of age range plus gender. Hint: add additional checkboxes to allow the user to control which set of logic will be applied. We should also change the code to use the CASE statements (rather than IF statements). The CASE statements often provide an easier to understand view of the logic.

## Testing the completed report

After we Save and Compile our report, we'll run it again. Now, we will get an expanded Request Option Page. After this, we've checkmarked a couple of the selection criteria, as shown in the following screenshot:

Age 12 or less:	<input type="checkbox"/>
Age 13 - 18:	<input type="checkbox"/>
Age 19 - 34:	<input checked="" type="checkbox"/>
Age 35 - 50:	<input type="checkbox"/>
Age Over 50:	<input type="checkbox"/>
Female:	<input checked="" type="checkbox"/>
Male:	<input type="checkbox"/>

Now, let's preview our report. Using the sample data previously illustrated, our report output shows two records in the following screenshot; one selected on the basis of Gender and the other on Age:

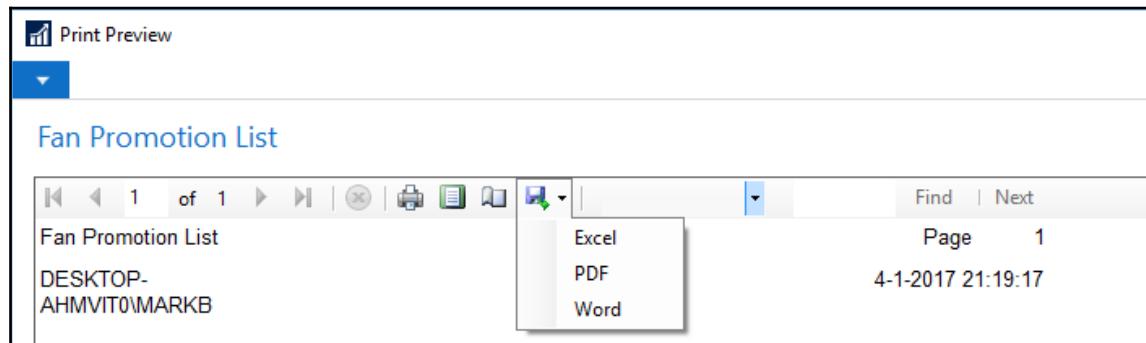
The screenshot shows a report preview window with the following details:

- Print Preview** button at the top left.
- Fan Promotion List** title at the top center.
- Toolbar with standard report navigation icons (back, forward, search, etc.) and a zoom level of 100%.
- Page header information:
  - Page 1 of 1
  - Date: 4-1-2017 21:19:17
  - User: DESKTOP-AHMVIT0\MARKB
- Table data:

Name	Address	Address 2	City	State	Country	Post Code
Andrea Good					Austria	

## Output to Excel

An easy way to get the data to a mailing list is now to output it to Excel, where we can easily manipulate it into a variety of formats without further programming, as shown in the following screenshot:



Here's what the output looks like in Excel:

	A	B	C	D	E	F	G	H	I	L	N
1	Fan Promotion List										Page 1
	DESKTOP-										4-1-2017 21:19:17
3	AHMVIT0\MARKB										
4											
6	Name	Address	Address 2		City	State	Country		Post Code		
7	Andrea Good						Austria				

At this point, we have a report that runs and is useful. It can be enhanced to support more complex selection criteria. As usual, there are a number of different ways to accomplish essentially the same result. Some of those paths would be significantly different for the developer, but nearly invisible to the user. Some might not even matter to the next developer who has to work on this report. What is important at this point 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 usually not very important.

## Review questions

1. All NAV objects can contain C/AL code. True or False?
2. Which object type has a wizard to "jump start" development? Choose one:
  - a) Page
  - b) XMLport
  - c) Table
  - d) Report
3. All C/AL Assignment statements include the symbol:= True or False?
4. One setting defines how parameters are passed to functions, whether a parameter is passed by reference or by value. Choose that one setting identity:
  - a) DataType
  - b) Subtype
  - c) Var
  - d) Value
5. If an object type has a Wizard, we must start with the Wizard before proceeding to the object Designer form. True or False?
6. The C/AL code cannot be inserted into the RDLC generated by the Visual Studio Report Designer (or the SQL Server Report Builder). True or False?
7. When a table definition is changed, the "Force" option should always be used when saving the changes. True or False?
8. Object numbers and names are so flexible that we can (and should) choose our own approach to numbering and naming. True or False?

9. In what formats can objects be exported? Choose two:
  - a) .fob
  - b) .txt
  - c) .NET
  - d) .XML
  - e) .gif
10. BEGIN - END are always required in IF statements. True or False?
11. Which object export format should be used to transmit updates to client sites?  
Choose one:
  - a) .fob
  - b) .txt
  - c) .NET
12. All NAV development work starts from the Object Designer. True or False?
13. Modifiable functions include which of the following? Choose two:
  - a) Application Management
  - b) DATE2MDY
  - c) Mail
  - d) STRLEN
14. Report heading text can either be typed in manually or brought into SSRB through Label Parameters. True or False?
15. Whenever possible, the controlling logic to manage data should be resident within the tables. True or False?

16. Filter wildcards include which three of the following?
  - a) ?
  - b) ::
  - c) \*
  - d) ^
  - e) @
17. Choice of the proper version of the FIND statement can make a significant difference in processing speed. True or False?
18. When we are working in the Object Designer, changing C/AL code, the Object Designer automatically backs up our work every few minutes so we don't have to do so. True or False?
19. 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?
20. Arithmetic operators and functions include which of the following? Choose two:
  - a) \*
  - b) >
  - c) =
  - d) /

# Summary

*"Furniture or gold can be taken away from you, but knowledge and a new language can easily be taken from one place to the other, and nobody can take them away from you."*

- David Schwarzer

In this chapter, we covered Object Designer navigation, along with navigation of the individual Designers, such as Table, Page, Report, and so on. We covered a number of C/AL language areas, including functions and how they may be used, variables of various types (both development and system), basic C/AL syntax, expressions, and operators. Some of the essential C/AL functions that we covered included user dialogs, SETRANGE filtering, GET, variations of FIND, 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 creating a new report that included embedded C/AL code and a Request Page. In the next chapter, we will expand our exploration and practice in the use of C/AL. We will learn about additional C/AL functions, flow control structures, input/output functions, and filtering.

# 7

## Intermediate C/AL

*"You need to 'listen deeply' - listen past what people say they want to hear what they need."*

*- Jon Meads*

*"People's behavior makes sense if you think about it in terms of their goals, needs, and motives."*

*- Thomas Mann*

In the previous chapter, you learned enough C/AL to create a basic operational set of code. In this chapter, you will learn more C/AL functions and pick up a few more good habits along the way. If you are getting started as a professional NAV Developer, C/SIDE's built-in C/AL functions represent a significant portion of the knowledge that you will need on a day-to-day basis. If you are a manager or consultant needing to know what NAV can do for your business or your customer, an understanding of these functions will help you too.

Our goal is to competently manage I/O, create moderately complex program logic structures, and understand data filtering and sorting as handled in NAV and C/AL. Because the functions and features in C/AL are designed for business and financial applications, we can do a surprising amount of ERP work in NAV with a relatively small number of language constructs.

Keep in mind that anything discussed in this chapter relates only indirectly to those portions of NAV objects that contain no C/AL, for example, **MenuSuites**, **SQL Server Report Builder (SSRB)**, and **Visual Studio Report Designer (VSRD)** report layouts. This chapter's goals are to accomplish the following:

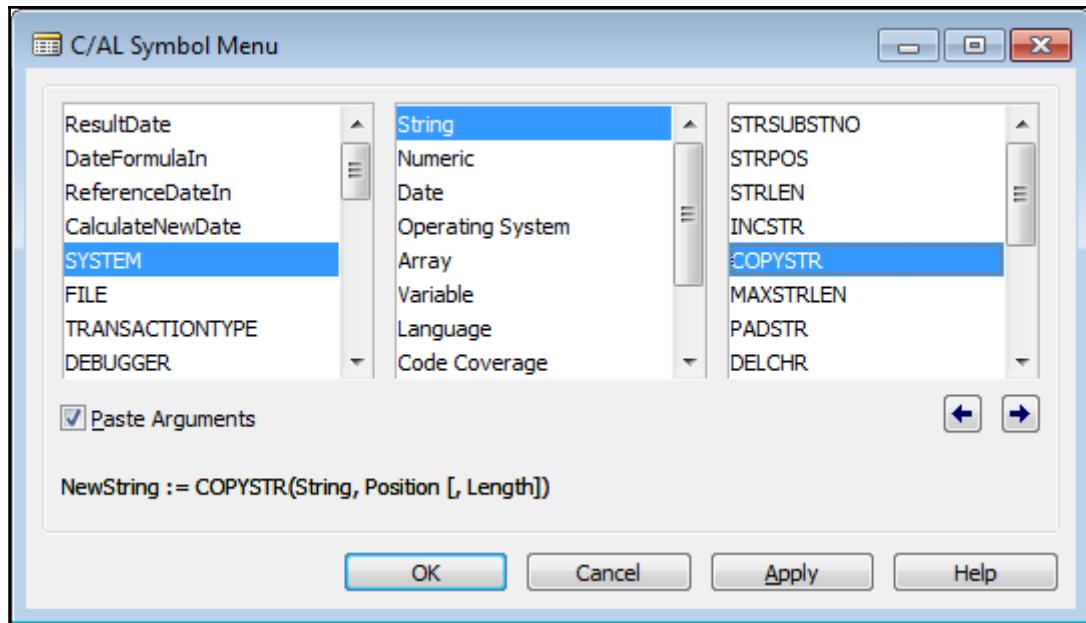
- Review some C/AL development basics
- Learn about a variety of useful (and widely used) C/AL functions
- Better understand of filtering
- Apply what you've learned to expand your WDTU application using some of the C/AL development tools

As described in Chapter 1, *Introduction to NAV 2017*, all internal NAV logic development is done in C/AL and all C/AL development is done in C/SIDE. Some user interface design is done using Visual Studio. It is also possible to have integrated .NET objects for a variety of purposes.

## C/AL Symbol Menu

As an Integrated Development Environment, C/SIDE contains a number of tools designed to make our C/AL development effort easier. One of these is the C/AL Symbol Menu. When we are in one of the Object Designers where C/AL code is supported, the **C/ALSymbolMenu** window can be accessed via either the menu option, **View | C/AL Symbol Menu**, or by pressing *F5*.

The three-column display has variables and object categories in the left column. If the entry in the left column is an object 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 part of the highlighted center-column entry. In a few cases (such as BLOB fields), additional information is 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** window is a very useful multi-purpose tool for the developer. It serves the following purposes:

- as a quick reference to the available C/AL functions
- documentation of the syntax of those functions
- access to **Help** on those functions
- display of what other systems refer to as the Symbol Table
- as a source of variable names or function structures to paste into our code



The code editor in NAV2017 is brought forward from Visual Studio and supports full Intellisense which provides automatic syntax guidance, variable references, word completion, and other coding assists. Some of these features are similar to or the same as capabilities provided by the C/AL Symbol Menu.

Use of the **C/AL Symbol Menu** window for reference purposes is especially helpful when we are a beginning C/AL developer, but also after we have become experienced developers. It is a guide to the inventory of available code tools, with some very handy built-in programming aids.

The **C/AL Symbol Menu** window displays the highlighted function's syntax at the bottom left of the screen. It also provides quick access to *Developer and IT Pro Help* to further study the highlighted function and its syntax. Pressing **F1** may just bring up the general *Developer and IT Pro Help* rather than a specific entry, or it may bring up an entry only somewhat related to the focus location.

The second use of the **C/AL Symbol Menu** window is as a symbol table. The symbol table for our object is visible in the left column of the **C/ALSymbolMenu** display. The displayed symbol set (the variable set), is context sensitive. It will include all system-defined symbols, all our **Global** symbols, and the **Local** symbols from the function that had focus at the time we accessed the **C/AL Symbol Menu**. Although 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 we have to know the name of the first Global symbol to determine the scope of a particular variable (that is, if an entry appears in the symbol list before the first Global, it is 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 option available. This function will be enabled if we 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 or highlighting and double-clicking. In each of these cases, the element with focus will be pasted into our 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 our code. If the element is a function whose syntax appears at the lower left of the screen, the result of the paste action (**Apply**, **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 our code. If the **Paste Arguments** checkbox is selected (as shown in the preceding screenshot), then the complete syntax string, as shown on screen, will be pasted into our code. This can be a very convenient way to create a template to help us enter the correct parameters with the correct syntactical structure and punctuation more quickly.

When we are in the **C/AL Symbol Menu**, we 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 we are creating or modifying software, we should always document what we have done. It is often difficult for developers to spend much time (that is, money) on documentation because many don't enjoy doing it and the benefits to customers are difficult to quantify in advance. A reasonable goal is to provide enough documentation so that a knowledgeable person can later understand what we have done as well as the reasons why.



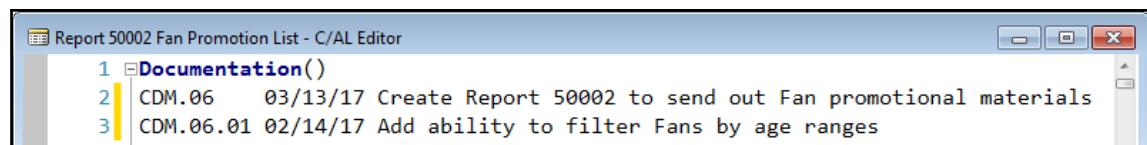
When packaging your solution as an extension, you are not allowed to add documentation to standard Microsoft objects.

If we choose good variable names, the C/AL code will tend to be self-documenting. If we lay our code out neatly, use indentation consistently, and localize logical elements in functions, then the flow of our code should be easy to read. We should also include comments that describe the functional reason for the change. This will help the next person working in this code not only be able to follow the logic of the code, but to understand the business reasons for that code.

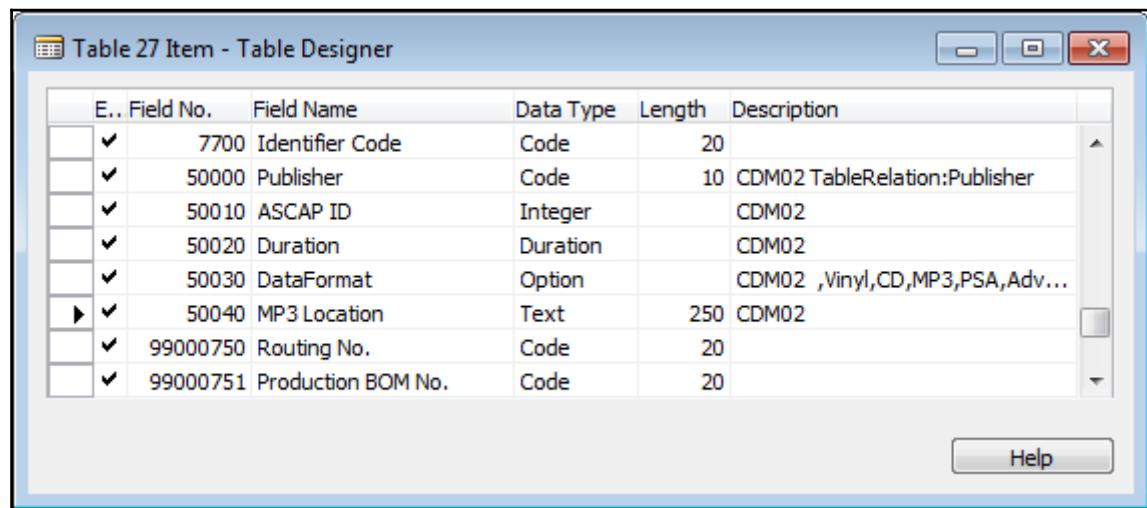
In case of a brand new function, a simple statement of purpose is often all that is necessary. In case of a modification, it is extremely useful to have comments providing a functional definition of what the change is intended to accomplish, as well as a description of what has been changed. If there is external documentation of the change, including a design specification, the comments in the code should 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 task of the next person to maintain or upgrade that code 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 we impose. This is a new report, which we created in Chapter 6, *Introduction to C/SIDE and C/AL*. The comment is coded to indicate the organization making the change (in this case, **CDM**) 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, we will start with **CDM.06.00**, followed by the date of the change. Some organizations would also include an identifier for the developer in the **Documentation** section comments:



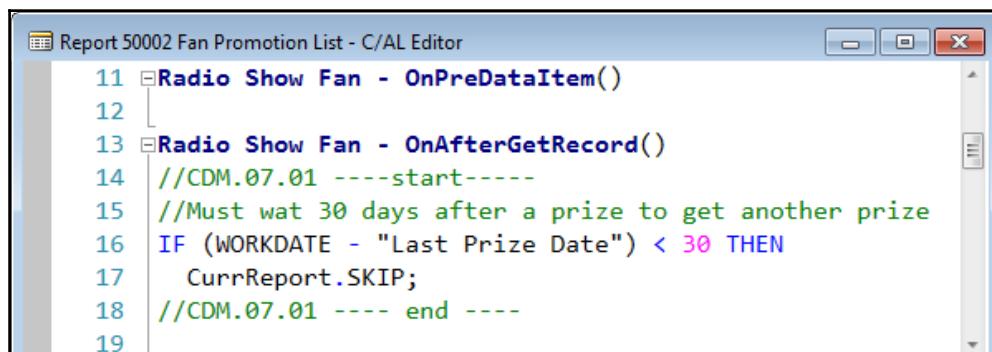
We can make up our own standard format that will identify the source and date of the work, but we should have a standard and use it. When we add a new data element to an existing table, the **Description** property should receive the same modification identifier that we would place in the code comments:



When we make a subsequent change to an object, we 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--it will be ignored. If the comment spans multiple physical lines, each line of the comment must also be preceded by two forward slashes.

In the following screenshot, we have used // to place comments inline to identify a change:



The screenshot shows the Microsoft Dynamics NAV C/AL Editor window titled "Report 50002 Fan Promotion List - C/AL Editor". The code editor displays the following C/AL code:

```
11 Radio Show Fan - OnPreDataItem()
12
13 Radio Show Fan - OnAfterGetRecord()
14 //CDM.07.01 -----start-----
15 //Must wait 30 days after a prize to get another prize
16 IF (WORKDATE - "Last Prize Date") < 30 THEN
17     CurrReport.SKIP;
18 //CDM.07.01 ----- end -----
```

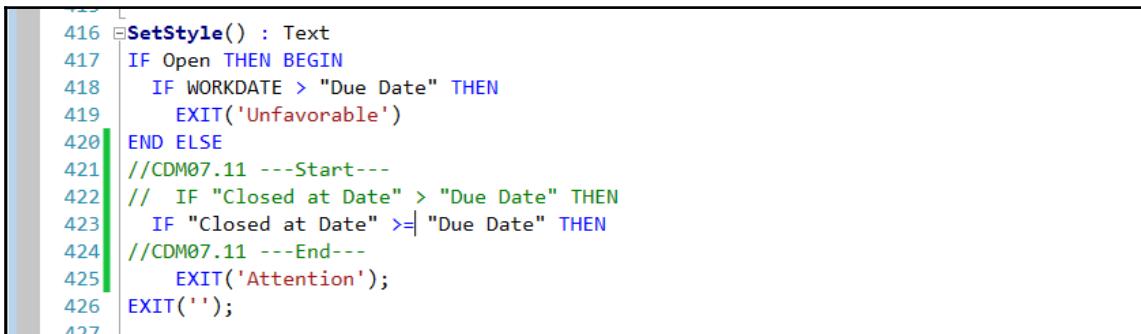
In the preceding screenshot code, we have made the modification version number **01**, resulting in a changed version code of **CDM.07.01**. In the following code, modifications are highlighted by bracketing the additional code with comment lines containing the modification identifier and **start** and **end** text indicators. Published standards do not include the dashed lines shown here, but doing something that makes comments stand out makes it easier to spot modifications when we are visually scanning code.

A 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, we should use // when our comment is relatively short. If we decide to use { }, it's a good idea to insert a // comment, at least at the beginning and end of the material inside the braces to make the comments more visible. It is also highly recommended to put each of the braces on a separate line to make them more obvious. Some experienced developers recommend using // on all removed code lines, to make the deletions easier to spot later. Evolving standards now recommend against any use of the braces for commenting out code, as the use of brackets may conflict with the upgrade merge processes supplied by Microsoft.

Consider the following lines of code as an example:

```
{//CDM.07.02 start deletion -----
//CDM.07.02 Replace validation with a call to an external function
...miscellaneous C/AL validation code
//CDM.07.02 end deletion ----- }
```

If we delete code that is part of the original Microsoft distribution, we should leave the original statements in place, but commented out so the old code is inoperative (an exception to this may apply if a source code control system that tracks all changes is in use). The same concept applies when we change the existing code; leave the original code in place commented out with the new version being inserted, as shown in the following screenshot. This approach does not necessarily apply to the code that we created originally:



The screenshot shows a portion of Microsoft Dynamics NAV source code in a text editor. The code is written in C/AL (Microsoft AL) and is part of a function named `SetStyle()`. The code checks if the `WORKDATE` is greater than the `"Due Date"`. If it is, it exits with the message `'Unfavorable'`. If it's not, it exits with `'Attention'`. There are two sections of code that are identical except for the condition: one for `"Due Date"` and another for `"Closed at Date"`. The first section is uncommented, while the second is commented out with `//`. A vertical green bar highlights the line `421 //CDM07.11 ---Start---`, indicating the start of a comment block.

```
416 SetStyle() : Text
417 IF Open THEN BEGIN
418   IF WORKDATE > "Due Date" THEN
419     EXIT('Unfavorable')
420   END ELSE
421   //CDM07.11 ---Start---
422   // IF "Closed at Date" > "Due Date" THEN
423   IF "Closed at Date" >= "Due Date" THEN
424   //CDM07.11 ---End---
425     EXIT('Attention');
426   EXIT('');
427
```

**Comment Selection** and **Uncomment Selection** options have been added to the **Edit** menu option list in NAV 2017. When we are commenting or uncommenting large chunks of code, these can be useful. See the Help **C/AL Comments**.

When we make changes such as these, we don't want to forget to update the object version numbers located in the **Version List** field on the **Object Designer** screen. It's also a good idea to take advantage of one of the previously mentioned source code management tools to track modifications.

From our previous experience, we know that the format of the internal documentation is not what's critical. What is critical is that the documentation exists, is consistent in format, and accurately describes the changes that have occurred. The internal documentation should be a complement to the external documentation that defines the original functional requirements, validation specifications, and recommended operating procedures.

Yet another approach, one that is especially suitable for modifications that exceed a small number of lines of code or that will be called from multiple places, is to create a new function for the modification. Name the function so that its purpose is obvious, and then call the function from the point of use. In this case, that function might be named something like `CheckDatePrizeLastWon`. In this case, the function would only have one line of code (not a good example), and we would pass in the `Last Prize Date` value and the function would return a Boolean value telling us whether or not the individual was eligible for a new prize.

## Source code management

Instead of documenting within the code, many developers choose to use source code management tools, such as GitHub or Visual Studio Team Services.

Tools like these allow you to create change requests and connect changed code to these requests. Once the change is validated, it is connected to the documentation, automatically creating a knowledge database outside of the source code. This keeps the source code clean and readable.

Out-of-the-box Microsoft does not support integration with any source code management tool, but we can find many third-party solutions on the Web. Some tools and additional information can be found at these websites:

- <https://www.idyn.nl/products/nav-development/object-manager-advanced>
- <https://www.stati-cal.com>
- [https://www.youtube.com/playlist?list=PLhZ3P-LY7Cqlh\\_z1rLOAAAAk9Y2clUhHl](https://www.youtube.com/playlist?list=PLhZ3P-LY7Cqlh_z1rLOAAAAk9Y2clUhHl)
- <https://markbrummel.blog/?s=github>

## Validation functions

C/AL includes a number of utility functions designed to facilitate data validation or initialization. Some of these functions are as follows:

- TESTFIELD
- FIELDERROR
- INIT
- VALIDATE

## TESTFIELD

The TESTFIELD function is widely used in standard NAV code. With TESTFIELD, we can test a variable value and generate an error message in a single statement if the test fails. The syntax is as follows:

```
Record.TESTFIELD (Field, [Value])
```

If a **Value** is specified and the field does not equal that value, the process terminates with an error condition and an error message is issued.

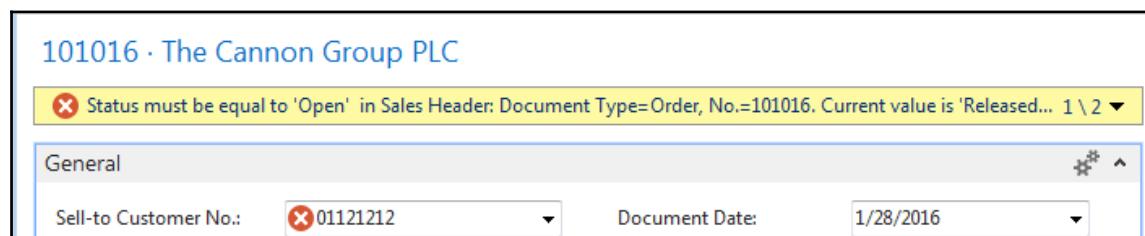
If no **Value** is specified, the field contents are checked for values of zero or blank. If the field is zero or blank, then an error message is issued.

The advantage of **TESTFIELD** is ease of use and consistency in code, and in the message displayed. The disadvantage is the error message is not as informative as a careful developer would provide.

The following screenshot of the **TESTFIELD** function usage is from Table 18 – Customer. This code checks to make sure the **Sales Order** field **Status** is equal to the option value **Open** before allowing the value of the field **Sell-to Customer No.** to be entered:

```
TESTFIELD(State, Status::Open);
```

An example of the error message generated when attempting to change the **Sell-to Customer No.** field when **Status** is not equal to the option value **Open** follows:



## FIELDERROR

Another function, which is very similar to the **TESTFIELD** function, is **FIELDERROR**. However, 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. **FIELDERROR** is designed to display an error message and then terminate the process. This approach is followed in much of the standard NAV logic, especially in the Posting Codeunits (for example, Codeunits 12, 80, 90). The syntax is as follows:

```
TableName.FIELDERROR(Fieldname [, OptionalMsgText]);
```

If we include our own message text by defining a Text Constant in the **C/AL Globals | Text Constants** tab (so the message can be multilingual), we will have the following line of code:

```
Text001      must be greater than Start Time
```

Then, we can reference the Text Constant in code as follows:

```
IF Rec."End Time" <= "Start Time" THEN  
    Rec.FIELDERROR("End Time",Text001);
```

The result is an error message from FIELDERROR, as shown in the following screenshot:

A screenshot of a Microsoft Dynamics application window titled 'Playlist Header'. At the top, there is an error message in a yellow box: 'End Time must be greater than Start Time in Playlist Header No.=222'. Below the message is a table with the following data:

No.	Radio Show No.	Description	Broadcast Date	Duration	Start Time	End Time
221	RS001	CeCe and Friends	5/22/2018	2 hours	6:00:00 AM	8:00:00 AM
222	RS003	Ask Cole!	5/22/2018	2 hours	8:00:00 AM	7:00:00 AM

An error message that simply identifies the data field, but does not reference a message text looks like the following screenshot, with the record key information displayed:

A screenshot of a Microsoft Dynamics application window titled 'Playlist Header'. At the top, there is an error message in a yellow box: 'End Time must not be 7:00:00 AM in Playlist Header No.=222'. Below the message is a table with the following data:

No.	Radio Show No.	Description	Broadcast Date	Duration	Start Time	End Time
221	RS001	CeCe and Friends	5/22/2018	2 hours	6:00:00 AM	8:00:00 AM
222	RS003	Ask Cole!	5/22/2018	2 hours	8:00:00 AM	7:00:00 AM

Because the error message begins with the name of the field, we will need to be careful that our **Text Constant** is structured to make the resulting error message easy to read.

If we don't include our own message text, the default message comes in two flavors. The first instance is the case where the referenced field is not empty. Then the error message presumes that the error is due to a wrong value, as shown in the previous screenshot. In this case, where the referenced data field is empty, the error message logic presumes the field should not be empty, as shown in the following screenshot:

The screenshot shows a Microsoft Dynamics NAV application window titled "Playlist Header". At the top, there is a yellow error bar with the text "You must specify End Time in Playlist Header No.= '221'." Below the error bar is a search bar with the placeholder "Type to filter (F3)" and a dropdown menu set to "No.". A message "No filters applied" is displayed. The main area contains a grid of records with the following columns: No., Radio Show No., Description, Broadcast Date, Duration, Start Time, and End Time. The "End Time" column for record No. 221 has a red "X" icon in the header, indicating it is required. Record No. 221 has a red "X" icon in its first column. Record No. 222 has valid values in all columns.

No.	Radio Show No.	Description	Broadcast Date	Duration	Start Time	End Time
221	RS001	CeCe and Friends	5/22/2018	2 hours	6:00:00 AM	
222	RS003	Ask Cole!	5/22/2018	2 hours	8:00:00 AM	10:00:00 AM

## INIT

The `INIT` function initializes a record in preparation for its use, typically in the course of building a record entry to insert in a table. The syntax is as follows:

```
Record.INIT;
```

All the data fields in the record are initialized as follows:

- Fields that have an `InitValue` property defined are initialized to the specified value.
- Fields that do not have a defined `InitValue` are initialized to the default value for their data type.
- Primary key fields and timestamps are not automatically initialized. If they contain values, those will remain. If new values are desired, they must be assigned in code.

## VALIDATE

The syntax of the `VALIDATE` function is as follows:

```
Record.VALIDATE (Field [, Value])
```

VALIDATE will fire the OnValidate trigger of Record.Field. If we have specified a value, it is assigned to the field and the field validations are invoked.

If we don't specify a value, then the field validations are invoked using the field value that already exists in the field. This function allows us to easily centralize our code design around the table--one of NAV's strengths.

For example, if we were to code changing Item."BaseUnitofMeasure" from one unit of measure to another, 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, because quantity equal 1 is a requirement of the **Base Unit of Measurement** field. Making the unit of measure change with a simple assignment statement would not catch a quantity value error.

The following are two forms of using VALIDATE that give the same end result:

```
Item.VALIDATE("Base Unit of Measure", 'Box');  
Item."Base Unit of Measure" := 'Box';  
Item.VALIDATE("Base Unit of Measure");
```

## Date and Time functions

NAV provides a considerable number of Date and Time functions. In the following, we will cover several of those that are more commonly used, especially in the context of accounting date-sensitive activity:

- The TODAY, TIME, and CURRENTDATETIME functions
- The WORKDATE function
- The DATE2DMY, DATE2DWY, DMY2DATE, DWY2DATE, and CALCDATE functions

## TODAY, TIME, and CURRENTDATETIME

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 (formerly referenced as **GMT** or **Greenwich Mean Time**) and then displayed in local time. If we are using the Windows client, these use the time setting in the NAV Client. If the system operates in multiple time zones at one time, search Microsoft Dynamics NAV Help on time zone for the several references on how to deal with multiple time zones.

The syntax for each of these follows:

```
DateField := TODAY;  
TimeField := TIME;  
DateTimeField := CURRENTDATETIME;
```

These are often used 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 the 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.

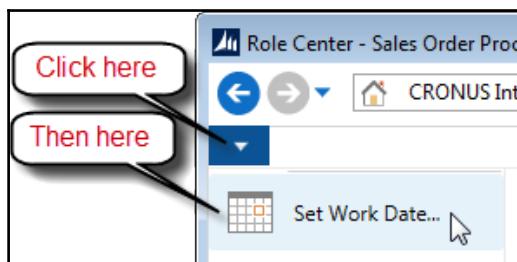
The undefined date in NAV 2017 is represented by the earliest valid DATETIME in SQL Server, which is January 1, 1753 00:00:00:000. The undefined date in NAV is represented as 0D (zero D, as in Days), with subsequent dates handled through December 31, 9999. A date outside this range will result in a runtime error.

The Microsoft Dynamics NAV **undefined time (0T)** is represented by the same value as an **undefined date (0D)** is represented.

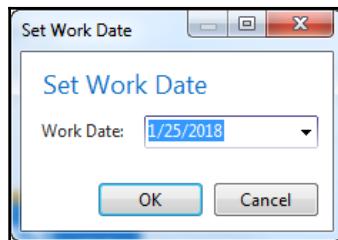
If a two-digit year is entered or stored as a date and has a value of 30 to 99, it is assumed to be in the 1900s. If the two-digit date is in the range of 00 to 29, then it is treated as a 2000s date.

## WORKDATE

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. However, at any time, the operator can set the Work Date to any date by accessing the Application Menu and clicking on **Set Work Date...**, and then entering the new Work Date:



The user can also click on the Work Date displayed in the status bar at the bottom of the RTC. The following screenshot shows the **Set Work Date** screen:



For data entry purposes, the current Work Date can be entered by the operator by simply typing the letter w or W or the word WORKDATE in the date entry field. NAV will automatically convert that entry to the current Work Date.

The syntax to access the current WorkDate value from within C/AL code is as follows:

```
DateField := WORKDATE;
```

The syntax to set the WorkDate to a new date from within C/AL code is as follows:

```
WORKDATE (newdate);
```

## DATE2DMY function

DATE2DMY allows us to extract the sections of a date (Day of the month, 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 us to choose which value (Day, Month, or Year) will be assigned to the IntegerVariable field. The following table provides the DATE2DMY extraction choices:

DATE2DMY extraction choice	Integer value result
1	two-digit day (1-31)
2	two-digit month (1-12)
3	four-digit year

## DATE2DWY function

DATE2DWY allows us to extract the sections of a date (Day of the week, Week of the year, and Year) from a Date field in exactly the same fashion as DATE2DMY. The ExtractionChoice parameter allows us to choose which value (Day, Week, or Year) will be assigned to IntegerVariable, as shown in the following table:

DATE2DWY extraction choice	Integer value result
1	two-digit day (1 - 7 for Monday - Sunday)
2	two-digit week (1 - 53)
3	four-digit year

## DMY2DATE and DWY2DATE functions

DMY2DATE allows us 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 (MonthValue or YearValue) 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 us to create a date from integer values representing the day of the week (from 1 to 7 representing Monday to Sunday), week of the year (from 1 to 53), followed by 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. By default, such a week is assigned to the year in which it has four or more days. In that case, the year of the result will vary, depending on the day of the week in the parameters (in other words, the year of the result may be one year greater than the year specified in the parameters). This is a perfect example of why thorough testing of our code is always appropriate.

## CALCDATE

CALCDATE allows us to calculate a date value assigned to a Date data type variable. The calculation is based on a Date Expression applied to a Base Date (Reference Date). If we don't specify a BaseDateValue, the current system date is used as the default date. We can specify the BaseDateValue 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 we can build a DateExpression. The rules for the CALCDATE function, DateExpression, are similar to the rules for **DateFormula**, which is described in Chapter 3, *Data Types and Fields*.

If there is a CW, CM, CP, CQ, or CY (Current Week, Current Month, Current Period, Current Quarter, or Current Year) parameter in an expression, then the result will be evaluated based on the BaseDateValue. If we have more than one of these in our expression, the results are unpredictable. Any such expression should be thoroughly tested before releasing to users.

If our 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. Also, if we create our own Date Expression in the form of a string constant within our inline C/AL code, surrounding the constant with <> delimiters as part of the string, it will make the constant language independent. Otherwise, the Date Expression constant will be language dependent.

Regardless of how we have constructed our 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 it is by using a Report whose sole task is to evaluate our expression and display the result. If we want to try different Base Dates, we can use the Request Page, accept the BaseDate as input, then calculate and display the DateVariable in the OnValidate trigger.

Some sample CALCDATE expression evaluations are as follows:

- ('<CM>', 031017D) will yield 03/31/2017, that is, the last day of the Current Month for the date 3/10/2017
- ('<-WD2>', 031216D) will yield 03/08/2016, that is, the WeekDay #2 (the prior Tuesday) before the date 3/12/2016
- ('<CM+1D>', BaseDate) where BaseDate equals 03/10/17, will yield 04/01/2017, 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)

## Data conversion and formatting functions

Some data type conversions are handled in the normal process flow by NAV without any particular attention on part of the Developer, such as Code to Text and Char to Text. Some data type conversions can only be handled through C/AL functions. Formatting is included because it can also include a data type conversion. Rounding does not do a data type conversion, but it does result in a change in format (the number of decimal places). Let's review the following functions:

- The ROUND function
- The FORMAT function
- The EVALUATE function

## ROUND function

The ROUND function allows us to control the rounding precision for a decimal expression. The syntax for the ROUND function is as follows:

```
DecimalResult := ROUND (Number [, Precision] [, Direction] )
```

Here, Number is 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. Some examples of Precision values follow:

Precision value	Rounding effect
100	To a multiple of 100
1	To an integer format
01	To two decimal places (the US default)
0.01	Same as .01
.0001	To four decimal places

If no Precision value is specified, the Rounding default is controlled by a value set in **General Ledger Setup** in the **Appln. Rounding Precision** field on the **Application** tab. If no value is specified, Rounding will default to two decimal places. If the precision value is, for example, .04 rather than .01, the rounding will be done to multiples of four at the number of decimal places specified.

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 and the default)
>	Round up
<	Round down

Consider the following statement:

```
DecimalValue := ROUND (1234.56789, 0.001, '<')
```

It would result in a DecimalValue containing 1234.567, whereas the following statements would both result in a DecimalValue containing 1234.568:

```
DecimalValue := ROUND (1234.56789, 0.001, '=')  
DecimalValue := ROUND (1234.56789, 0.001, '>')
```

## 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 *Developer and IT Pro Help* file for the FORMAT function and FORMAT property. Whenever possible, we should always apply FORMAT in its simplest 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. We should make sure that we include the extremes of the range of possible values in our testing.

The optional `OutputLength` parameter can be zero (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 value designed to control the maximum character length and padding of the formatted string result.

The last optional parameter has two mutually exclusive sets of choices. One set, represented by an integer `FormatNumber`, allows the choice of a particular predefined (standard) format, of which there are four to nine choices depending on the `ExpressionToFormat` data type. The format parameter of the number 9 is used for XMLport data exporting. Use of the optional number 9 parameter will convert C/SIDE format data types into XML standard data types. The other set of choices allows us to build our own format expression.

The *Developer and IT Pro Help* information for the FORMAT property provides a relatively complete description of the available tools from which we can build our 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 formats and formatted data results.

Even though the FORMAT function is non-executable, an erroneous FORMAT function will result in a runtime error that will terminate the execution of the process. Thus, to avoid production crashes, it is very important that we thoroughly test any code where FORMAT is used.

## EVALUATE function

The EVALUATE function is essentially the reverse of the FORMAT function, allowing conversion of a string value into the defined data type. The syntax of the EVALUATE function is as follows:

```
[ BooleanVariable := ] EVALUATE ( ResultVariable,  
StringToBeConverted [, 9]
```

The handling of a runtime error can be done by specifying BooleanVariable or including EVALUATE in an expression to deal with an error, such as an IF statement. The ResultVariable data type will determine what data conversion the EVALUATE function will attempt. The format of the data in StringToBeConverted must be compatible with the data type of ResultVariable, otherwise, a runtime error will occur.

The optional parameter, number 9, is only used for XMLport data exporting. Use of the optional number 9 parameter will convert C/SIDE format data types into XML standard data types. This deals 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 for an EVALUATE result can include decimal, Boolean, datetime, date, time, integer, and duration.

## FlowField and SumIndexField functions

In Chapter 3, *Data Types and Fields*, we discussed SumIndexFields and FlowFields in the context of table, field, and key definition. To recap briefly, SumIndexFields are defined in the screen where the table keys are defined. They allow very rapid calculation of values in filtered data. In most ERP and accounting software systems, the calculation of group totals, periodic totals, and such require time-consuming processing of all the data to be totaled.

SIFT allows a NAV system to respond almost instantly with totals in any area where the SumIndexField was defined and is maintained. 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 almost instantly! SumIndexFields are the basis of FlowFields that have a Method of Sum or Average; such a FlowField must refer to a data element that is defined as a SumIndexField.

When we access a record that has a `SumIndexField` defined, there is no visible evidence of the data sum that `SumIndexField` represents. When we 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 or report, it is automatically calculated by NAV; the developer doesn't need to do so. However, in any other scenario, the developer is responsible for calculating `FlowFields` using the `CALCFIELDS` function.

`FlowFields` are one of the key areas where NAV systems are subject to significant processing bottlenecks. Even with the improved NAV 2017 design, it is still critical that the Table 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 existing keys or add new keys to improve `FlowField` performance.



Note that even though we can manage indexes in SQL Server independent of the NAV key definitions, having two different definitions of keys for a single table may make our system more difficult to support in the long run because the SQL Server resident changes aren't always readily visible to the NAV developer.

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.



Including `SumIndexFields` in a List page display is almost always a bad idea, because each `SumIndexField` instance must be calculated as it is displayed. Applicable functions include `CALCFIELDS`, `CALCSUMS` and `SETAUTOCALCFIELDS`.

## 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. Specification of the `BooleanField` allows us to handle any runtime 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 (we need to be careful not to overlook this). 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.

When the following conditions are true, CALCFIELDS uses dynamically maintained SIFT data:

- The NAV key contains the fields used in the filters defined for the FlowField
- The SumIndexFields property on the operative key contain the fields provided as parameters for calculation
- The MaintainSIFTIndex property on the key is set to Yes; this is the default setting

If all these conditions are not true and CALCFIELDS is invoked, we will not get a runtime error as in previous NAV versions, but SQL Server will calculate the requested totals the hard way--by reading all the necessary records. This could be very slow and inefficient, and should not be used for frequently processed routines or large datasets. On the other hand, if the table does not contain a lot of data or the SIFT data will not be used very often, it may be better to have the MaintainSIFTIndex property set to No.

## SETAUTOCALCFIELDS function

The syntax for SETAUTOCALCFIELDS is as follows:

```
[BooleanField := ] Record.SETAUTOCALCFIELDS  
( FlowField1 [, FlowField2] [, FlowField3]...)
```

When SETAUTOCALCFIELDS for a table is inserted in code in front of record retrieval, the specified FlowFields are automatically calculated as the record is read. This is more efficient than performing CALCFIELDS on the FlowFields after the record has been read.

If we want to end the automatic FlowField calculation on a record, call the function without any parameters: [BooleanField := ] Record.SETAUTOCALCFIELDS () .

Automatic FlowField calculation equivalent to SETAUTOCALCFIELDS is automatically set on for the system record variables, Rec and xRec.

## CALCSUMS function

The CALCSUMS function is conceptually similar to CALCFIELDS for the calculation of Sums only. However, CALCFIELDS operates on FlowFields and CALCSUMS operates directly on the record where the SumIndexFields are defined for the keys. That difference means that we 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 the FlowFields).

The syntax for CALCSUMS is as follows:

```
[ BooleanField := ] Record.CALCSUMS ( SumIndexField1 [,SumIndexField2]  
, ...)
```

Prior to a statement of this type, we should specify a key that has the SumIndexFields defined (in order to maximize the probability of good performance). Before executing the CALCSUMS function, we also need to specify any filters that we want to apply to the Record from which the sums are to be calculated. The SumIndexFields calculations take into account the filters that are currently applied to the Record.

Executing the CALCSUMS function will cause the specified SumIndexFields totals to be calculated. Specification of the BooleanField allows us 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. If possible, CALCSUMS uses the defined SIFT. Otherwise, SQL Server creates a temporary SIFT on the fly.

Before the execution of CALCSUMS, SumIndexFields contain only the data from the individual record that was read. After the CALCSUMS execution, the included SumIndexFields contain the totals that were calculated by the CALCSUMS function (these totals are only in memory, not in the database). These totals can then be used the same as data in any field; however, if we want to access the individual record's original data for that field, we must either save a copy of the record before executing the CALCSUMS or we must reread the record. The CALCSUMS function 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 "AmountIncludingVAT". These FlowFields are all based on Sums of entries in the SalesLine table. The CalcFormula for Amount is  
Sum("SalesLine".Amount WHERE (DocumentType=FIELD(DocumentType), DocumentNo.=FIELD(No.))).



CALCSUMS can be used on any integer, big integer, or decimal field with any filter on any table, but for larger datasets creating a key with a SumIndexField is recommended.

To calculate a TotalOrderAmount value while referencing the Sales Header table, the code can be as simple as this:

```
"Sales Header".CALCFIELDS (Amount);  
TotalOrderAmount := "Sales Header".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.");  
"Sales Line".CALCSUMS (Amount);  
TotalOrderAmount := "Sales Line".Amount;
```

## Flow control

Process flow control functions are the functions that execute the decision making and resultant logic branches in executable code. IF-THEN-ELSE, discussed in Chapter 6, *Introduction to C/SIDE and C/AL*, is also a member of this class of functions. Here, we will discuss the following:

- REPEAT – UNTIL
- WHILE – DO
- FOR-TO and FOR-DOWNTO
- CASE – ELSE
- WITH – DO
- QUIT, BREAK, EXIT, and SKIP

## REPEAT-UNTIL

REPEAT-UNTIL allows us to create a repetitive code loop, which REPEATS a block of code UNTIL a specific conditional expression evaluates to TRUE. In that sense, REPEAT-UNTIL defines a block of code, operating somewhat like the BEGIN-END compound statement structure that we covered in Chapter 6, *Introduction to C/SIDE and C/AL*. 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 often part of the data input cycle along with the FIND-NEXT structure, which will be covered shortly.

Here is an example of the REPEAT-UNTIL structure to process and sum data in the 10-element array `CustSales`:

```
LoopCount := 0;  
REPEAT  
    LoopCount := LoopCount + 1;  
    TotCustSales := TotCustSales + CustSales[LoopCount];  
UNTIL LoopCount = 10;
```

## WHILE-DO

A WHILE-DO control structure allows us to create a repetitive code loop that will DO (execute) a block of code WHILE a specific conditional expression evaluates to TRUE. WHILE-DO is different from REPEAT-UNTIL, both because it may need a BEGIN-END structure to define the block of code to be executed repetitively (REPEAT-UNTIL does not), and it has different timings for the evaluation of the exit condition.

The syntax of the WHILE-DO control structure is as follows:

```
WHILE <Condition> DO <Statement>
```

The Condition value can be any Boolean expression that evaluates to TRUE or FALSE. The Statement value can be simple or the most complex compound BEGIN-END statement. Most WHILE-DO loops will be based on 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. Thus, a WHILE-DO loop can be exited without processing.

A WHILE-DO structure to process data in the 10-element array CustSales follows:

```
LoopCount := 0;  
WHILE LoopCount < 10  
DO BEGIN  
    LoopCount := LoopCount + 1;  
    TotCustSales := TotCustSales + CustSales[LoopCount];  
END;
```

In NAV, REPEAT-UNTIL is much more frequently used than WHILE-DO.

## FOR-TO or FOR-DOWNT0

The syntax for a FOR-TO and FOR-DOWNT0 control statements follow:

```
FOR <Control Variable> := <Start Number> TO <End Number> DO <Statement>  
FOR <Control Variable> := <Start Number> DOWNT0 <End Number> DO <Statement>
```

A FOR control structure is used when we wish to execute a block of code a specific number of times.

The Control Variable value is an Integer variable. Start Number is the beginning count for the FOR loop and End Number is the final count for the loop. If we wrote the statement FOR LoopCount := 5 TO 7 DO [block of code], then [block of code] would be executed three times.

FOR-TO increments the Control Variable. FOR-DOWNT0 decrements the Control Variable.

We must be careful not to manipulate the Control Variable value in the middle of our loop. Doing so will likely yield unpredictable results.

## CASE-ELSE statement

The CASE-ELSE statement is a conditional expression, which is very similar to IF-THEN-ELSE, except that it allows 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 able to be automatically converted to the data type of the ExpressionToBeEvaluated. Each ValueSet 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';
    '16'...'20': Customer."Territory Code" := 'WEST';
    'N': Customer."Territory Code" := 'NORTH';
    '27'...'38': Customer."Territory Code" := 'SOUTH';
    ELSE Customer."Territory Code" := 'FOREIGN';
END;
```

In the preceding code example, we see several alternatives for the Value Set. The first line (EAST) Value Set contains a list of values. If "SalespersonCode" is equal to '2', '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 '16' through '20'. The third line, (NORTH) ValueSet, is just a single value ('N'). If we look through the standard NAV code, we will see that a single value is the most frequently used CASE structure in NAV. In the fourth line of our example (SOUTH), the Value Set is again a range ('27'..'38'). If nothing in any Value Set matches ExpressionToBeEvaluated, the ELSE clause will be executed which sets Customer."Territory Code" equal to 'FOREIGN'.

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 ['16'..'20'] THEN
    Customer."Territory Code" := 'WEST'
ELSE IF Customer."Salesperson Code" = 'N' THEN
    Customer."Territory Code" := 'NORTH'
ELSE IF Customer."Salesperson Code" IN ['27'..'38'] THEN
    Customer."Territory Code" := 'SOUTH'
ELSE Customer."Territory Code" := 'FOREIGN';
```

The following is a slightly 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 the same as in 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 we are writing code referring to fields within a record, the most specific syntax for field references is the fully qualified reference [ RecordName.FieldName ]. 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 because the compiler assumes a default record qualifier based on code context. This happens automatically for variables within a page 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 in that object. In Report and XMLport objects, the Data Item record is the implicit record qualifier for the fields referenced within the triggers of that Data Item, such as `OnAfterGetRecord` and `OnAfterImportRecord`.

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 follows:

```
WITH <RecordQualifier> DO <Statement>
```

Typically, the DO portion of this statement will be followed by a BEGIN-END code block allowing a compound statement. The scope of the WITH-DO statement is terminated by the end of the DO statement.

When we execute a WITH-DO statement, RecordQualifier becomes the implicit record qualifier used by the compiler until the end of that statement or until that qualifier is overridden by a nested WITH-DO statement. A 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, take a look at the following lines of code:

```
WITH Customer DO
BEGIN
    Address := '189 Maple Avenue';
    City := 'Chicago';
END;
```

Best practice says that the WITH-DO statements should only be used in functions within a Codeunit or a Report.

The WITH-DO statements nested one within another are legal code, but are not used in standard NAV. 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, Report, or XMLport.

Of course, wherever the references to record variables other than the implicit one occur within the scope of WITH-DO, we 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 a maintenance developer who works on this code later.

## QUIT, BREAK, EXIT, and SKIP

This group of C/AL functions also controls process flow. Each acts to interrupt flow in different places and with different results. To get a full appreciation for how these functions are used, we should review them in place in NAV 2017.

### QUIT function

The QUIT function is the ultimate processing interrupt for Report or XMLport objects. When QUIT is executed, processing immediately terminates, even for the OnPostObject triggers. No database changes are committed. 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 follows:

```
CurrReport.QUIT;  
CurrXMLport.QUIT;
```

### BREAK function

The BREAK function terminates the DataItem in which it occurs. BREAK can only be used in Data Item triggers in Reports and XMLports. It can be used to terminate the sequence of processing one DataItem segment of a report while allowing subsequent DataItem processing to continue.

The BREAK syntax follows:

```
CurrReport.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. It can be used to end the processing of the trigger or to pass a return value from a local function. A return value cannot be used for system-defined triggers or local functions that don't have a return value defined. If EXIT is used without a return value, a default return value of zero is returned. The syntax for EXIT follows:

```
EXIT( [<ReturnValue>] )
```

## SKIP function

When executed, the SKIP function will skip the remainder of the processing in the current record cycle of the current trigger. Unlike BREAK, it does not terminate the DataItem processing completely. It can be used only in the OnAfterGetRecord trigger of a Report or XMLport object. In 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 that single iteration of the trigger without interfering with any subsequent processing.

The SKIP syntax is one of the following:

```
CurrReport.SKIP;  
CurrXMLport.SKIP;
```

## Input and Output functions

In the previous chapter, you learned about the basics of the FIND function. You learned about FIND ('-') to read from the beginning of a selected set of records, FINDSET to read a selected set of records, and FIND ('+') to begin reading at the far end of the selected set of records. Now, we will review additional functions that are generally used with FIND functions in typical production code. While we are designing code that uses the MODIFY and DELETE record functions, we need to consider possible interactions with other users on the system. There might be someone else modifying and deleting records in the same table that our application is updating.

We may want to utilize the `LOCKTABLE` function to gain total control of the data briefly while updating it. We can find more information on `LOCKTABLE` in the online *C/AL Reference Guide Help*. The SQL Server database supports Record Level Locking. There are a number of factors that we should consider when coding data locking in our 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 follows:

```
IntegerValue := Record.NEXT ( ReadStepSize )
```

The full assignment statement format is rarely used to set an `IntegerValue`. In addition, there is no documentation for the usage of a non-zero `IntegerValue`. When `IntegerValue` goes to zero, it means a NEXT record was not found. In early versions of NAV 2017, the Help text for `NEXT` does not properly explain the use or value setting for `IntegerValue`.

If the `ReadStepSize` value is negative, the table will be read in reverse; if `ReadStepSize` 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 1, in which case, every record will be read and the read direction will be forward.

In a typical data-read loop, the first read is a `FIND` or `FINDSET` function, followed by a `REPEAT-UNTIL` loop. The exit condition is the expression, `UNTIL Record.NEXT=0;`. The C/AL for `FINDSET` and `FIND ('-')` are structured alike.

The full C/AL syntax for this typical loop looks 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 add new records to a 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 the insertion would result in a duplicate Primary Key), the process will terminate with an error. Generally, we should handle a detected error in code using the `BooleanValue` and supplying our own error-handling logic, rather than allow a default termination.

The `TriggerControlBoolean` value controls whether or not the table's `OnInsert` trigger fires when the `INSERT` occurs. The default value is `FALSE`. If we let the default `FALSE` control, we run the risk of not performing error checking that the table's designer assumed would be run when a new record was added.



When we are reading a table, and we also need to `INSERT` records into that same table, the `INSERT` should be done to a separate instance of the table. We can use either a global or local variable for that second instance. If we `INSERT` into the same table we are reading, we run the risk of reading the new records as part of our processing (likely a very confusing action). We also run the risk of changing the sequence of our processing unexpectedly due to the introduction of new records into our dataset. While the database access methods are continually improved by Microsoft, and this warning may be overcautious, it is better to be safe than sorry.

## MODIFY function

The purpose of the `MODIFY` function is to modify (update) 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. The code should either handle a detected error or gracefully terminate the process. The `TriggerControlBoolean` value controls whether or not the table's `OnModify` trigger fires when this `MODIFY` occurs. The default value is `FALSE`, which would not perform any `OnModify` processing. `MODIFY` cannot be used to cause a change in a Primary Key field. In that case, the `RENAME` function must be used.

There is system-based checking to make sure a `MODIFY` is done using the current version of the data record by making sure another process hasn't modified and committed the record after it was read by this process. Our logic should refresh the record using the `GET` function, then change any values, and then call the `MODIFY` function.

## Rec and xRec

In the `Table` and `Page` objects, the system automatically provides us with the system variables, `Rec` and `xRec`. Until 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`, we 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 `DELETE` fails and the `BooleanValue` option is not used, the process will terminate with an error statement. Our code should handle any detected error or terminate the process gracefully, as appropriate.

The `TriggerControlBoolean` value is `TRUE` or `FALSE`, and it controls whether or not the table's `OnDelete` trigger fires when this `DELETE` occurs. The default value is `FALSE`. If we let the default `FALSE` prevail, we run the risk of not performing error checking that the table's designer assumed would be run when a record was deleted.

In NAV 2017, there is improved checking to make sure a `DELETE` is using the current version of the record and to make sure another process hasn't modified and committed the record after it was read by this process. It is still good practice for the program to refresh the record (using the `GET` function) before the `DELETE` function is called.

## MODIFYALL function

MODIFYALL is the high-volume version of the MODIFY function. If we have a group of records in which we wish to modify one field in all of these records to the same new value, we should use MODIFYALL. It is controlled by the filters that are applied at the time of invoking. The other choice for doing a mass modification would be to have a FIND-NEXT loop in which we modified each record one at a time. The advantage of MODIFYALL is that it allows the developer and the system to optimize code for the volume update. Any system optimization will be a function of what SQL statements are generated by the C/AL compiler.

The syntax for MODIFYALL is as follows:

```
Record.MODIFYALL (FieldToBeModified, NewValue  
[,TriggerControlBoolean ] )
```

The TriggerControlBoolean value, a TRUE or FALSE entry, controls whether or not the table's OnModify trigger fires when this MODIFY occurs. The default value is FALSE, which would result in the field OnValidate trigger not being executed. 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 will reassign all the TerritoryCodes for a particular Salesperson to NORTH is as follows:

```
Customer.RESET ;  
Customer.SETRANGE("Salesperson Code", 'DAS') ;  
Customer.MODIFYALL("Territory Code", 'NORTH', TRUE) ;
```

## DELETEALL function

DELETEALL is the high-volume version of the DELETE function. If we have a group of records that we wish to delete, use DELETEALL. The other choice would be a FIND-NEXT loop, in which we delete each record one at a time. The advantage of the DELETEALL function is that it allows the developer and the system to optimize code for the volume deletion. Any system optimization will be a function of what SQL statements are generated by the C/AL compiler.

The syntax for DELETEALL is as follows:

```
Record.DELETEALL ( [,TriggerControlBoolean] )
```

The TriggerControlBoolean value, a TRUE or FALSE entry, 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 little to 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 are set and does not do any referential integrity error checking.

## Filtering

Few other systems have filtering implemented as comprehensively as NAV, nor do they have it tied so neatly to the detailed retention of historical data. The result of NAV's features is that even the most basic implementation of NAV includes very powerful data analysis capabilities available to the end user.

As the developers, we should appreciate the fact that we cannot anticipate every need of any user, let alone anticipate all the needs of all users. We know we should give the users as much freedom as possible to allow them to selectively extract and review data from their system. Wherever feasible, users 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 experienced application software designers and developers, presumably we have considerable insight into good ways to analyze and present the data. On that basis, it may be appropriate for us to provide some predefined selections. In some cases, constraints of the data structure allow only a limited set of options to make sense. In such a case, we should provide specific accesses to data (through pages and/or reports). However, we should allow more sophisticated users to access and manipulate the data flexibly on their own.

When applying filters 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 very high in the currently active key (in other words, near the beginning of the key field sequence) may result in poor (or even very poor) response time for the users. In the same context, in a system suffering from a poor response time during processing, we should first investigate the relationships of active keys to applied filters, as well as how the keys are maintained. This may require SQL Server expertise in addition to NAV 2017 expertise.

Both the SETCURRENTKEY and SETRANGE functions are important in the context of data filtering. These were reviewed in Chapter 6, *Introduction to C/SIDE and C/AL*, so we won't review them again here.

## SETFILTER function

SETFILTER allows us to define and apply any filter expression that could be created manually, including various combinations of ranges, C/AL operators, and even wild cards. The SETFILTER syntax is as follows:

```
Record.SETFILTER ( Field, FilterString [, FilterValue1], . . . ] );
```

SETFILTER also can be applied to Query objects with similar syntax:

```
Query.SETFILTER ( ColumnName, FilterString  
[ , FilterValue1], . . . ] );
```

FilterString can be a literal, such as 1000..20000 or A\*|B\*|C\*, but this is not good practice. Optionally (and preferably), we 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 us to create filters whose data values can be defined dynamically at runtime. A new SETFILTER replaces any previous filtering in the same filtergroup (this will be discussed in more detail shortly) on that field or column prior to setting the new filter.

A pair of SETFILTER examples are as follows:

```
Customer.SETFILTER("Salesperson Code", 'KKS' | 'RAM' | 'CDS');  
Customer.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 a degree of flexibility which is not provided by the first option because in the second option, the variables could be assigned other values.

## 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 are as follows:

```
FromRecord.COPYFILTER(FromField, ToRecord.ToField)
```

The `From` and `To` fields must be of the same data type. The `From` and `To` tables do not have to be the same format:

```
ToRecord.COPYFILTERS(FromRecord)
```

Note that the `COPYFILTER` field-based function begins with the `FromRecord` variable, while that of `COPYFILTERS` record-based function begins with the `ToRecord` variable. `ToRecord` and `FromRecord` must be different instances of the same table.

## GETFILTER and GETFILTERS functions

These functions allow us 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
```

Similar functions exist for Query Objects. Those syntaxes are as follows:

```
ResultString := FilteredQuery.GETFILTER(filteredColumn)  
ResultString := FilteredQuery.GETFILTERS
```

The text contents of `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.**

## FILTERGROUP function

The FILTERGROUP function can change or retrieve the **Filtergroup** that is applied to a table. A filtergroup contains a set of filters that were previously applied to the table by the SETFILTER or SETRANGE functions, or as table properties defined in an object. The FILTERGROUP syntax is as follows:

```
[CurrentGroupInteger] := Record.FILTERGROUP ([NewGroupInteger])
```

Using just the Record.FILTERGROUP ([NewFilterGroupInteger]) portion sets the active Filter Group.

Filtergroups can also be used to filter Query Data Items. All the currently defined filtergroups are active and apply in combination (in other words, they are logically ANDed, resulting in a logical intersection of the sets). The only way to eliminate the effect of a filtergroup is to remove the filters in a group.

The default filtergroup for NAV is 0 (zero). Users have access to the filters in this filtergroup. Other filtergroups, numbered up through 6, have assigned NAV uses. We should not redefine the use of any of these filtergroups, but use higher numbers for any custom filtergroups in our code.

See the *Developer and IT Pro Help* section for FILTERGROUP function and Understanding Query Filters for more information.

One use of a Filter Group is to assign a filter that the user cannot see is operative and cannot change. Our code could change the Filter Group, set a special filter, and then return the active Filter Group to its original state. The following lines of code are an example:

```
Rec.FILTERGROUP (42);  
Rec.SETFILTER(Customer."Salesperson Code",MySalespersonID);  
Rec.FILTERGROUP (0);
```

This could be used to apply special application-specific permissions to a particular system function, such as filtering out access to customers by salesperson so that each salesperson can only examine data for their own customers.

## 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 (:=) are present, the MARK function will give us 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 value for SetMarkBoolean is FALSE. The MARK functions should be used carefully, and only when a simpler solution is not readily available. MARKing records can cause significant performance problems on large datasets.

## 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 TRUE or FALSE value to tell us 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. The default value for SeeMarkedRecordsOnlyBoolean is FALSE.

Although it may not seem logical, there is no option to see only the unmarked records.

For additional information, refer to the blog entry at <https://markbrummel.wordpress.com/2014/03/07/tip-36-using-mark-and-markedonly-in-the-role-tailored-client/>.

## RESET function

This function allows us to RESET, that is, clear, all filters that are currently applied to a record. 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. Filters in FILTERGROUP 1 are not reset. The syntax is `FilteredRecord.RESET;`.

## InterObject communication

There are several ways for communicating between objects during NAV processing. We will review some of the more commonly used ways.

## Communication through 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) uses Codeunit 396, NoSeriesManagement, to access 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 are defined when the function is developed. 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 developers, when we design the function, we 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, we need to create an object, which in turn calls other objects. We 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. Our user-interface object will be responsible for invoking the subordinate objects after having communicated setup and filter parameters.

There is an important 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 we will need to build our own data passing function.

In order to properly manage these relatively complex processes, we need to be familiar with the various versions of RUN and RUNMODAL functions. We 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 or report object from within another object through 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 reinitialized each time the object is called. The object can be reinitialized using CLEAR (Object).
- RUNMODAL does not allow any other object to be active in the same user session while it is running; whereas RUN allows another object instance to run in parallel with the object instance initiated by RUN.

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 *Developer and IT Pro Help* section relative to this topic. There is also Pattern documentation on this topic defined at

<https://community.dynamics.com/nav/w/designpatterns/108.posting-routine-select-behaviour>.

# Enhancing the WDTU application

Now that we have some new tools with which to work, let's enhance our WDTU application. This time, our goal is to implement functionality to allow the Program Manager to plan the Playlist schedules for Radio Shows. The process, from the user's point of view, will essentially be as follows:

1. Call up Playlist document page that displays Header, Details, and Factbox workspaces.
2. Enter Playlist Header using the Radio Show table data.
3. Enter Playlist Lines using the Resource table DJ data; the Radio Show table data for News, Weather, or Sports shows; and the Item table data for Music, PSAs, and Advertisements.
4. The Factbox will display the Required program-element fields from Radio Show/Playlist Header. These will include News (Yes or No), Weather (Yes or No), Sports (Yes or No), and Number of required PSAs and Advertisements.
5. The Factbox will also track each of the five possible required elements.

Since this development effort is an exercise to learn more about developing NAV applications, we have some specific NAV C/AL components we want to use, so we can learn more about them. Among those are the following:

- Create a CASE statement as well as a multipart IF statement for contrast
- Add code to the OnValidate trigger of fields in a table
- Implement a lookup into a related table to access needed data
- Cause FlowFields to be processed for display
- Implement a FactBox to display Radio Show requirements for News, Sports, Weather, PSAs, and Ads
- Create a new function, passing a parameter in and getting results passed back

As with any application enhancement, there will be a number of auxiliary tasks we'll have to accomplish to get the job done. These include adding some new fields to one or more tables. Not surprisingly, adding new data fields often leads to adding the new fields to one or more pages for maintenance or display. We'll have to create some test data in order to test our modifications. It's not unusual in the course of an enhancement to also find that other changes are needed to support the new functionality.

## Modify table fields

Because we want the NAV tables to be the core of the design and to host as much of the processing as makes sense, we will start our enhancement work with table modifications.

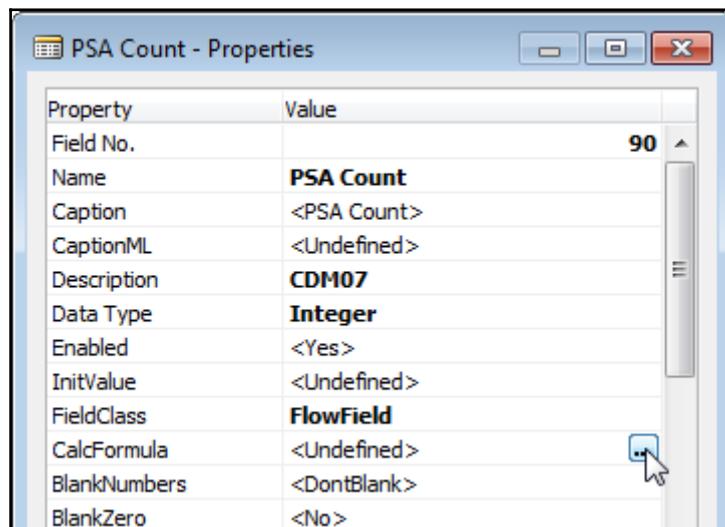
The first table modification is to add the data fields to the Playlist Header, shown in the following screenshot, to support the definition and tracking of various program segment requirements. In the Radio Show table, each show has requirements defined for a specific number of PSAs and Advertisements, and for the presence of News, Sports, and Weather spots. The Playlist Header needs this requirements information stored along with the associated TSA and Ad counts for this show instance. We will obtain the News, Sports, and Weather line counts by means of a function call:

The screenshot shows the 'Table 50002 Playlist Header - Table Designer' window. A new row of fields has been added to the table, highlighted with a blue oval. These fields represent requirements for PSAs, Ads, and Media types. The table has columns for Field No., Field Name, Data Type, and Length/Description.

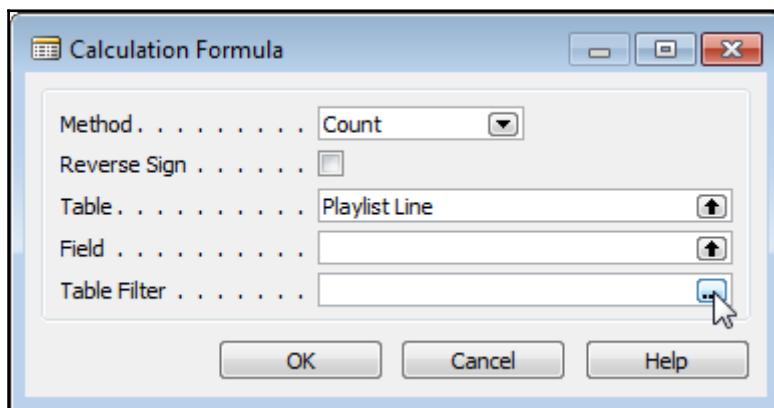
E..	Field No.	Field Name	Data Type	Length	Description
✓	10	No.	Code	20	
✓	20	Radio Show No.	Code	20	
✓	30	Description	Text	30	
✓	40	Broadcast Date	Date		
✓	50	Duration	Duration		
✓	60	Start Time	Time		
✓	70	End Time	Time		
✓	80	PSAs Required	Integer	CDM.07	
✓	90	PSA Count	Integer	CDM.07	
✓	100	Ads Required	Integer	CDM.07	
✓	110	Ad Count	Integer	CDM.07	
✓	120	News Required	Boolean	CDM.07	
✓	130	Sports Required	Boolean	CDM.07	
✓	140	Weather Required	Boolean	CDM.07	

Help

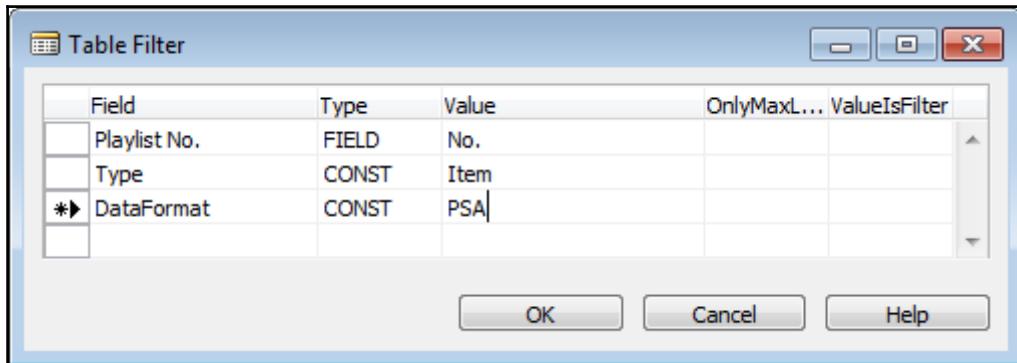
Because the Playlist Line includes an **Option** field that identifies the PSA and Advertisement records, we will use a **FlowField** to calculate the counts for each of those line types. We will construct the **FlowField** definition for the PSA Count field, starting in the **Properties** form of the field, as shown in the following screenshot:



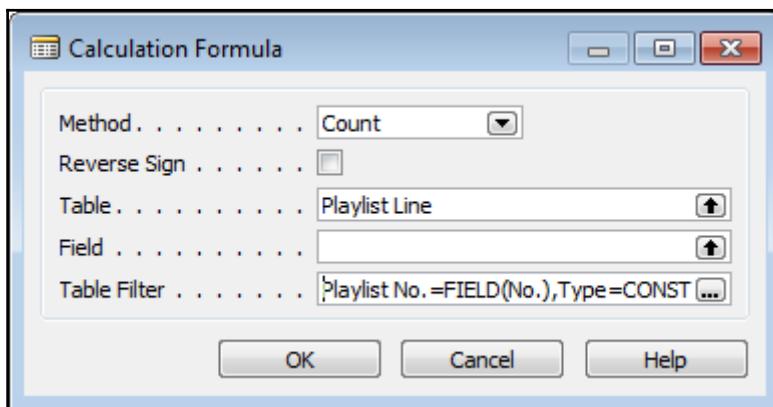
When we click on the **CalcFormula** line ellipsis, the **Calculation Formula** screen appears as shown in the following screenshot:



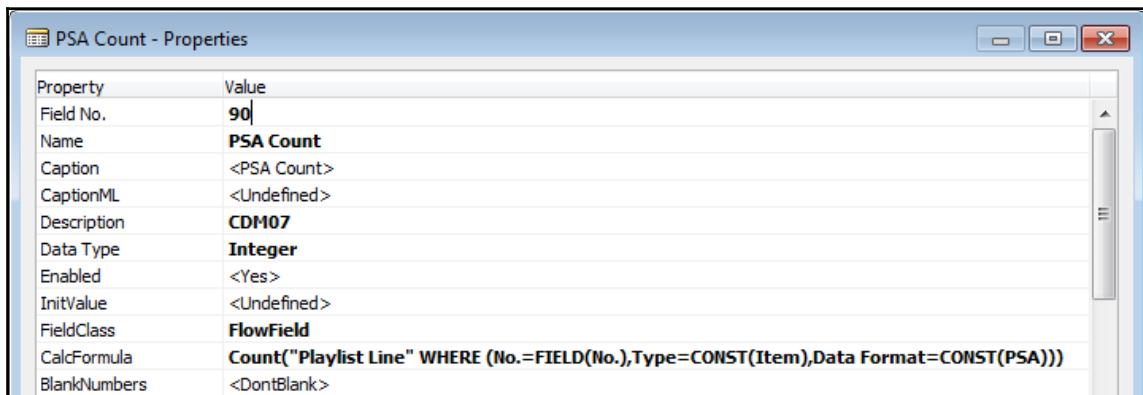
In this case, the **Method** we want to use is Count. Then, when we click on the **Table Filter** ellipsis, we will have the opportunity to enter the components defining the filters that should be applied to the Playlist Line to isolate the records that we want to count:



When we complete the **Table Filter** definition and click on the **OK** button, we will return to the **Calculation Formula** screen with the **Table Filter** field filled in (if we just click on *Esc*, the data we entered will not be saved):

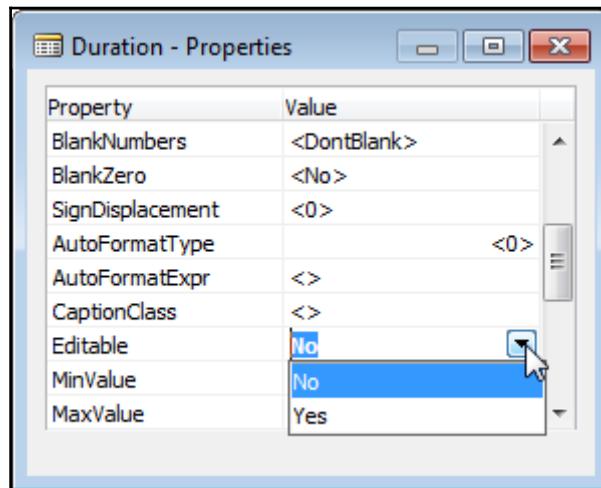


Again, click on the **OK** button to return to the **Properties** screen:



Go through the same sequence for the Ad Count field.

The only additional change we want to make to the Playlist Line table is to ensure the **Duration** field is not editable. We do this so **Start Time** and **End Time** entries define Duration rather than the other way around. Making the **Duration** field non-editable is done by simply setting the field's **Editable** property to **No**:



## Adding validation logic

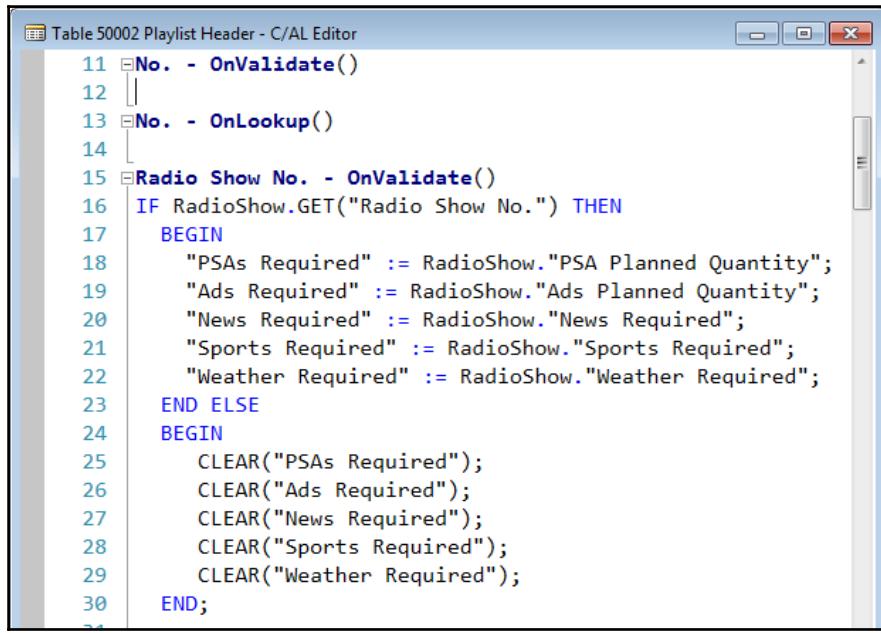
We need validation logic for both our Playlist tables, Header and Line.

### Playlist Header validations

The **Playlist Header** data fields are as follows:

- **No.:** This is the ID number for this instance of a radio show; its contents are user defined
- **Radio Show No.:** This is selected from the `Radio Show` table
- **Description:** This is displayed by means of a FlowField from the `Radio Show` table
- **Broadcast Date:** This is the show's scheduled broadcast date; it also serves as the Posting Date for any data analysis filtering
- **Start Time:** This show's scheduled broadcast start time
- **End Time:** This show's scheduled broadcast end time
- **Duration:** This show's broadcast length, displayed by means of a FlowField from the `Radio Show` table
- **PSAs Required and Ads Required:** These show if PSAs and Ads are required for broadcast during the show; they are copied from the `Radio Show` table, but are editable by the user
- **News Required, Sports Required, and Weather Required:** This checks whether or not each of these program segments are required during the show; they are copied from the `Radio Show` table, but are editable by the user

When the user chooses the Radio Show to be scheduled, we want the five different feature requirements fields in the Playlist Header to be filled in by C/AL logic, as shown in the following screenshot:



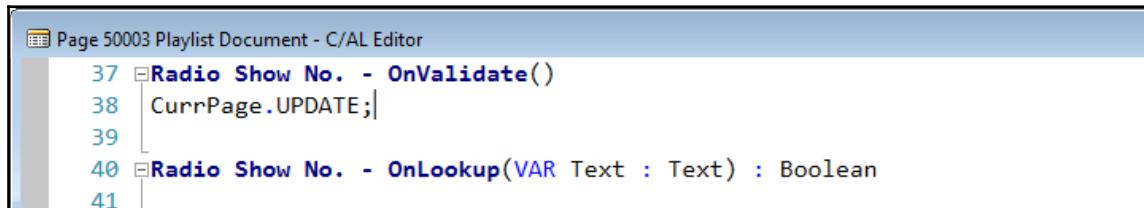
```

Table 50002 Playlist Header - C/AL Editor
11 11  No. - OnValidate()
12 12  |
13 13  No. - OnLookup()
14 14  |
15 15  Radio Show No. - OnValidate()
16 16  IF RadioShow.GET("Radio Show No.") THEN
17 17  BEGIN
18 18  "PSAs Required" := RadioShow."PSA Planned Quantity";
19 19  "Ads Required" := RadioShow."Ads Planned Quantity";
20 20  "News Required" := RadioShow."News Required";
21 21  "Sports Required" := RadioShow."Sports Required";
22 22  "Weather Required" := RadioShow."Weather Required";
23 23  END ELSE
24 24  BEGIN
25 25  CLEAR("PSAs Required");
26 26  CLEAR("Ads Required");
27 27  CLEAR("News Required");
28 28  CLEAR("Sports Required");
29 29  CLEAR("Weather Required");
30 30  END;
31

```

Even though the Radio Show No. was entered in the data field, our validation code needs to read the Radio Show record (here, defined as the Global variable RadioShow). Once we have read the Radio Show record, we can assign all five show feature requirements fields from the Radio Show record into the Playlist Header record.

Then, because two fields in the Playlist Header record are Lookup FlowFields, we need to Update the page after the entry of the **Radio Show No.**. The Update is done through a CurrPage.UPDATE command, as shown in the following screenshot:



```

Page 50003 Playlist Document - C/AL Editor
37 37  Radio Show No. - OnValidate()
38 38  CurrPage.UPDATE;
39 39  |
40 40  Radio Show No. - OnLookup(VAR Text : Text) : Boolean
41 41

```

The next Validation we need is to calculate the show End Time as soon as the Start Time is entered. The calculation is simple: add the length of the show to the Start Time. We have defined the Duration field in the Playlist Header to be a Lookup reference to the source field in the Radio Show record. As a result, to calculate with that field would require using a CALCFIELDS function first. Instead, we'll obtain the show length from the Radio Show record:



The screenshot shows the C/AL Editor for Table 50002 Playlist Header. The code in the OnValidate() trigger for the Start Time field is as follows:

```
46 46 Start Time - OnValidate()
47 47 RadioShow.GET("Radio Show No.");
48 48 "End Time" := "Start Time" + RadioShow."Run Time";
49 49
50 50 Start Time - OnLookup()
```

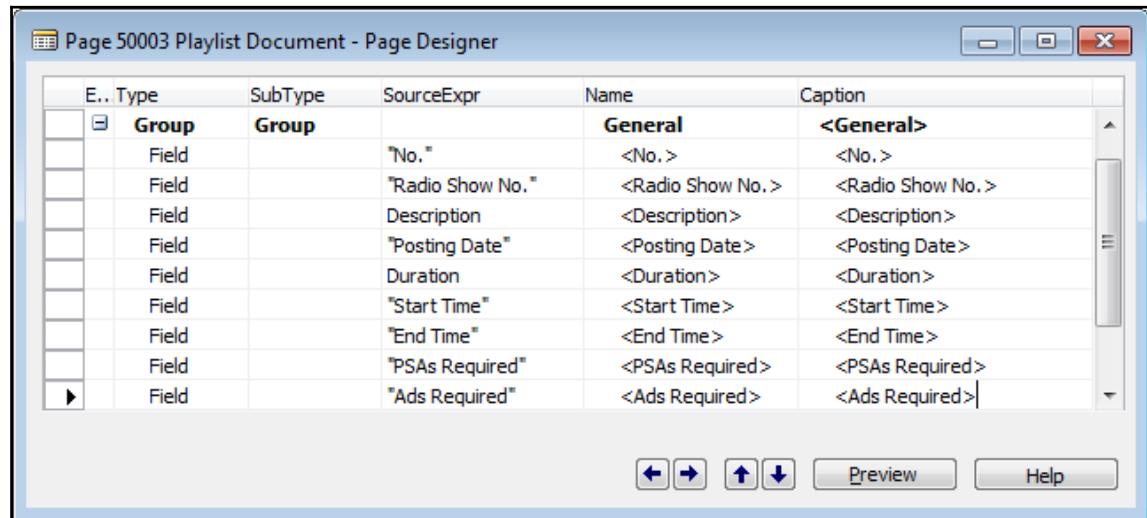
Now, we see we have one of those situations we sometimes encounter when developing a modification. It might have been better to have the Playlist Header Duration field be a **Normal** data field rather than a **FlowField**. If this is the only place where we will use Duration from the Playlist Header for calculation or assignment, then the current design is fine. Otherwise, perhaps, we should change the Duration field to a **Normal** field and assign Run Time from Radio Show to it at the same time the several requirements fields are assigned. At this point though, for the purposes of our WDTU scenario, we will stick with what we have already created.

## Creating the Playlist Subpage

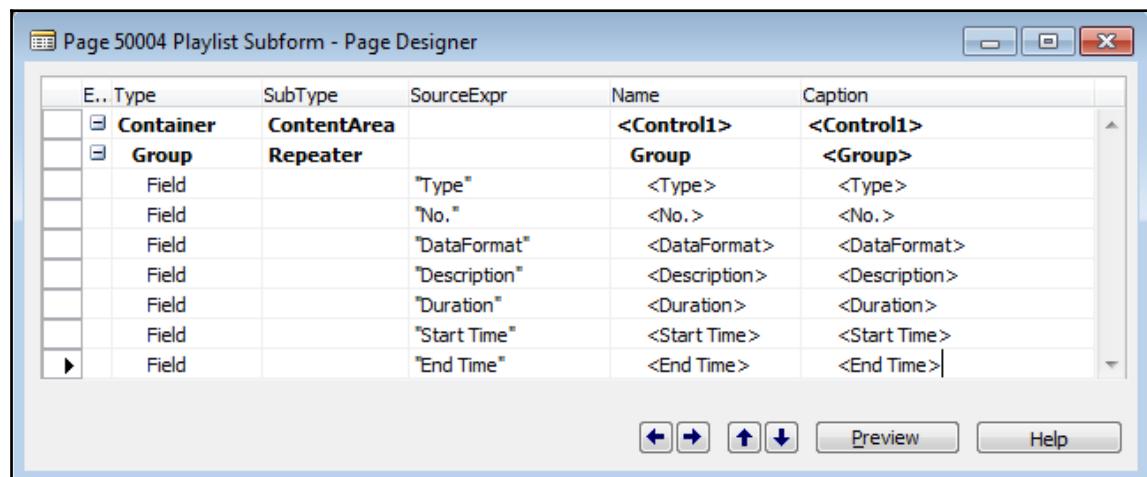


As of the time of writing this book, Microsoft still insists on referring to Subpages with the obsolete term Subform, a leftover from the Classic version of NAV. We will use the more accurate Subpage term but, as of this writing, your Development Environment objects will still sadly read "Subform".

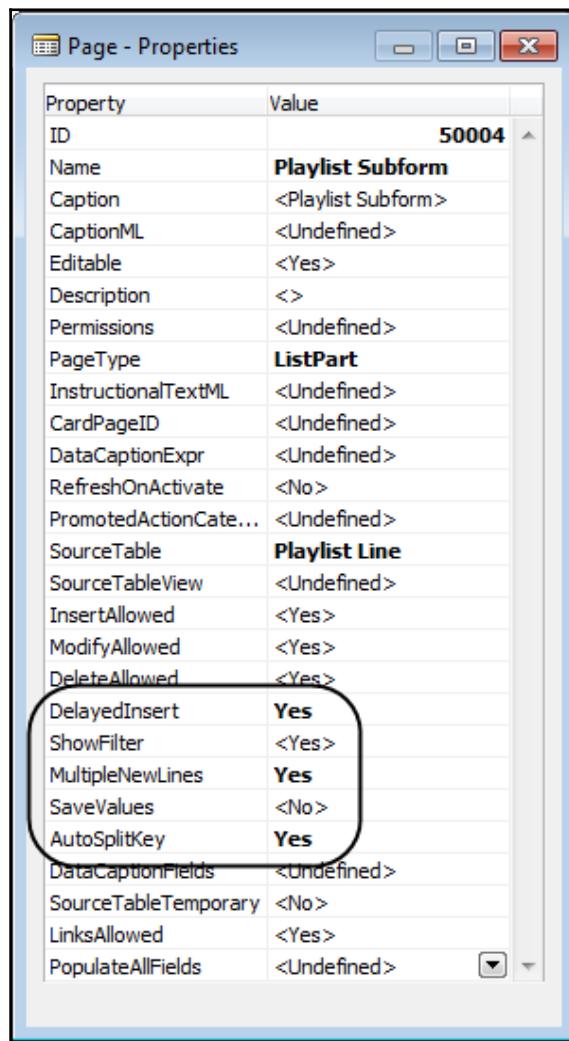
In Chapter 2, *Tables*, a homework assignment was to create Page 50003 Playlist Document. We should have used the Page Wizard to create that page, giving us something like this:



Another necessary part of a Document page is the Subpage. Our Subpage can be created using the Page Wizard to create a ListPart based on Table **50003 Playlist Line**. After we **Finish** the Wizard, we'll save and compile the page as **50004 Playlist Subpage**. The result will look like the following screenshot (*in which you can see the obsolete Subform identification*):

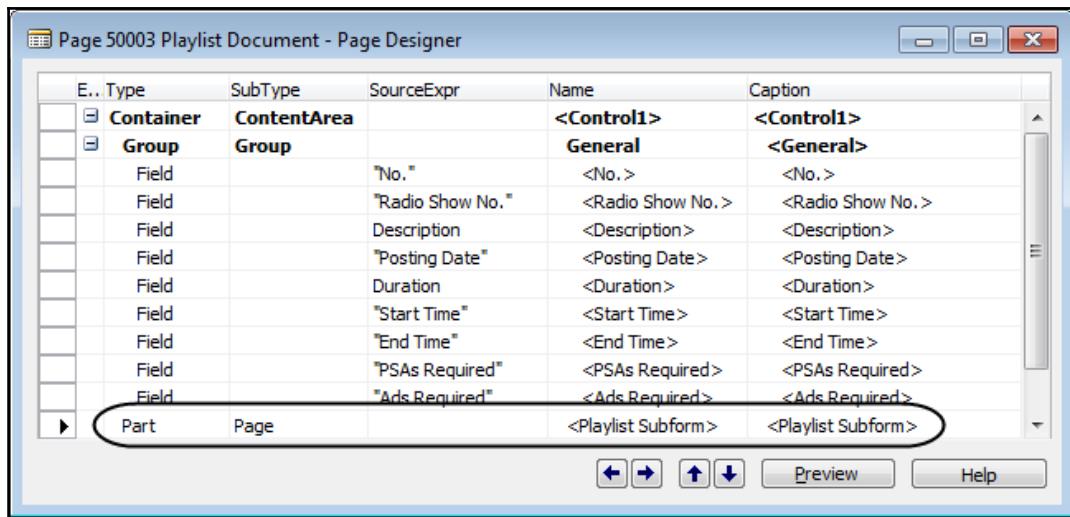


To make the document work the way we are used to having NAV Document forms work, we will need to set some properties for this new page. See the bolded properties circled in the following screenshot:

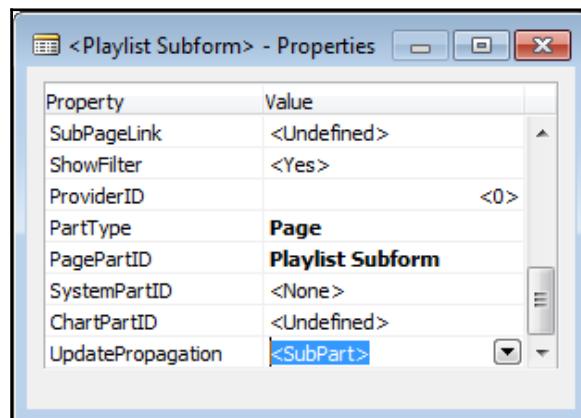


We have set the **DelayedInsert** and **AutoSplitKey** properties all to Yes. These settings will allow the Playlist Lines to not be saved until the primary key fields are all entered (**DelayedInsert**) and will support the easy insertion of new entries between two existing lines (**AutoSplitKey**).

Finally, we will need to connect our new **Playlist Subpage Listpart Page 50004** to the **Playlist Document Page 50003** to give us a basic and complete document page. All we need to do to accomplish that is add a new Part line to **Page 50003** as shown in the following screenshot:

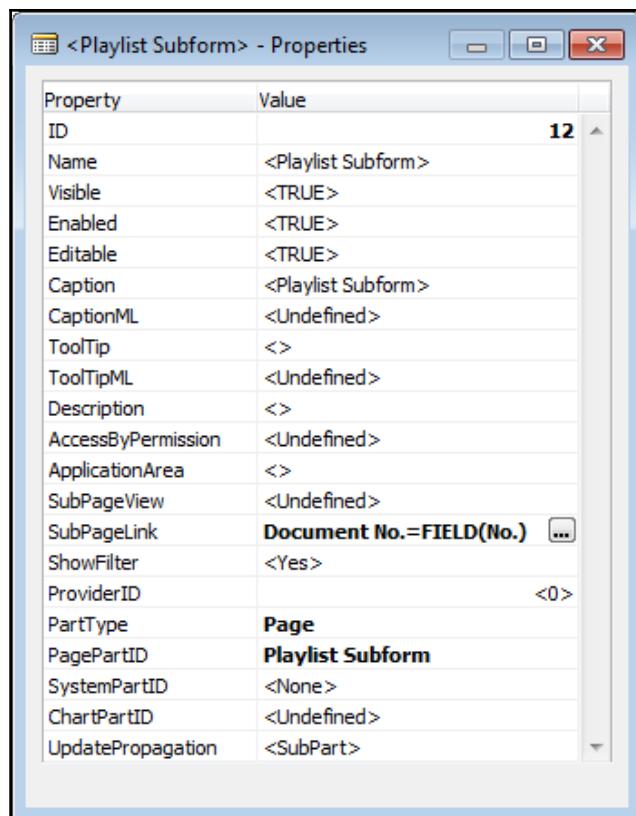


Entering the **PagePartID** property in the Part line properties, as shown in the following screenshot, will populate the **Name** and **Caption** fields, as shown in the preceding screenshot:



Note the highlighted property, **UpdatePropagation**, in the preceding screenshot. This property can be set to either **SubPart** (the default) or **Both**. This controls how the parent and child pages are updated when a change is made in the data displayed by the child page (the SubPart). If the property value is **SubPart**, only the child page information display is updated. If the property value is **Both**, both the child and parent page displays are updated. This is useful when data in the SubPage is updated and the change affects the data displayed in the parent page.

The last step in this phase of our development is to set the Page Part property **SubPageLink** to automatically filter the contents of the Subpage to only be the lines that are children of the parent record showing in the card-like portion of the Document page (at the top) of the page:



## Playlist Line validations

The Playlist Line data fields are as follows:

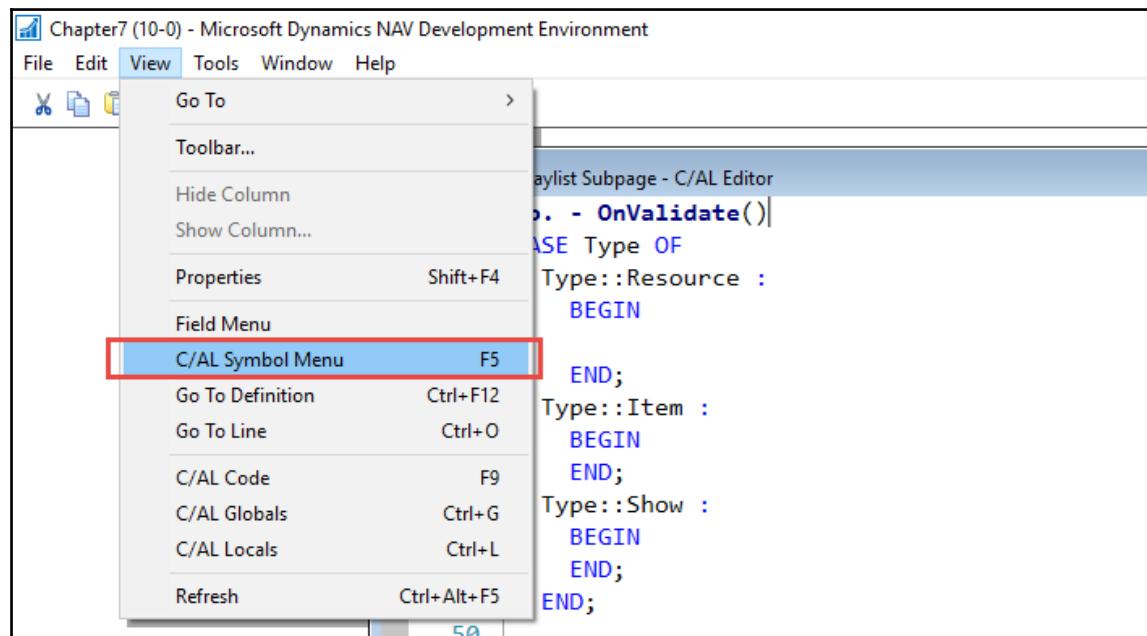
- **Document No.:** This is the automatically assigned link to the No. field in the parent Playlist record (the automatic assignment is based on the page, Playlist Subpage properties, which we'll take care of when working on the Playlist pages)
- **Line No.:** This is an automatically assigned number (based on page properties), the rightmost field in the Playlist Line primary key
- **Type:** This is a user-selected Option, defining if this entry is a Resource, such as an announcer; a Show, such as a News show; or an Item, such as a recording to play on the air
- **No.:** This is the ID number of the selected entry in its parent table
- **DataFormat:** This is information from the Item table for a show or recording
- **Description:** This is assigned from the parent table, but can be edited by the user
- **Duration, Start Time, and End Time:** This is information about a show or recording indicating the length and its position within the schedule of this Radio Show

The source of contents of the **No.**, **DataFormat**, **Description**, and time-related fields of the record depend on the **Type** field. If the **Type** is **Resource**, the fields are filled in from the **Resource** table; for **Item**, from the **Item** table; for **Show**, from the **Radio Show** table. To support this, our **OnValidate** code looks at the **Type** entry and uses a **CASE** statement to choose which set of actions to take.

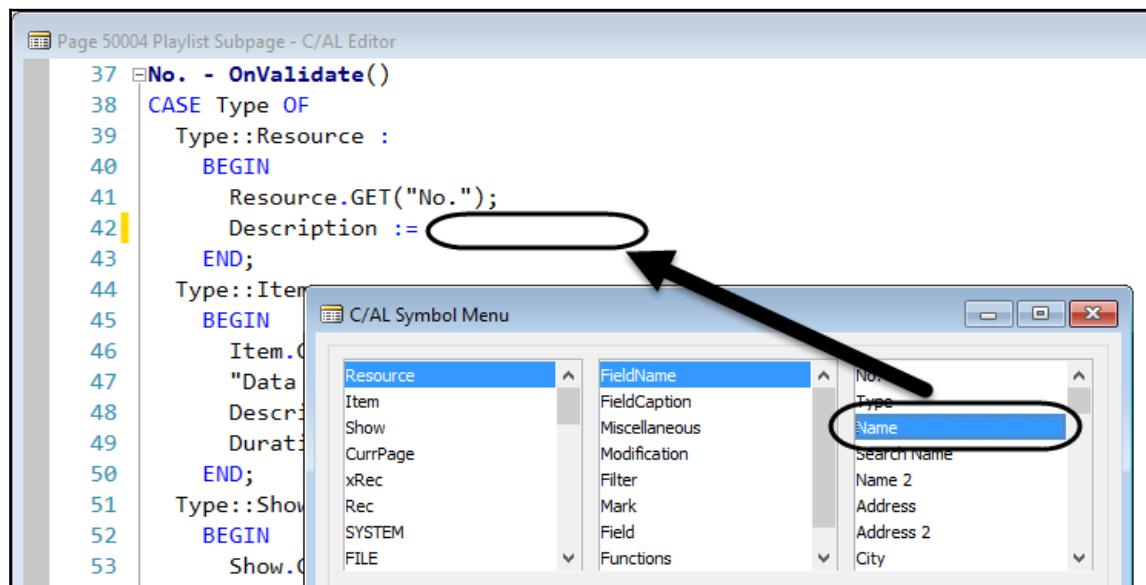
First, we will build the basic **CASE** statement, as shown in the following code screenshot, and compile it. That way, we can ensure that we've got all the components of the structure in place and only need to fill in the logic for each of the option choices:

```
37  □ No. - OnValidate()
38  CASE Type OF
39    Type::Resource :
40      BEGIN
41      END;
42    Type::Item :
43      BEGIN
44      END;
45    Type::Show :
46      BEGIN
47      END;
48
49
```

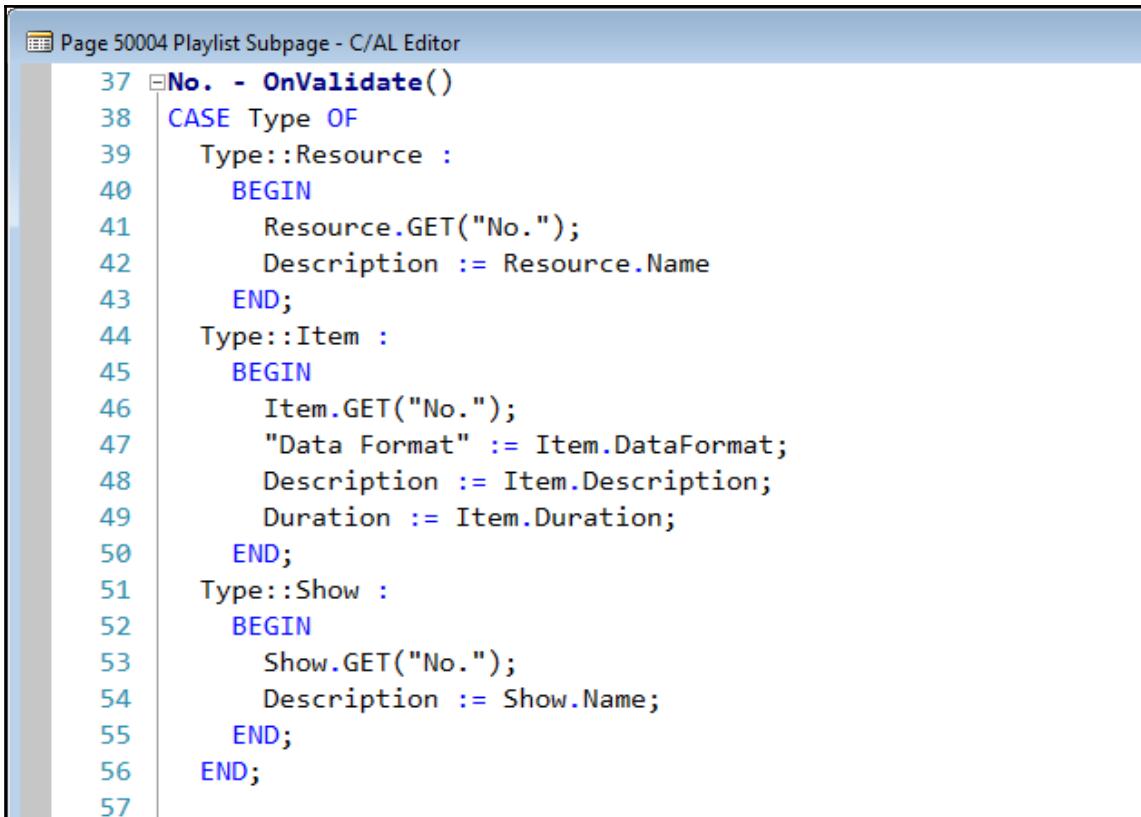
Next, we must add a Global Variable for each of the tables from which we will pull the data: Resource, Item, and Radio Show. That done, use of the **C/AL Symbol Menu** makes it much easier to find the correct field names for each of the variables we want to select to assign to the Playlist Line record fields or use Intellisense:



Once the C/AL Symbol Menu is displayed, first, we will select the source object or variable in the left column. In this case, that's the Resource table. Then, we will select the subcategory in the middle column. We want a list of the fields in the **Resource** table, so we will select **FieldName**. Then, in the third column, a list of **Resource FieldNames** is displayed. We will select **Name** to assign to the **Playlist Line Description** field. When we double-click on the desired **FieldName**, it will be inserted at the point of focus in the C/AL Editor, as shown in the following screenshot:



When we are all done constructing the CASE statement, it should look like this:



The screenshot shows the C/AL Editor interface with the title "Page 50004 Playlist Subpage - C/AL Editor". The code editor contains the following C/AL code:

```
37 37  No. - OnValidate()
38 38  CASE Type OF
39 39  Type::Resource :
40 40  BEGIN
41 41      Resource.GET("No.");
42 42      Description := Resource.Name
43 43  END;
44 44  Type::Item :
45 45  BEGIN
46 46      Item.GET("No.");
47 47      "Data Format" := Item.DataFormat;
48 48      Description := Item.Description;
49 49      Duration := Item.Duration;
50 50  END;
51 51  Type::Show :
52 52  BEGIN
53 53      Show.GET("No.");
54 54      Description := Show.Name;
55 55  END;
56 56
57 57
```

The last set of OnValidate code we need to add is to calculate the End Time from the supplied Start Time and Duration (or Start Time from the End Time and Duration), as shown in the following screenshot:

The screenshot shows the C/AL Editor for Table 50003 Playlist Line. The code is organized into several sections:

- Section 43: **Start Time - OnValidate()**
  - IF Duration <> 0 THEN
  - "End Time" := "Start Time" + Duration;
- Section 47: **Start Time - OnLookup()**
- Section 49: **End Time - OnValidate()**
  - IF "Start Time" <> 0T THEN
  - Duration := "End Time" - "Start Time";
- Section 53: **End Time - OnLookup()**

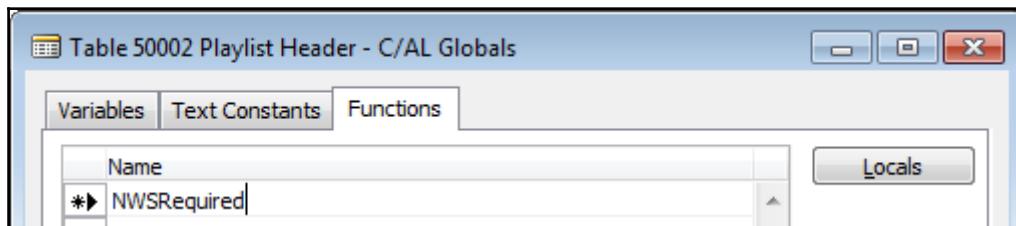
Obviously, the design could be expanded to have the Duration value be user editable, along with an appropriate change in the C/AL logic. After our initial work on the Playlist functionality is completed, making that change would be a good exercise for you, as would be the addition of the "housekeeping" commands to clear out fields that are not used by the assigned record Type, such as clearing the **DataFormat** field for a Show record.

## Creating a function for our Factbox

For this application, we want our FactBox to display information relating to the specific Radio Show we are scheduling. The information to be displayed includes the five show segment requirements and the status of fulfillment (counts) of those requirements by the data entered to date. The requirements come from the Playlist Header fields: PSAs Required, Ads Required, News Required, Sports Required, and Weather Required. The counts come from summing up data in the Playline Line records for a show. We can use the Playlist Header fields' PSA Count and Ad Count for those two counts. These counts can be obtained through the FlowField property definitions we defined earlier for these two fields.

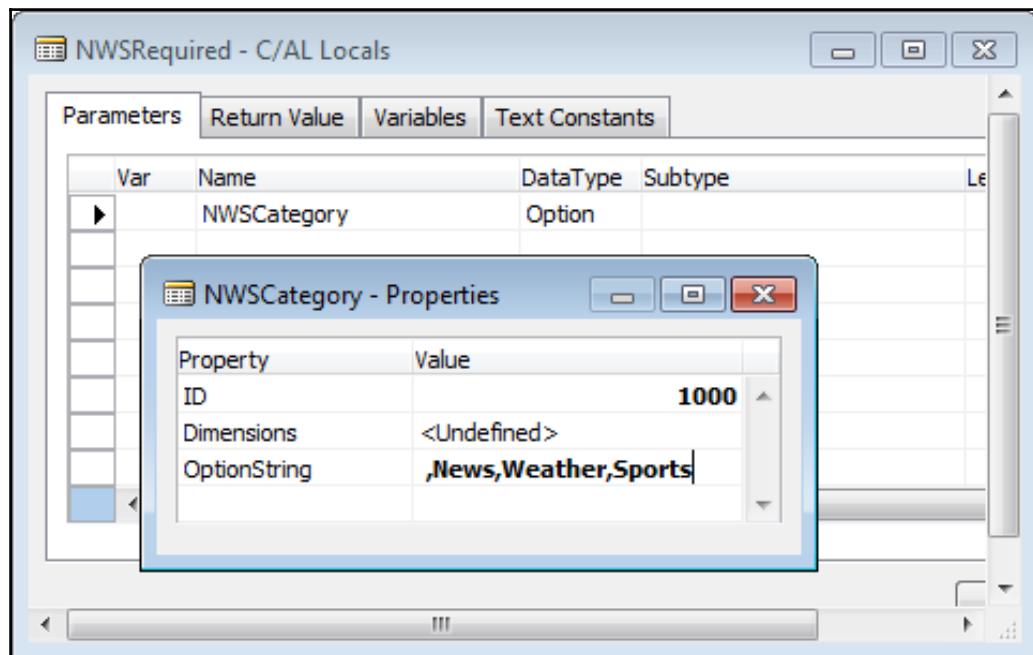
For the other three counts, we must read through the Playlist Lines and sum up each of the counts. To accomplish that, we'll create a Function that we can call from the FactBox page. Since our new function is local to the Playlist Header and Playlist Lines tables, we will define the function in the Playlist Header (table 50002).

The first step in defining the function is to enter its name in the **Functions** tab of the **Globals** screen for the object. Open the object, Table 50002, by clicking on **Table | Design**, then click on **View | C/AL Globals**, and then click on the **Functions** tab. Enter the name of the new function, access its **Properties** (Properties icon or *Shift + F4*), and set the **Local** property to No so the function can be called from the Factbox Page:

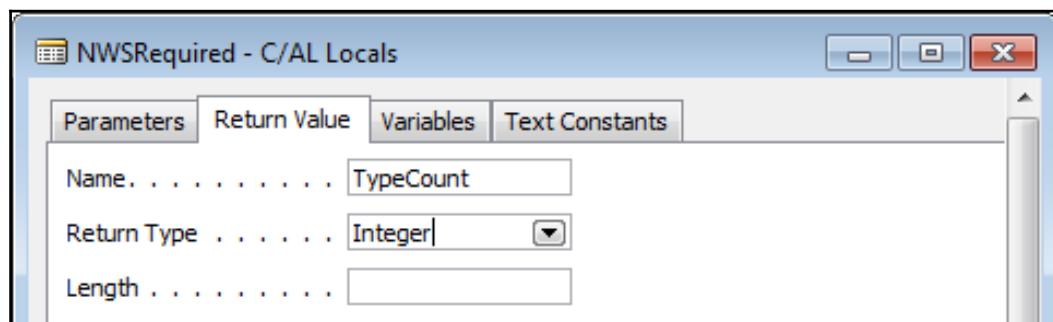


Then, after clicking on the **Locals** button and displaying the **Parameters** tab, we can enter the parameters we want to pass to the function. In this case, we want the parameter to be passed by value, not by reference, so we do not check the **Var** checkbox on the **Parameter** line. For more information about parameter passing, look at the **Help** sections, *C/AL Function Calls* and *How to: Add a Function to a Codeunit*, as well as the **Create a Function** section in *Chapter 6, Introduction to C/SIDE and C/AL*. The name of the Parameter is local to the function.

Since we want a single function that will serve to count Playlist Lines that are News, Weather, or Sports, the parameter we pass it in will be an Option code of News, Weather, or Sports. The **Option Subtype** sequence must be the same as those in the **Type** field in the **Playlist Lines** table; otherwise, we will have **Options** mismatching:



The return value we want back from this function is an Integer count. Again, the variable name is local. We define the Return Value on the appropriate tab, as shown in the following screenshot:



The logic of our counting process is described as follows:

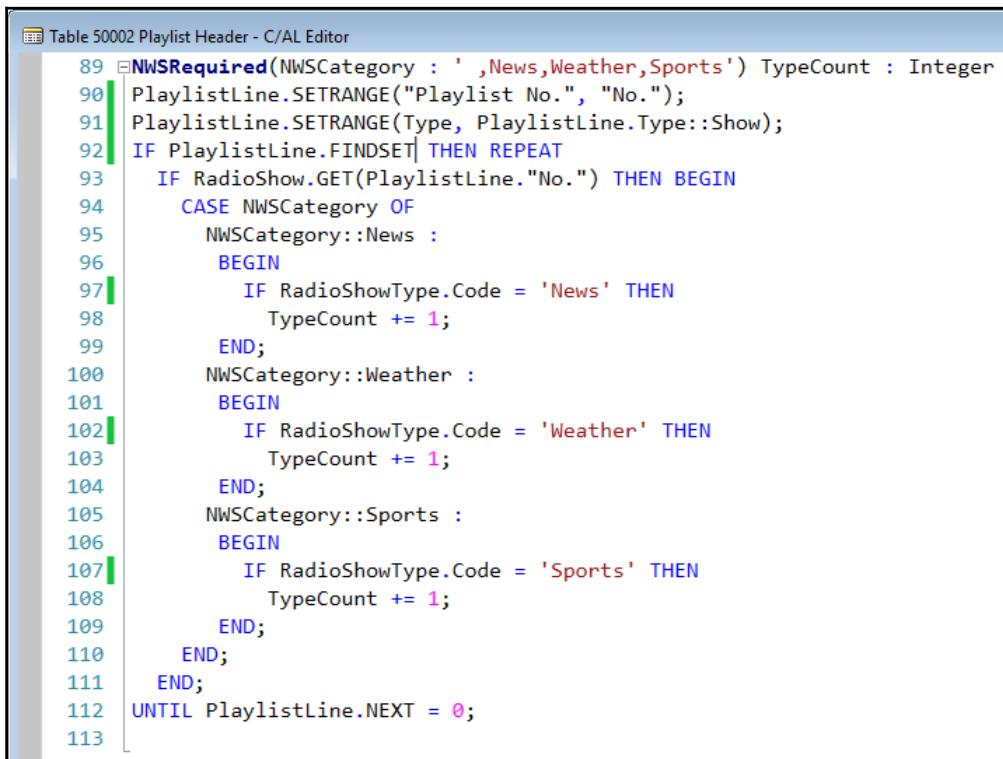
1. Filter the Playlist Line table for the Radio Show we are scheduling and for segment entries (Playlist Line) that represent shows.
2. Look up the Radio Show record for each of those records.
3. Using the data from the Radio Show record, look up the Radio Show Type code.
4. In the Radio Show Type record, use the News, Weather, or Sports fields to determine which Playlist Line counter should be incremented.

Based on this logic, we must have **Local Variables** defined for the three tables: PlaylistLine, RadioShow, and RadioShowType. The following screenshot shows those Local Variables:

The screenshot shows a software interface titled "NWSRequired - C/AL Locals". It has tabs at the top: "Parameters", "Return Value", "Variables", and "Text Constants". The "Variables" tab is selected. A table below lists three variables:

Name	DataType	Subtype	Len
PlaylistLine	Record	Playlist Line	
RadioShow	Record	Radio Show	
*► RadioShowType	Record	Radio Show Type	

Translating our pseudocode into executable C/AL, our function looks like the following screenshot:



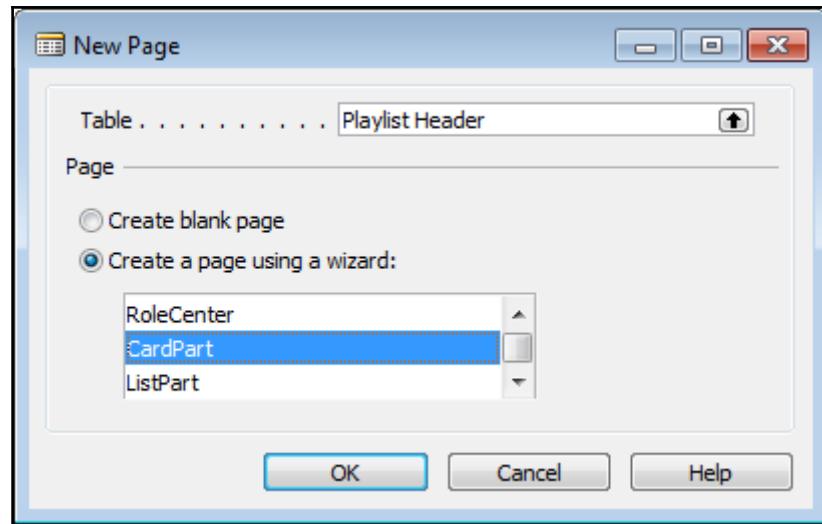
The screenshot shows the C/AL Editor interface with the title "Table 50002 Playlist Header - C/AL Editor". The code is as follows:

```
89  NWSRequired(NWSCategory : 'News,Weather,Sports') TypeCount : Integer
90  PlaylistLine.SETRANGE("Playlist No.", "No.");
91  PlaylistLine.SETRANGE(Type, PlaylistLine.Type::Show);
92  IF PlaylistLine.FINDSET| THEN REPEAT
93    IF RadioShow.GET(PlaylistLine."No.") THEN BEGIN
94      CASE NWSCategory OF
95        NWSCategory::News :
96          BEGIN
97            IF RadioShowType.Code = 'News' THEN
98              TypeCount += 1;
99            END;
100       NWSCategory::Weather :
101         BEGIN
102           IF RadioShowType.Code = 'Weather' THEN
103             TypeCount += 1;
104           END;
105       NWSCategory::Sports :
106         BEGIN
107           IF RadioShowType.Code = 'Sports' THEN
108             TypeCount += 1;
109           END;
110         END;
111     END;
112   UNTIL PlaylistLine.NEXT = 0;
113
```

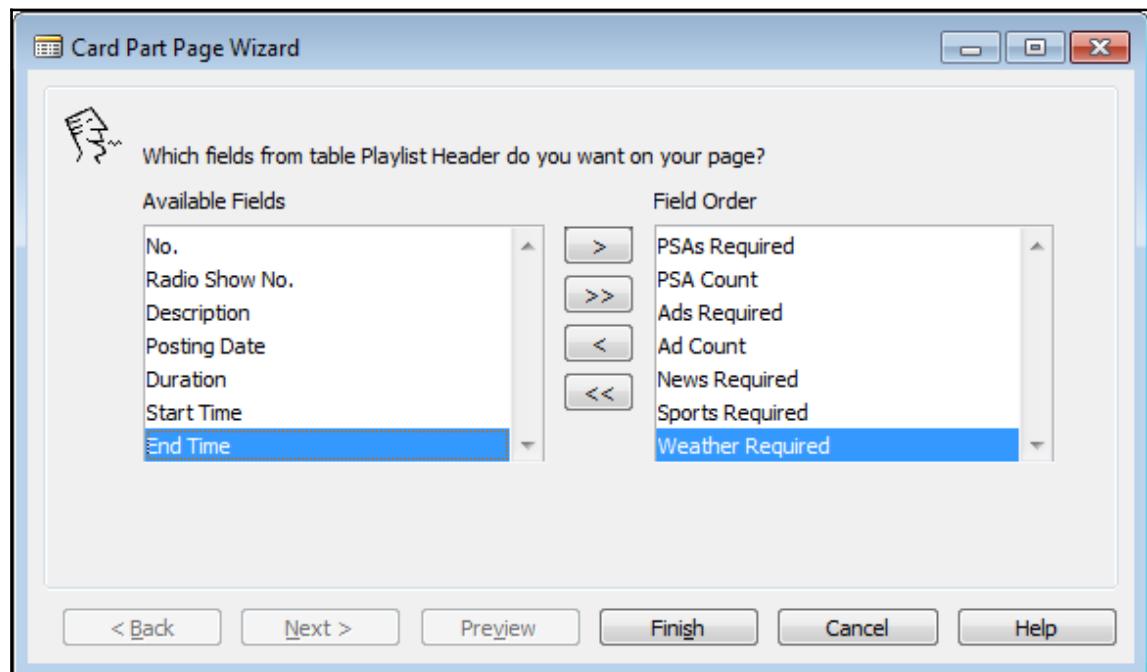
In the process of writing this code, we notice another design flaw. We defined the type of Radio Show with a code that allows users to enter their choice of text strings. We just wrote code that depends on the contents of that text string being specific values. A better design would be to have the critical field be an Option data type so we could depend on the choices being members of a predefined set. However, the Code field is our Primary Key field, and we probably shouldn't use an Option field as the Primary Key. We will continue with our example with the design as is, but you should consider how to improve it. Making that improvement will be excellent practice with C/AL for you.

## Creating a Factbox Page

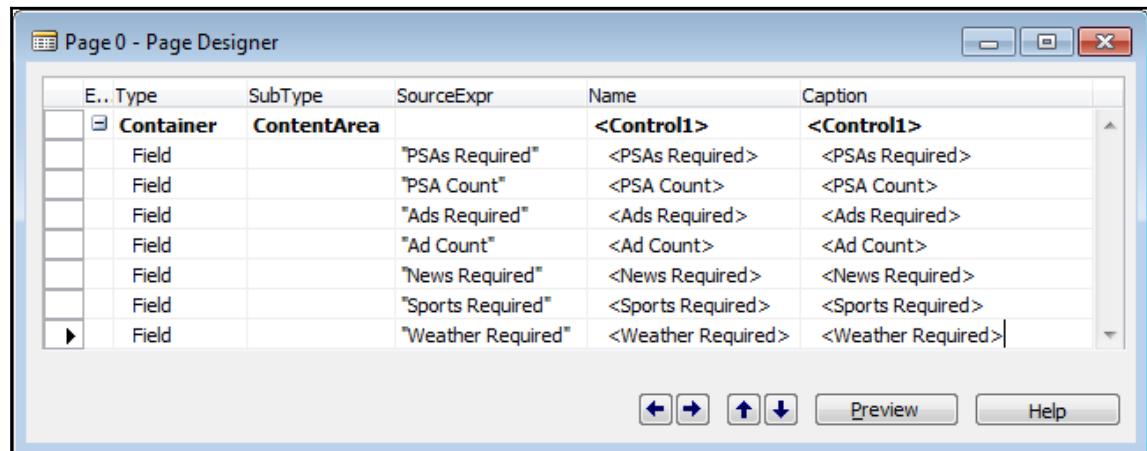
All the hard work is now done. We just have to define a Factbox page and add it to the Playlist page. We can create a Factbox page using the **New Page** Wizard to define a **CardPart**:



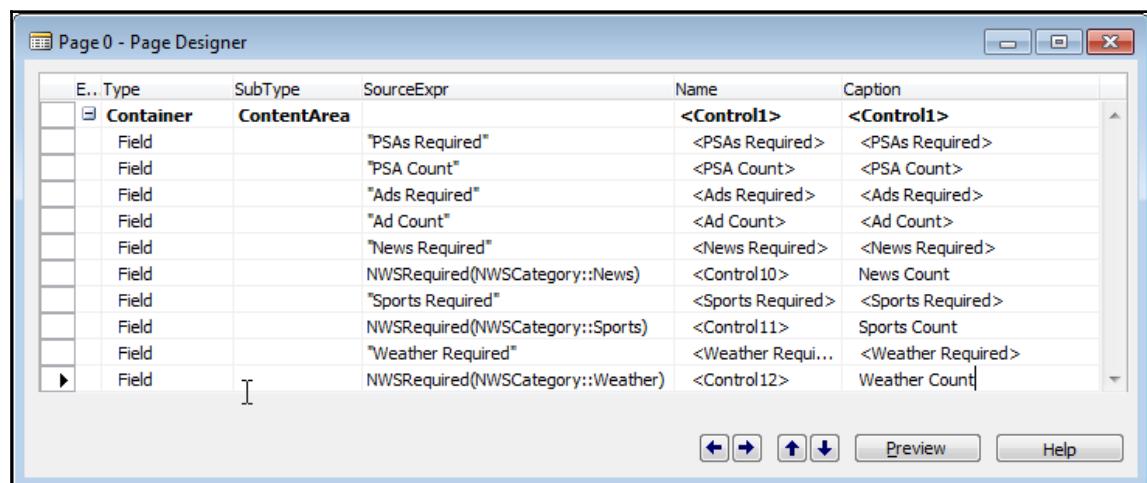
Our fact box will contain the fields from the Playlist Header that relate to two of the five required show segments: the **PSA Count** and **Ad Count** fields, as shown in the following screenshot:



Once we exit the Page Wizard into the Page Designer, we will have a page layout that looks like the following screenshot:

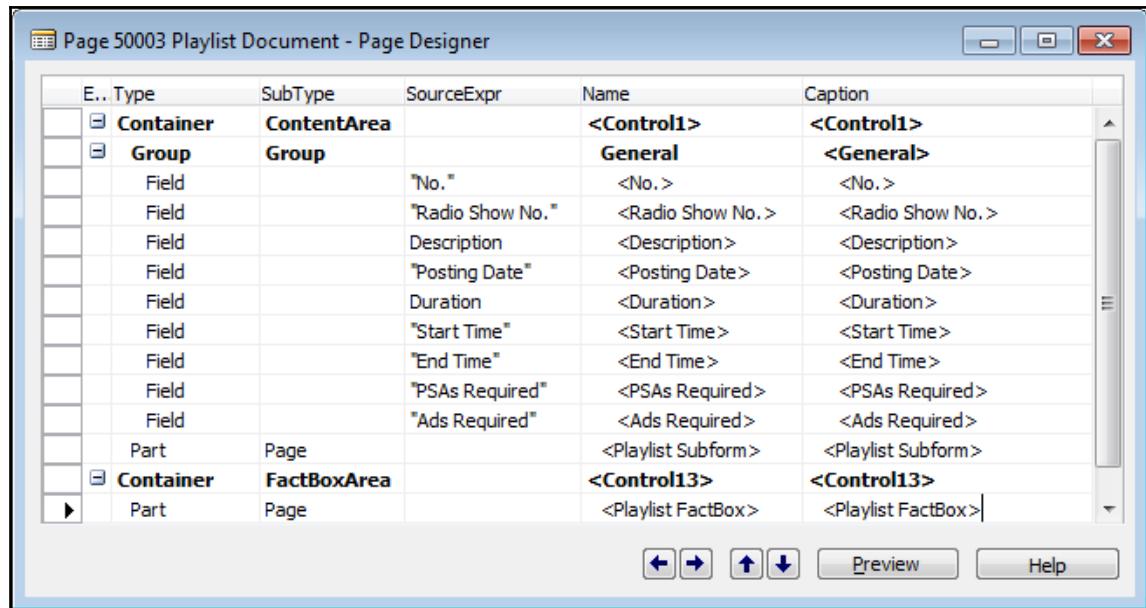


At this point, we will need to add the logic to take advantage of the **NWSRequired** function that we created earlier. This function is designed to return the count of the segment type identified in the calling parameter. Since a line on a page can be an expression, we can simply code the function calls right on the page lines with an appropriate caption defined, as we can see in the following screenshot. The only other task has been to define a Global variable, **NWSCategory**, which is defined as an **Option** with the choices of **News**, **Weather**, and **Sports**. This variable is used for the calling parameter. We will intersperse the Count lines for a consistent appearance on the Page, as shown in the following screenshot:

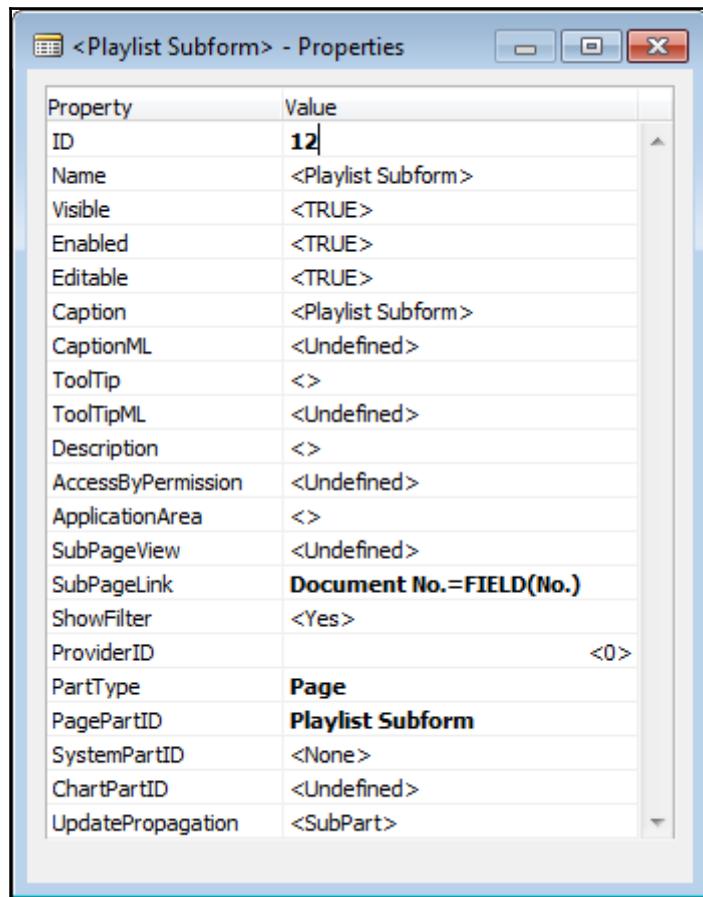


We will save the new FactBox page as Page 50010, named Playlist FactBox.

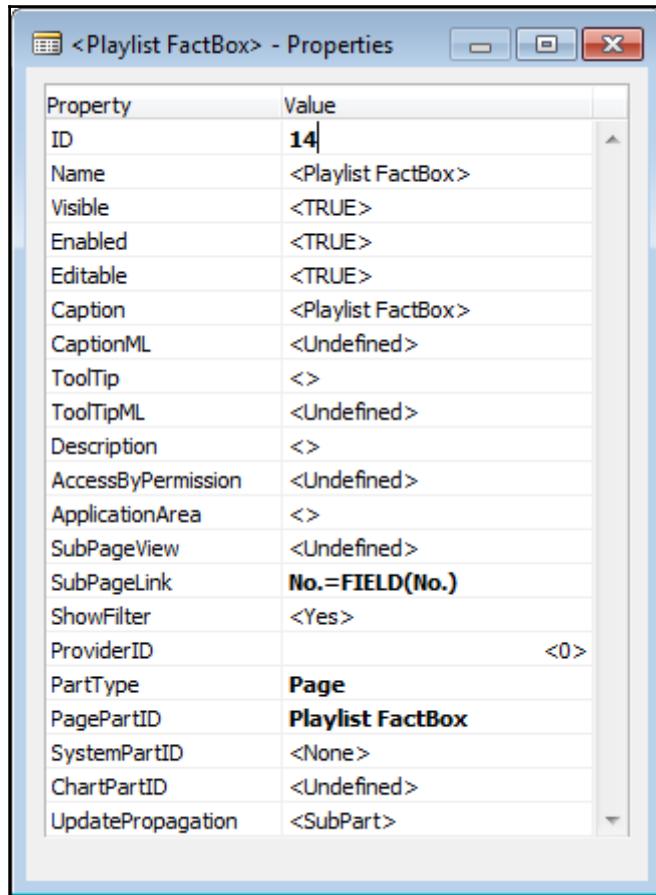
One final development step is required. We must connect the new FactBox CardPart to the Playlist Document. All that is required is to define the FactBox Area and add our FactBox as an element in that area:



Properties for the Playlist Subpage Pagepart look like the following screenshot:

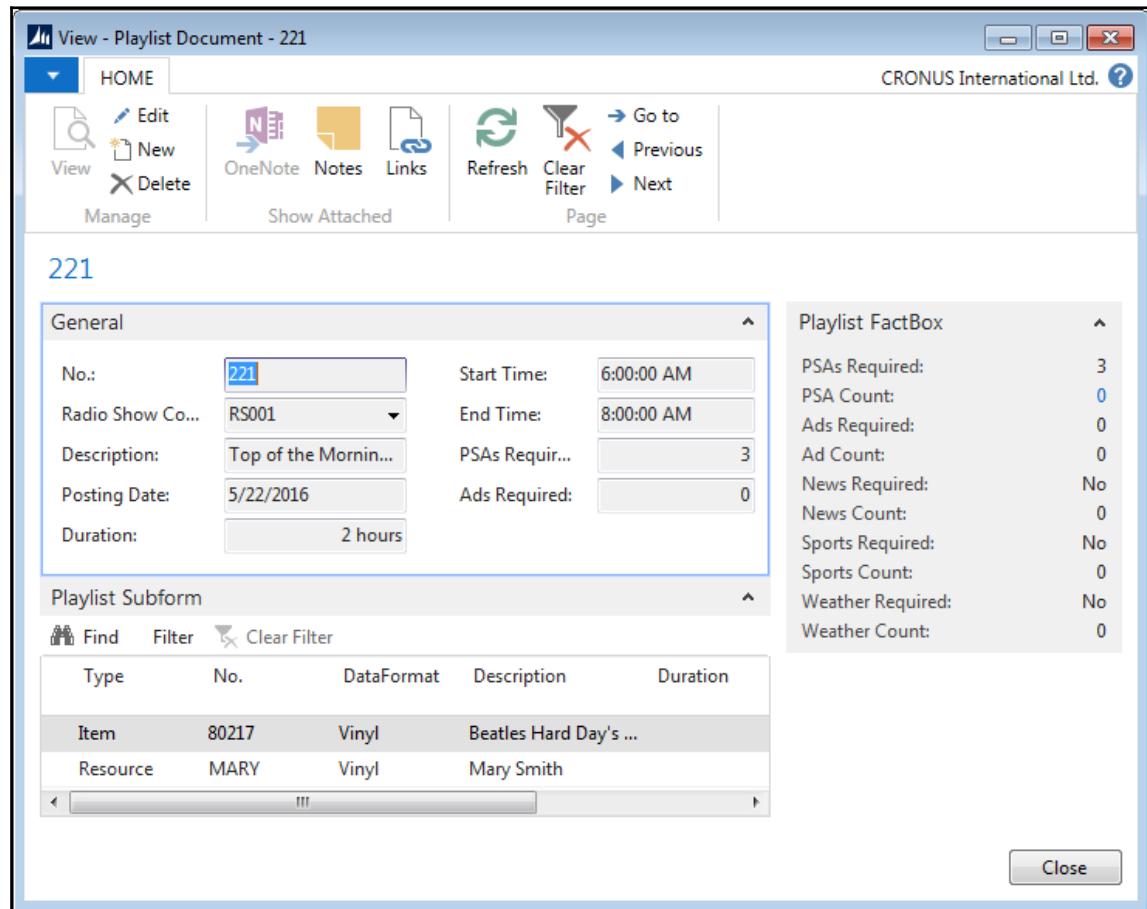


Properties for the FactBox Pagepart are shown in the following screenshot:



Multiple FactBoxes can be part of a primary page. If we look at Page 21 - Customer Card, we will see a FactBox Area with eight FactBoxes, of which, two are System Parts.

The end result of our development effort is shown in the following screenshot when we Run Page 50000 - Playlist with some sample test data, which we entered by running the various tables:



## Review questions

1. Which three of the following are valid date related NAV functions?
  - a) DATE2DWY
  - b) CALCDATE
  - c) DMY2DATE
  - d) DATE2NUM
2. RESET is used to clear the current sort key setting from a record. True or False?
3. Which functions can be used to cause FlowFields to be calculated? Choose two:
  - a) CALCSUMS
  - b) CALCFIELDS
  - c) SETAUTOCALCFIELDS
  - d) SUMFLOWFIELD
4. Which of the following functions should be used within a report `OnAfterGetRecord` trigger to end processing just for a single iteration of the trigger? Choose one:
  - a) EXIT
  - b) BREAK
  - c) QUIT
  - d) SKIP
5. The WORKDATE value can be set to a different value from the System Date. True or False?
6. Only one FactBox is allowed on a page. True or False?
7. Braces {} are used as a special form of a repeating CASE statement. True or False?

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. A FILTERGROUP function should not be used in custom code. True or False?
10. The REPEAT – UNTIL looping structure is most often used to control data reading processes. True or False?
11. Which of the following formats of MODIFY will cause the table's OnModify trigger to fire? Choose one:
  - a) MODIFY
  - b) MODIFY(TRUE)
  - c) MODIFY(RUN)
  - d) MODIFY(READY)
12. MARKING a group of records creates a special index; therefore, MARKING is especially efficient. True or False?
13. A CASE statement structure should never be used in place of a nested IF statement structure. True or False?
14. An Average FlowField requires which one of the following?
  - a) A record key
  - b) SQL Server database
  - c) An Integer variable
  - d) A Decimal variable
15. The TESTFIELD function can be used to assign new values to a variable. True or False?
16. The VALIDATE function can be used to assign new values to a variable. True or False?

17. The C/AL Symbol Menu can be used for several of the following purposes.  
Choose two:
  - a) Find applicable functions
  - b) Test coded functions
  - c) Use entries as a template for function syntax and arguments
  - d) Translate text constants into a support language
18. Documentation cannot be integrated into in-line C/AL code. True or False?
19. If MAINTAINSIFTINDEX is set to NO and CALCFIELDS is invoked, the process will terminate with an error. True or False?
20. SETRANGE is often used to clear all filtering from a single field. True or False?

## Summary

In this chapter, we covered a number of practical tools and topics regarding C/AL coding and development. We started by reviewing methods, and then we dived into a long list of functions that we will need on a frequent basis.

We covered a variety of selected data-centric functions, including some for computation and validation, some for data conversion, and others for 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. Finally, we put a number of these to work in an enhancement for our WDTU application.

In the next chapter, we will move from the details of the functions to the broader view of C/AL development integration into the standard NAV code and debugging techniques.

# 8

## Advanced NAV Development Tools

*"Quality isn't something you lay on top of subjects and objects like tinsel on a Christmas tree. Quality must be in the heartwood."*

- Robert Pirsig

*"Often when you think you're at the end of something, you're at the beginning of something else."*

- Fred Rogers

Because NAV is extremely flexible and suitable for addressing many problem types, there are a lot of choices for advanced NAV topics. We'll try to cover those that will be most helpful in the effort of putting together a complete implementation.

First, we will review the overall structure of NAV as an application software system, aiming for a basic understanding of the process flow of the system along with some of the utility functions built into the standard product. Before designing modifications for NAV, it is important to have a good understanding of the structural "style" of the software so that our enhancements are designed for a better fit.

Second, we will review some special components of the NAV system that allow us to accomplish more at less cost. These resources include features such as XMLPorts and Web Services that we can build on, and which help us use standard interface structures to connect with the world outside of NAV system. Fortunately, NAV has a good supply of such features.

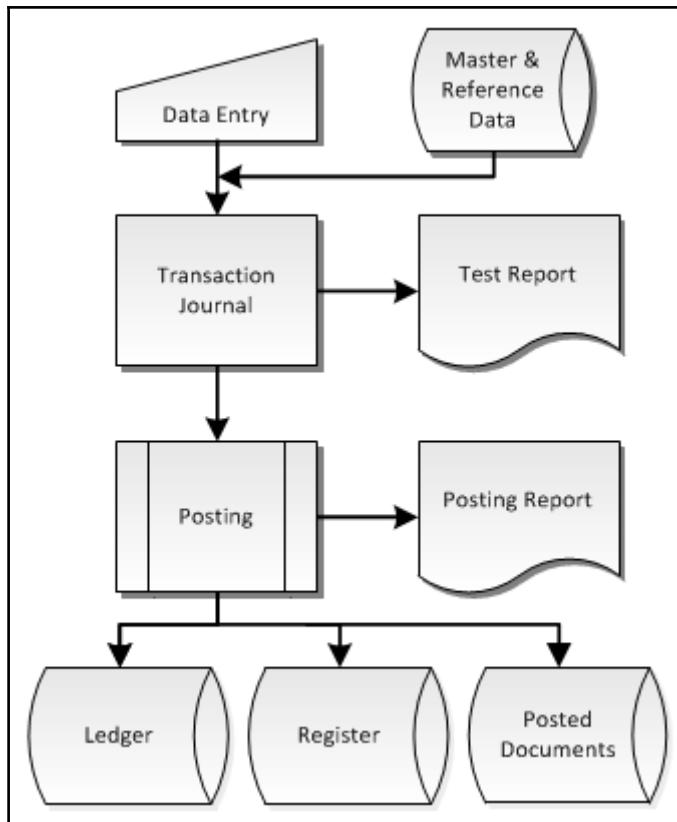
Topics we will cover in this chapter include:

- Role Center pages
- The Navigation Pane and Action menus
- XMLports
- Web services

## NAV process flow

Primary data, such as sales orders, purchase orders, production orders, and financial transactions, flow through the NAV system, as follows:

- **Initial Setup:** This is where the Essential Master data, reference data, and control and setup data is entered. Most of this preparation is done when the system (or a new application) is prepared for production use.
- **Transaction Entry:** Transactions are entered into Documents and then transferred as part of a Posting sequence into a **Journal** table, or data may be entered directly into a **Journal** table. Data is preliminarily validated as it is entered with master and auxiliary data tables being referenced as appropriate. Entry can be via manual keying, an automated transaction generation process, or an import function that brings in transaction data from another system.
- **Validate:** This step provides for additional data validation processing of a set of one or more transactions, often in batches, prior to submitting it to Posting.
- **Post:** This step posts a Journal Batch that includes completing transaction data validation, adding entries to one or more Ledgers, perhaps also updating a Register and Document history.
- **Utilize:** This is where we access the data via Pages, Queries, and/or Reports, including those that feed Web Services and other consumers of data. At this point, total flexibility exists. Whatever tools are appropriate for users' needs should be used, whether internal to NAV or external (external tools are often referred to as Business Intelligence (BI) tools). NAV's built-in capabilities for data manipulation, extraction, and presentation should not be overlooked.
- **Maintenance:** This is the continued maintenance of all NAV data as appropriate:



The preceding diagram provides a simplified picture of the flow of application data through a NAV system. Many of the transaction types include additional reporting, multiple ledgers to update, and even auxiliary processing. However, this represents the basic data flow in NAV whenever a Journal and a Ledger table are involved.

When we enhance an existing NAV functional area, such as Jobs or Service Management, we may need to enhance related process flow elements by adding new fields to journals, ledgers, posted documents, and so on.



It's always a good idea to add new fields rather than changing the use of standard fields. It makes debugging, maintenance, and upgrading all easier.

When we create a new functional area, we will likely want to replicate the standard NAV process flow in some form for the new application's data. For example, for our WDTU application, we will handle the entry of Playlists in the same fashion as a Journal is entered.

A day's Playlists would be similar to a Journal Batch in another application. After a day's shows have been broadcast, the completed Playlist will be posted into the Radio Show Ledger as a permanent record of the broadcasts.

## Initial Setup and Data Preparation

Data must be maintained as new Master data becomes available or when various system operating parameters and other elements change. 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 approach applies to both setup data entry and ongoing production transaction data entry.

For example, a sales person might initialize a new customer entry with name, address, and phone number, just entering the data to which they have easy 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. With this additional data, the new customer record is ready for production use.

The NAV data entry approach allows the system to be updated on an incremental basis as the data arrives, providing an operational flexibility many systems lack.. This works because often data comes into an organization on a piecemeal basis. The other side of this flexibility is added responsibility for users to ensure that partially updated information is completed in a timely fashion. For the user organization who can't manage that, it may be necessary to create special procedures or even system customizations that enforce the necessary discipline.

## 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. Validations are based on the evaluation of the individual transaction data plus the related Master records and associated reference tables, for example, lookups being satisfied, application or system setup parameter constraints being met, and so on.

## Testing and Posting the Journal batch

Any additional validations that are needed 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 Posting processes. The actual Posting of a Journal batch occurs after the transaction data is completely validated. Depending on the specific application, 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 to one or more Ledgers and, sometimes, to a document history table. When a Journal Entry is Posted to a Ledger, it becomes part of the permanent accounting record. Most data cannot be changed or deleted once it resides in a Ledger (an example exception would be the Due Date on a Payable).

Register tables may also be updated during Posting, recording the ID number ranges of ledger entries posted, when posted, 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 revisions to be made by Posting reversing entries, rather than by 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.

## Utilizing and maintaining the data

The data in a NAV system can be accessed by means of Pages, Queries, and/or Reports, 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 includes the ability to create page parts for graphing, with a wide variety of predefined chart page parts included as part of the standard distribution. We can also create our own chart parts using tools delivered with the system or available from blogs or MSDN.

The NAV database design approach could be referred to as a **rational normalization**. NAV isn't constrained by a rigid normalized data structure, where every 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 user convenience is improved by duplicating data across tables, NAV does so. In addition, the duplication of master file data into transactions allows for one-time modification of data when appropriate, such as a ship-to address in a Sales Order. That data duplication also often greatly simplifies data access for analysis.

## Data maintenance

As with any database-oriented application software, ongoing maintenance of Master data, reference data, and setup and control data is required. In NAV, maintenance of data uses many of the same data preparation tools as were initially used to set up the system.

## Role Center pages

One of the key features of NAV 2017 is the Role Tailored user experience centered on Role Centers tied to user work roles. The Role Tailored approach provides a single point of entry and access into the system for each user through their assigned Role Center. Each user's Role Center acts as their home page. Each Role Center focuses on the tasks needed to support its users' jobs throughout the day. Primary tasks are front and center, while the Action Ribbon and Departments Menu provide easy access to other functions, making them only a click or two away.

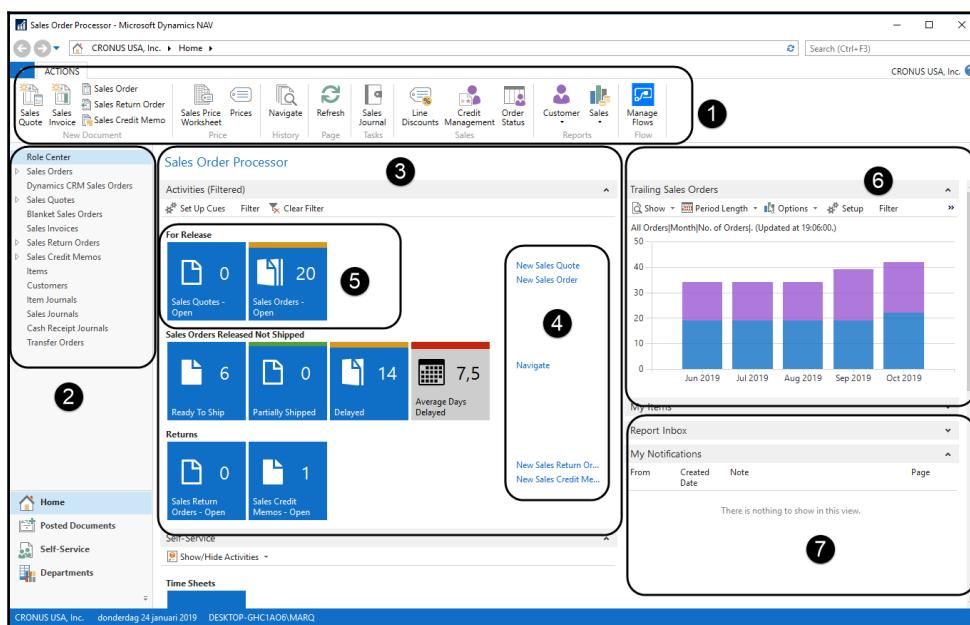
The standard NAV 2017 distribution from Microsoft contains more than two dozen different Role Center pages, identified for user roles such as Bookkeeper, Sales Manager, Shop Supervisor, Purchasing Agent, and so on (Page 9011 is identified as a **Foundation** rather than as a **Role Center** or **RC**). Some localized NAV distributions may have additional Role Center pages included. It is critical to realize that the Role Centers supplied out of the box are not generally intended to be used directly out of the box.

The Role Center pages should be used as templates for custom Role Centers tailored to the specific work role requirements of the individual customer implementation.

One of the very critical tasks of implementing a new system is to analyze the work flow and responsibilities of the system's intended users and configure Role Centers to fit the users. In some cases, the supplied Role Centers can be used with minimal tailoring. Sometimes it will be necessary to create complete new Role Centers. Even then, we will often be able to start with a copy of an existing Role Center Page, which we will modify as required. In any case, it is important to understand the structure of the Role Center Page and how it is built.

## Role Center structure

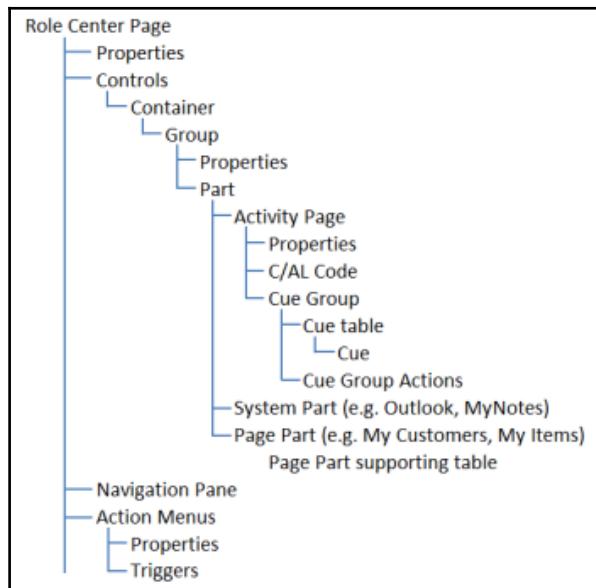
The following screenshot shows Page 9006 - Order Processor Role Center:



The components of the Role Center highlighted in the preceding screenshot are as follows:

1. Action Ribbon
2. Navigation Pane
3. Activity Pane
4. Cue Group Actions (in Cue Groups)
5. Cues (in Cue Groups)
6. Page Parts
7. System Part

A general representation of the structure of a **Role Center** Page is shown in the following outline:



We need to understand the construction of a Role Center Page so that we are prepared to modify an existing Role Center or create a new one. First, we'll take a look at Page 9006 - Order Processor Role Center in the Page Designer:

E.. Type	SubType	Name	Caption
▶ Container	<b>RoleCenterArea</b>	<Control1900000008>	<Control1900000008>
└ Group	Group	<Control1900724808>	<Control1900724808>
Part	Page	<SO Processor Activities>	<SO Processor Activities>
Part	Page	<Team Member Activities>	<Team Member Activities>
Part	Page	<My Customers>	<My Customers>
Part	Page	<Power BI Report Spinner Part>	<Power BI Report Spinner ...>
└ Group	Group	<Control1900724708>	<Control1900724708>
Part	Page	<Trailing Sales Orders Chart>	<Trailing Sales Orders Cha...>
Part	Page	<My Job Queue>	<My Job Queue>
Part	Page	<My Items>	<My Items>
Part	Page	<Report Inbox Part>	<Report Inbox Part>
Part	Page	<Connect Online>	<Connect Online>
Part	System	<MyNotes>	<MyNotes>

The Role Center page layout should look familiar, because it's very similar in structure to the pages we've designed previously. What is specific to a Role Center page? There is a Container control of **SubType** `RoleCenterArea`. This is required for a Role Center page. There are two Group Controls that represent the two columns (left and right) of the Role Center page display. Each group contains several parts that show up individually in the Role Center display.

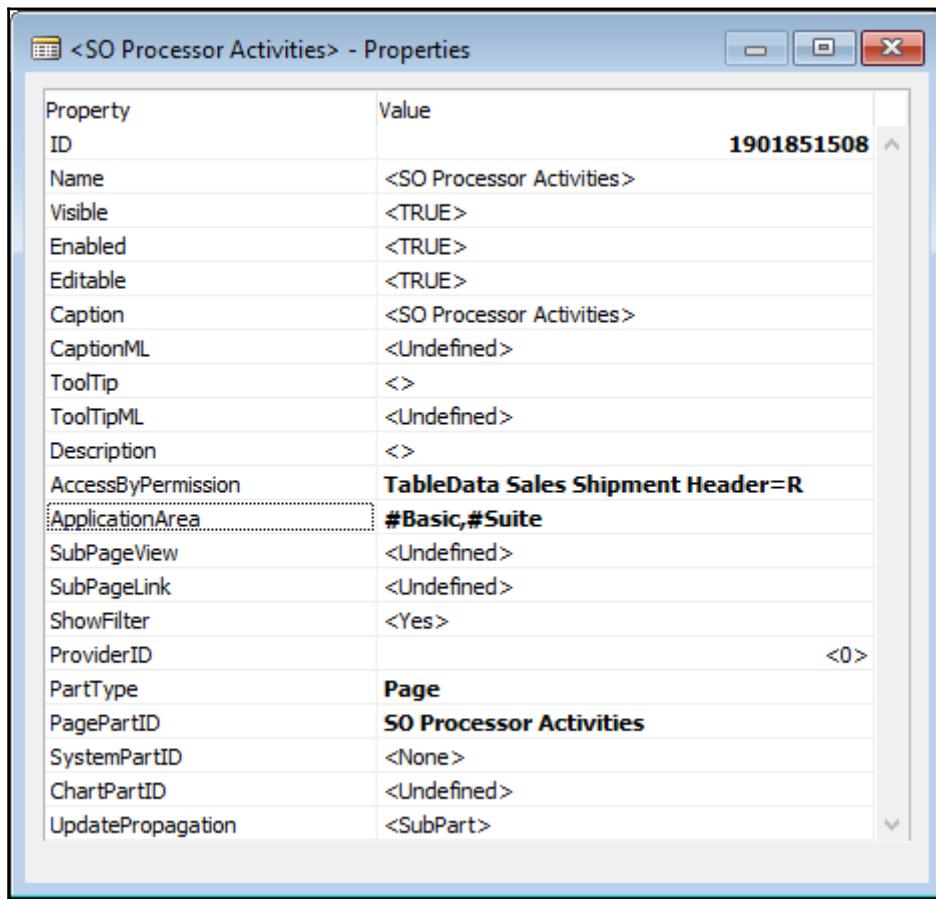
Role Center page Properties are accessed by highlighting the first blank line on the Page Designer form (the line below all of the defined controls), then by clicking on the Properties icon, or we can right-click and choose the **Properties** option, or click on **View | Properties** or press *Shift + F4*. Note that the **PageType** is `RoleCenter`, and there is no **Source Table**. The page properties not shown in this screenshot are all default values:

The screenshot shows a Windows-style dialog box titled "Page - Properties". It contains a table with two columns: "Property" and "Value". The properties listed are: ID (9006), Name (Order Processor Role Center), Caption (Role Center), CaptionML (ENU=Role Center), Editable (<Yes>), Description (<>), Permissions (<Undefined>), PageType (RoleCenter), InstructionalTextML (<Undefined>), and CardPageID (<Undefined>). The "PageType" property is highlighted in red.

Property	Value
ID	<b>9006</b>
Name	<b>Order Processor Role Center</b>
Caption	Role Center
CaptionML	<b>ENU=Role Center</b>
Editable	<Yes>
Description	<>
Permissions	<Undefined>
PageType	<b>RoleCenter</b>
InstructionalTextML	<Undefined>
CardPageID	<Undefined>

## Role Center activities page

As the Group Control has no underlying code or settings, we'll take a quick look at the first Part Control's Properties. The **PagePartID** property is Page SO Processor Activities:



The **ApplicationArea** property highlighted in the preceding screenshot works in combination with the **ApplicationArea** function to manage which application controls are visible for a particular user session. More information on these can be found in the online *Developer and IT-Pro Help for Microsoft Dynamics NAV*.

## Cue Groups and Cues

Now we'll focus on Page 9060 - SO Processor Activities. Designing that page, we see the following layout. Comparing the controls we see there to those of the Role Center, we can see this Page Part is the source of the **Activities** section of the Role Center Page. There are three **CueGroup** Controls - **ForRelease**, **Sales Orders Released Not Shipped**, and **Returns**. In each **CueGroup**, there are the **Field Controls** for the individual **Cues**:

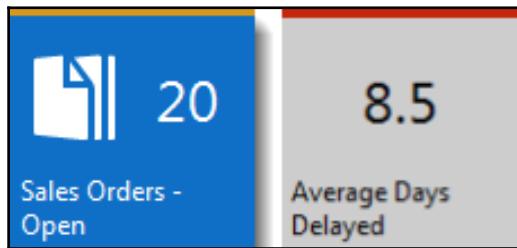
Page 9060 SO Processor Activities - Page Designer					
E..	Type	SubType	SourceExpr	Name	Caption
	Container	ContentArea		<Control1900000...	<Control1900000001>
	Group	CueGroup		<Control1>	<b>For Release</b>
	Field		"Sales Quotes - Open"	<Sales Quotes - ...	<Sales Quotes - Open>
	Field		"Sales Orders - Open"	<Sales Orders - ...	<Sales Orders - Open>
	Group	CueGroup		<Control8>	<b>Sales Orders Released Not Shipp...</b>
	Field		"Ready to Ship"	ReadyToShip	Ready To Ship
	Field		"Partially Shipped"	PartiallyShipped	Partially Shipped
	Field		Delayed	DelayedOrders	Delayed
	Field		"Average Days Delayed"	<Average Days ...	<Average Days Delayed>
	Group	CueGroup		<Control18>	<b>Returns</b>
	Field		"Sales Return Orders - Open"	<Sales Return O...	<Sales Return Orders - Open>
	Field		"Sales Credit Memos - Open"	<Sales Credit Me...	<Sales Credit Memos - Open>

An individual Cue is displayed as an iconic shortcut to a filtered list through a FlowField, or to a Query or other data source through a Normal field. The stack of papers in the Cue icon resulting from a filtered value represents an idea of the number of records in that list. The actual number of entries is also displayed next to the icon (see the **Sales Orders - Open** example in the following screenshot).



The Stack image is the default icon for a Cue and changes automatically based on the data. If required, the icon can be changed into any of several hundred other images. A very useful tool for finding these images can be found on Mibuso at <https://mibuso.com/downloads/dynamics-nav-image-library>.

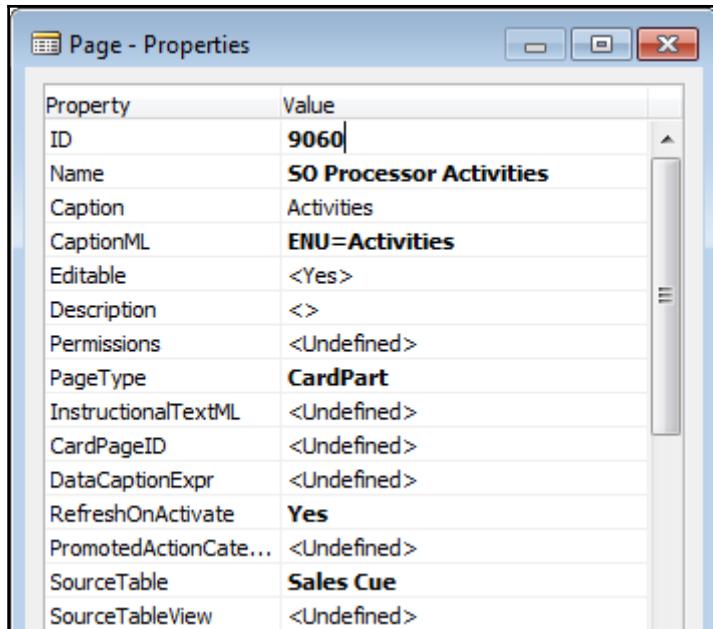
The purpose of this type of Cue is to provide a single-click point of access to a specific user task. The set of these Cues is intended to represent the full set of primary activities for a user based on their work Role:



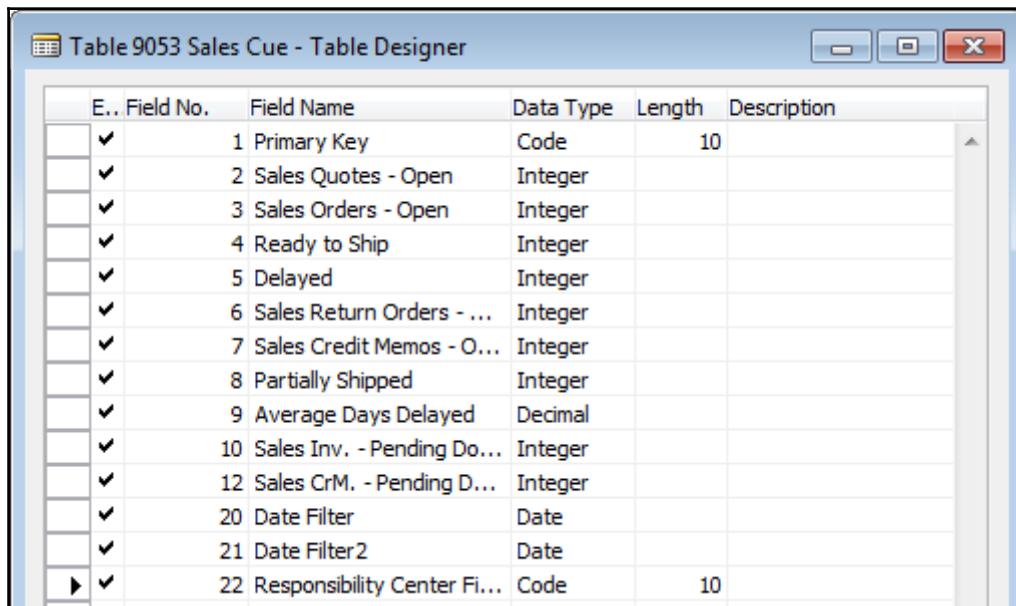
NAV 2017 provides another Cue format (such as the **Average Days Delayed** Cue in the preceding screenshot), which is based on a calculated value stored in a **Normal** field.

## Cue source table

In the Properties of the **SO Processor Activities** page, we can see that this is a **PageType** of **CardPart** tied to **SourceTable SalesCue**:



Next, we want to **Design** the referenced table, **Sales Cue**, to see how it is constructed:



The screenshot shows the Microsoft Dynamics NAV Table Designer window titled "Table 9053 Sales Cue - Table Designer". The table has the following columns: E.., Field No., Field Name, Data Type, Length, and Description. The data is as follows:

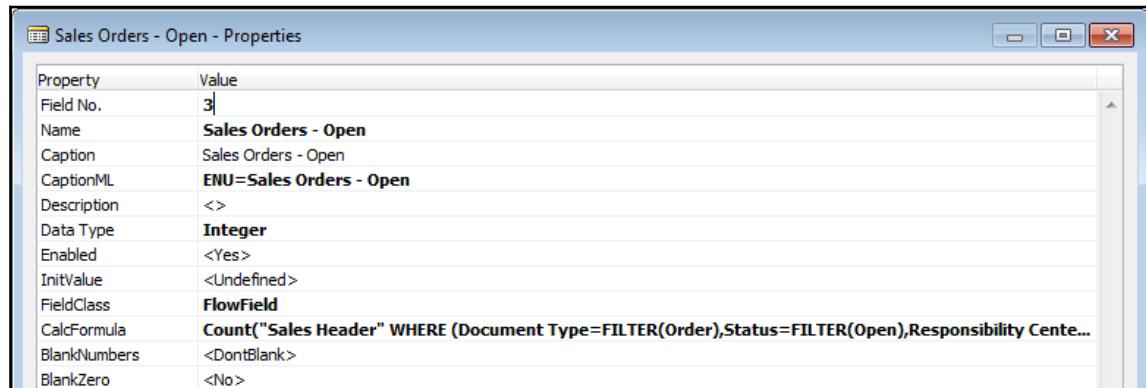
E..	Field No.	Field Name	Data Type	Length	Description
	1	Primary Key	Code	10	
	2	Sales Quotes - Open	Integer		
	3	Sales Orders - Open	Integer		
	4	Ready to Ship	Integer		
	5	Delayed	Integer		
	6	Sales Return Orders - ...	Integer		
	7	Sales Credit Memos - O...	Integer		
	8	Partially Shipped	Integer		
	9	Average Days Delayed	Decimal		
	10	Sales Inv. - Pending Do...	Integer		
	12	Sales CrM. - Pending D...	Integer		
	20	Date Filter	Date		
	21	Date Filter2	Date		
	22	Responsibility Center Fi...	Code	10	

As we see in the preceding screenshot, there is a simply structured table, with an integer field for each of the action Cues and a decimal field for the information Cue, all of which were displayed in the Role Center we are analyzing. There is also a key field, two fields identified as Date Filters, and a field identified as a Responsibility Center Filter.

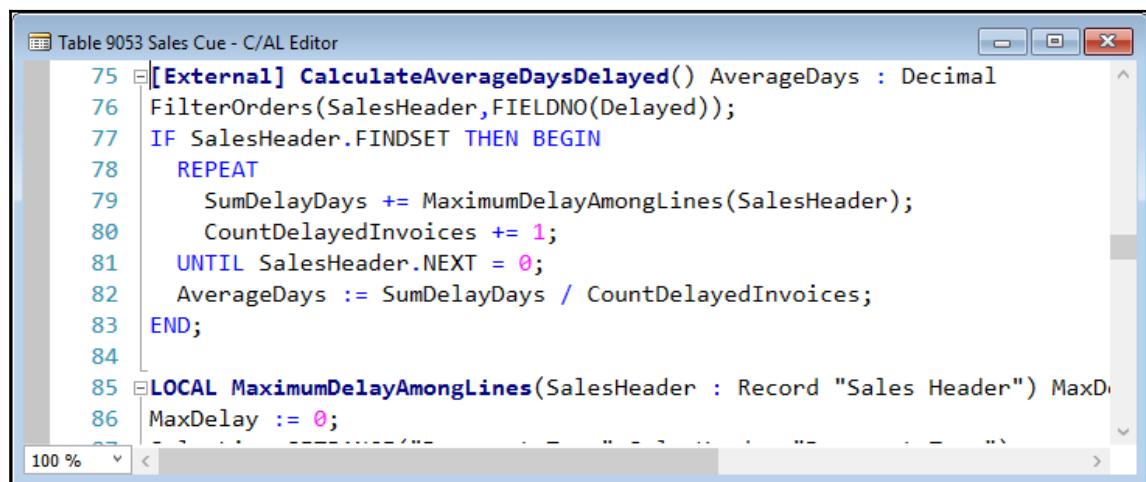


The Design Pattern used for Cue tables is the Singleton Pattern, which can be found on the Microsoft Design Patterns Wiki (<https://community.dynamics.com/nav/w/designpatterns/282.cue-table>).

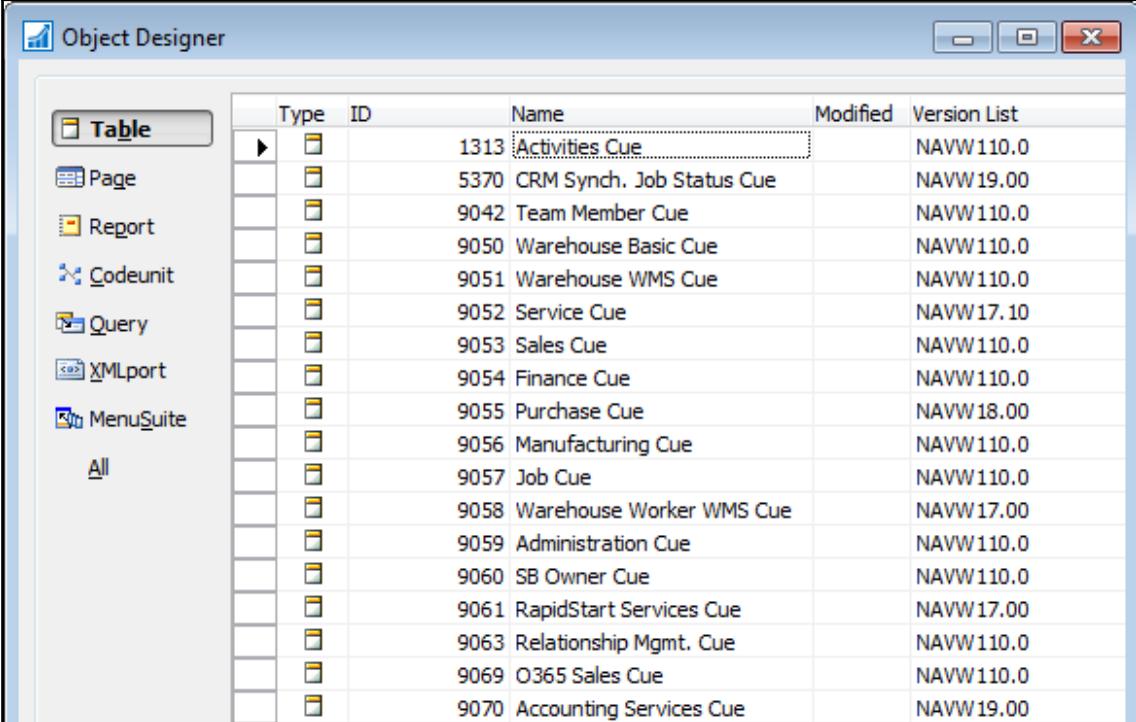
When we display the properties of one of these integer fields, **Sales Orders - Open**, we find it is a FlowField providing a Count of the Sales Orders with a Status of Open:



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. If we think about what the Cues show (a count) and how FlowFields work (a calculation based on a range of data records), we can see this is a simple, direct method of providing the information necessary to support Cue displays. Clicking on the action Cue (the count) opens up the list of records being counted. The information Cue is tied to a decimal field, which is computed by means of a function, as shown in the following screenshot, in the Cue table that is invoked from the Role Center page when the Cue is displayed:



The following screenshot shows the list of Cue tables. Each of the Cue tables contains a series of FlowFields and other fields that support a set of Cues. Some Cue tables service more than one of the Role Center pages:

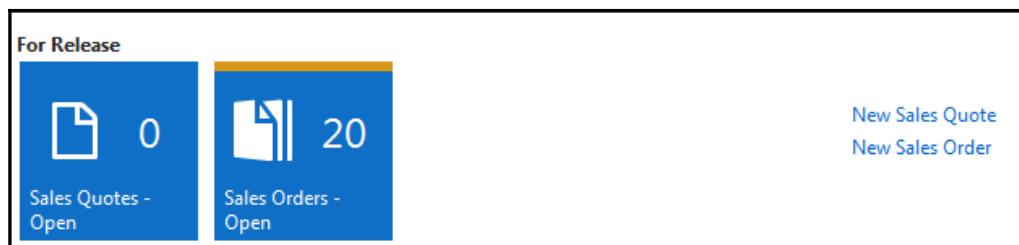


The screenshot shows the Microsoft Dynamics NAV Object Designer window. On the left, there is a navigation tree with 'Table' selected. The main area displays a table with the following columns: Type, ID, Name, Modified, and Version List. The table lists various Cue tables, each associated with a specific ID and name, along with their last modification date and version number.

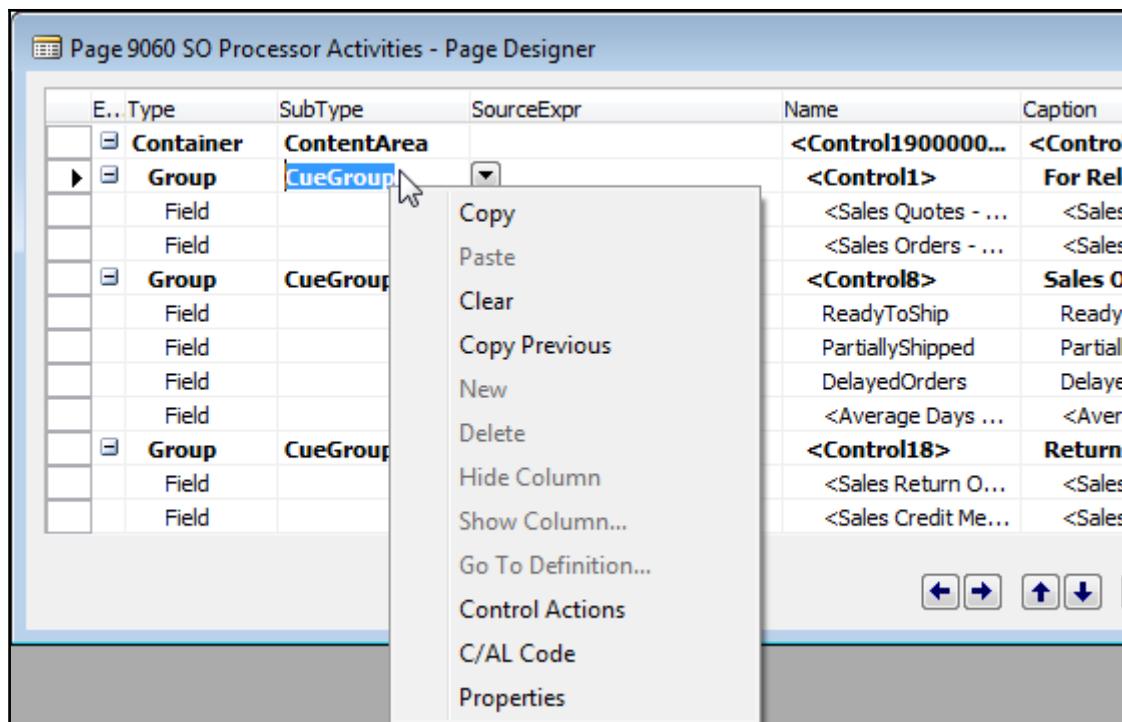
Type	ID	Name	Modified	Version List
Table	1313	Activities Cue		NAVW110.0
	5370	CRM Synch. Job Status Cue		NAVW19.00
	9042	Team Member Cue		NAVW110.0
	9050	Warehouse Basic Cue		NAVW110.0
	9051	Warehouse WMS Cue		NAVW110.0
	9052	Service Cue		NAVW17.10
	9053	Sales Cue		NAVW110.0
	9054	Finance Cue		NAVW110.0
	9055	Purchase Cue		NAVW18.00
	9056	Manufacturing Cue		NAVW110.0
	9057	Job Cue		NAVW110.0
	9058	Warehouse Worker WMS Cue		NAVW17.00
	9059	Administration Cue		NAVW110.0
	9060	SB Owner Cue		NAVW110.0
	9061	RapidStart Services Cue		NAVW17.00
	9063	Relationship Mgmt. Cue		NAVW110.0
	9069	O365 Sales Cue		NAVW110.0
	9070	Accounting Services Cue		NAVW19.00

## Cue Group Actions

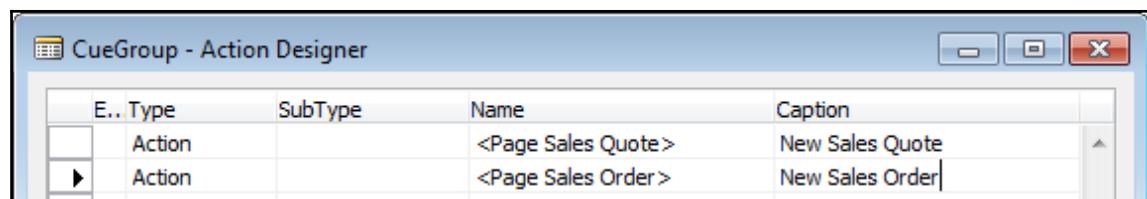
Another set of Role Center page components to analyze are the **Cue Group Actions**. While the Cues are the primary tasks that are presented to the user, the Cue Group Actions, displayed on the right, are a related secondary set of tasks:



Cue Group Actions are defined in the Role Center 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 **SubTypeCueGroup**. If we right-click on the **CueGroup** Control, one of the options available is **Control Actions**, as shown in the following screenshot:



When we choose **Control Actions**, the **Action Designer** form will be displayed, showing the two CueGroup actions in the **For Release** CueGroup in the SO Processor Role Center page. If we open the **Properties**, we will see that the "New" functionality is accomplished by setting the **RunPageMode** property to **Create**:



## System Part

Now that we have covered the components of the **Activities** portion of the Role Center page, let's take a look at the other components.

Returning to Page 9006 in the **Page Designer**, we examine the **Properties** of the System Part Control. This Page Part is the one that incorporates a view of the user's Notes data into the Role Center. Looking at this control's properties, we see a **PartType** of **System** and a **SystemPartID** of **MyNotes**, which displays as **My Notifications**:

The screenshot shows the 'Properties' window for the '<MyNotes>' part. The window has a title bar '<MyNotes> - Properties' and standard window controls (minimize, maximize, close). The main area is a grid of properties and their values:

Property	Value
ID	1901377608
Name	<MyNotes>
Visible	<TRUE>
Enabled	<TRUE>
Editable	<TRUE>
Caption	<MyNotes>
CaptionML	<Undefined>
ToolTip	<>
ToolTipML	<Undefined>
Description	<>
AccessByPermission	<Undefined>
ApplicationArea	<>
SubPageView	<Undefined>
SubPageLink	<Undefined>
ShowFilter	<Yes>
ProviderID	<0>
PartType	<b>System</b>
PagePartID	<Undefined>
SystemPartID	<b>MyNotes</b>
ChartPartID	<Undefined>
UpdatePropagation	<SubPart>

## Page Parts

Let's look at the second Group in Page 9006, the Group that defines the right-hand column appearing in the Role Center page. There are five Page Parts and a System Part defined:

E.. Type	SubType	Name	Caption
Container	RoleCenterArea	<Control1900000008>	<Control1900000008>
Group	Group	<Control1900724808>	<Control1900724808>
Part	Page	<SO Processor Activities>	<SO Processor Activities>
Part	Page	<Team Member Activities>	<Team Member Activities>
Part	Page	<My Customers>	<My Customers>
Part	Page	<Power BI Report Spinner Part>	<Power BI Report Spinner ...>
Group	Group	<Control1900724708>	<Control1900724708>
Part	Page	<Trailing Sales Orders Chart>	<Trailing Sales Orders Cha...>
Part	Page	<My Job Queue>	<My Job Queue>
Part	Page	<My Items>	<My Items>
Part	Page	<Report Inbox Part>	<Report Inbox Part>
Part	Page	<Connect Online>	<Connect Online>
	System	<MyNotes>	<MyNotes>

## Page Parts Not Visible

If we look at the display of the Role Center page generated by this layout again, we will see a chart (**Trailing Sales Orders**), followed by two ListParts (**My Items** and **Report InBox**) and, in turn, followed by the System Part (**My Notifications**). The two Page Parts, **My Job Queue** and **Connect Online**, do not appear. Those two Page Parts are defined by the **Visible** property equal to FALSE, which causes them not to display unless the Role Center Page is customized by the user (or an administrator or a developer) and the part added to the visible part list:

The screenshot shows the properties of a page part named <Connect Online>. The properties listed include:

Property	Value
ID	1903012608
Name	<Connect Online>
Visible	FALSE
Enabled	<TRUE>
Editable	<TRUE>
Caption	<Connect Online>
CaptionML	<Undefined>
ToolTip	<>
ToolTipML	<Undefined>
Description	<>
AccessByPermission	<Undefined>
ApplicationArea	<>
SubPageView	<Undefined>
SubPageLink	<Undefined>
ShowFilter	<Yes>
ProviderID	<0>
PartType	Page
PagePartID	Connect Online
SystemPartID	<None>
ChartPartID	<Undefined>
UpdatePropagation	<SubPart>

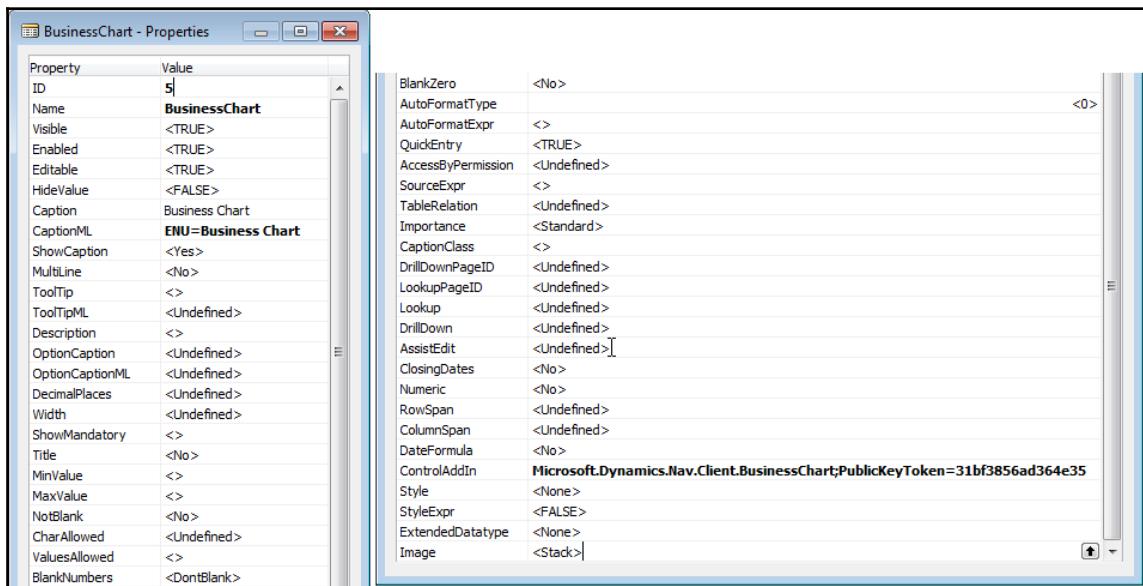
## Page Part Charts

The first Page Part in the second Group provides for a Chart to display using the Chart Control Add-in included in NAV 2017. The Chart Control Add-in is documented in the **Developer and IT Pro Help in Displaying Charts Using the Chart Control Add-in**. In the Page 9006, the Trailing Sales Orders Chart Page Part invokes Page 760 of the same name. Looking at Page 760 in the Page Designer, we see the following layout:

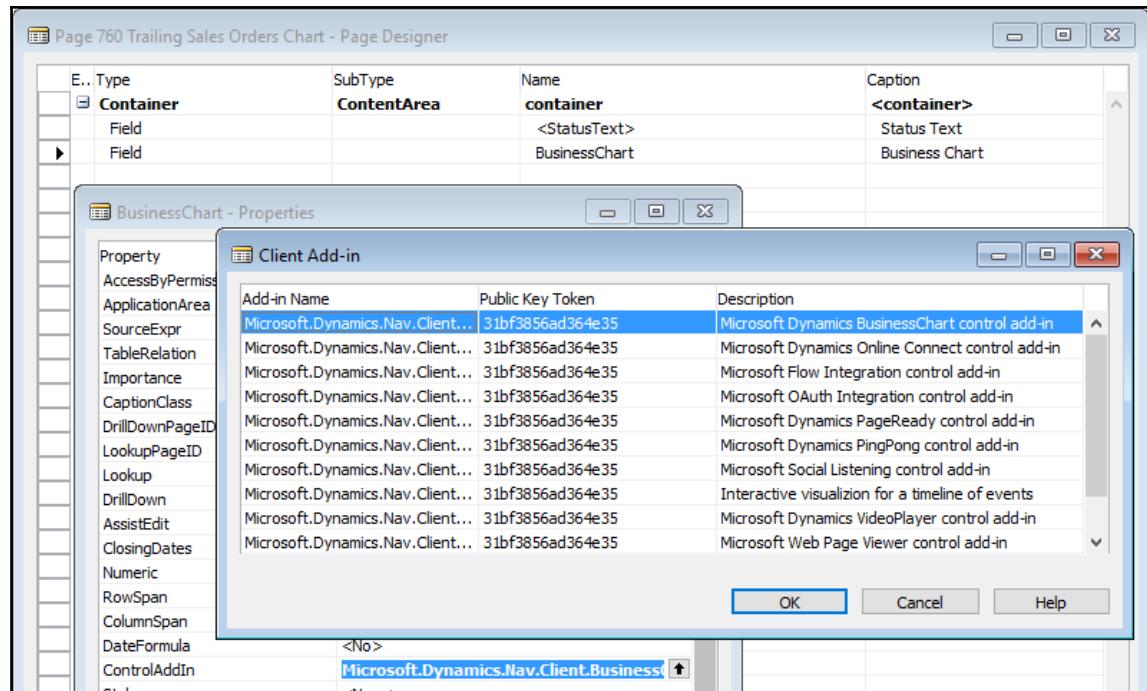
The screenshot shows the layout of Page 760. It contains a single row with two columns. The left column contains a Container page part with two Fields. The right column contains a ContentArea page part with three fields: StatusText, BusinessChart, and Business Chart.

Exp...	Type	SubType	SourceExpr	Name	Caption
	Container	ContentArea		container	<container>
	Field		StatusText	<StatusText>	Status Text
▶	Field			BusinessChart	Business Chart

The properties of the Field named **Business Chart** look like the following screenshot:



Note that the **ControlAddIn** property contains the necessary information to access the Chart Control Add-in. This property provides access to the screen, which is shown in the following screenshot, where the Client Add-ins that are available for use in our NAV system are listed. An Add-in is a Microsoft .NET Framework assembly (a module external to NAV, but registered with the system) that lets us add custom functionality to a NAV Windows Client. The **Client Add-in** screen shown here displays after clicking on the lookup arrow at the right end of the **ControlAddIn** property:



## Page Parts for User Data

Three of the Page Parts in Role Center Page 9006 provide data that is specific to the individual User. They track "My" data, information important to the user who is logged in. When we design any one of the pages, we can open the page properties to find out what table the page is tied to. Then, viewing any of those tables in the Table Designer, we will see that a highly ranked field is User ID. An example is the **My Item** table, which is shown here:

E..	Field No.	Field Name	Data Type	Length	Description
▶	1	User ID	Code	50	
▶	2	Item No.	Code	20	
▶	3	Description	Text	50	
▶	4	Unit Price	Decimal		
▶	5	Inventory	Decimal		

The User ID allows the data to be filtered to present user specific information to each user. In some cases, this data can be updated directly in the Role Center Page Part, for example, in My Customers and My Items. In other cases, such as My Job Queue, the data is updated elsewhere and is only viewed in the Role Center Page Part. If our users need to track other information in a similar manner, such as My Service Contracts, we can readily plagiarize the approach used in the standard Page Parts.

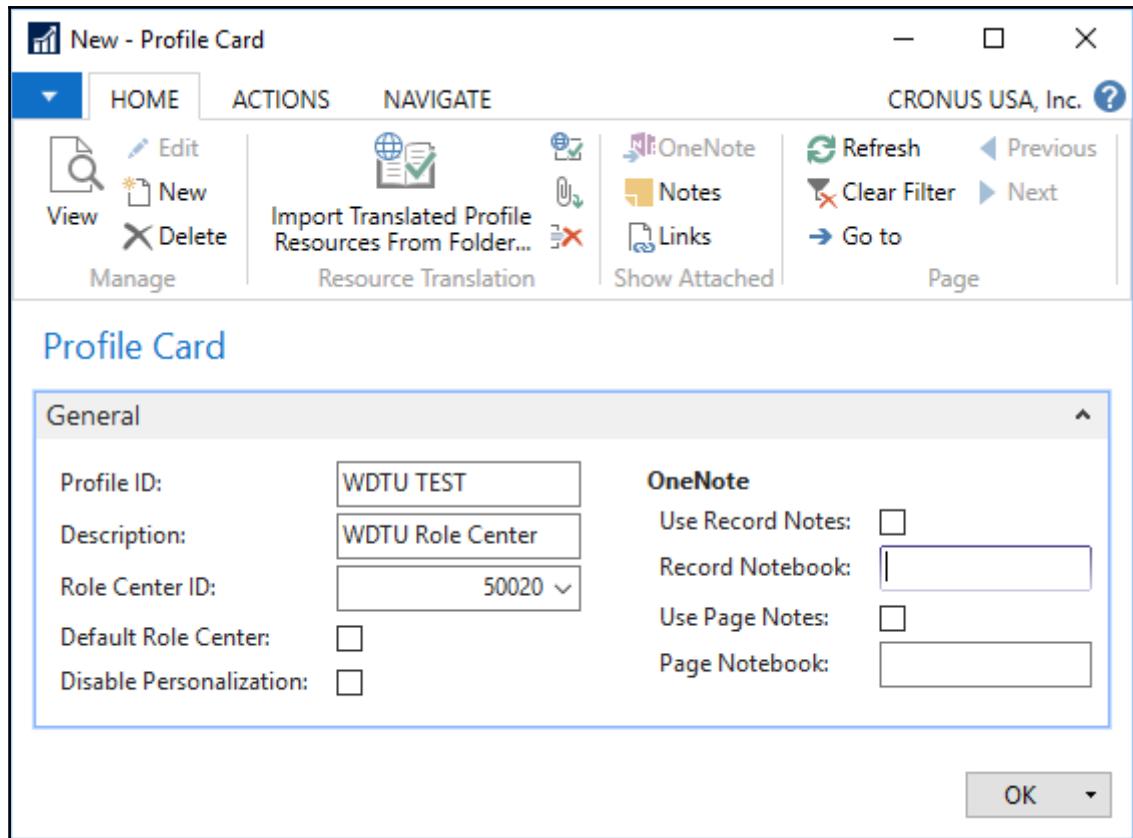
## Navigation Pane and Action Menus

The last major Role Center Page components we'll review are the Navigation Pane and the Action Ribbon. Even though there are two major parts of the Role Center Page that provide access to action choices, they both are defined in the **Action Designer** section of the **Page Designer**.

The display of Action Controls in a Role Center page is dependent on a combination of the definition of the controls in the Action Designer, certain properties of the page, and configuration/personalization of the page. Many of the default Role Centers provided with the product are configured as examples of possibilities of what can be done. Even if one of the default Role Centers seems to fit our customer's requirements exactly, we should create a copy of that Role Center page as another page object and reconfigure it. This way, we can document how that page was set up and make any necessary tweaks.

We'll start with Role Center Page 9006 because it is used as the default Role Center and is used in so many other examples. Copy Page 9006 into Page **50020 - WDTU Role Center** using the sequence **Object Designer** | **Page** | **Design** | **File** | **Save As...**, and with a new page object ID of 50020 and object Name of **WDTU Role Center**.

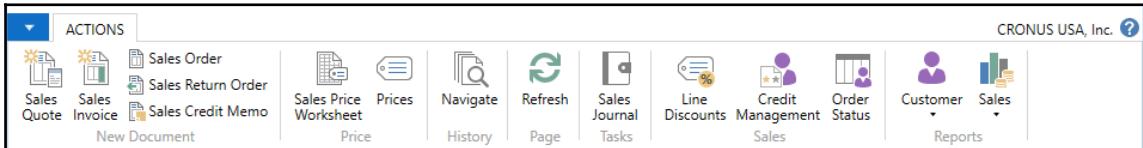
Once we have the new page saved, in order to use this page as a Role Center, we must create a Profile for the page. This is done within the Role Tailored Client and is typically a System Administrator task. Invoke the RTC and click on the **Departments** menu button in the Navigator Pane. Then click on **Administrator: Application Setup** | **Role Tailored Client** | **Lists: Profiles**. Click on the **New** icon and create a new profile like this one:



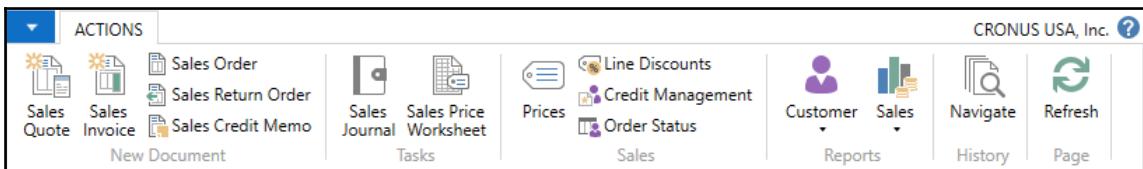
For the purpose of easy access to this Role Center for testing, we could also checkmark the **Default Role Center** box. Then, when we invoke the RTC, our test Role Center will be the one that displays (if no other profile is assigned to this user). Another approach to testing is to assign our User ID to use this profile.

When we are doing development work on a Role Center, we 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, we must launch it as the assigned Role Center for the active user.

A major area where action choices are presented in a Role Center (and also in other page types) is in the ribbon. The ribbon for the standard Page 9006 - Order Processor Role Center, as delivered from Microsoft, looks like the following screenshot:



After we have created our Role Center copy, the ribbon for Page 50020 - Order Processor RC WDTU looks like this (it may differ somewhat depending on the localization version you are using):



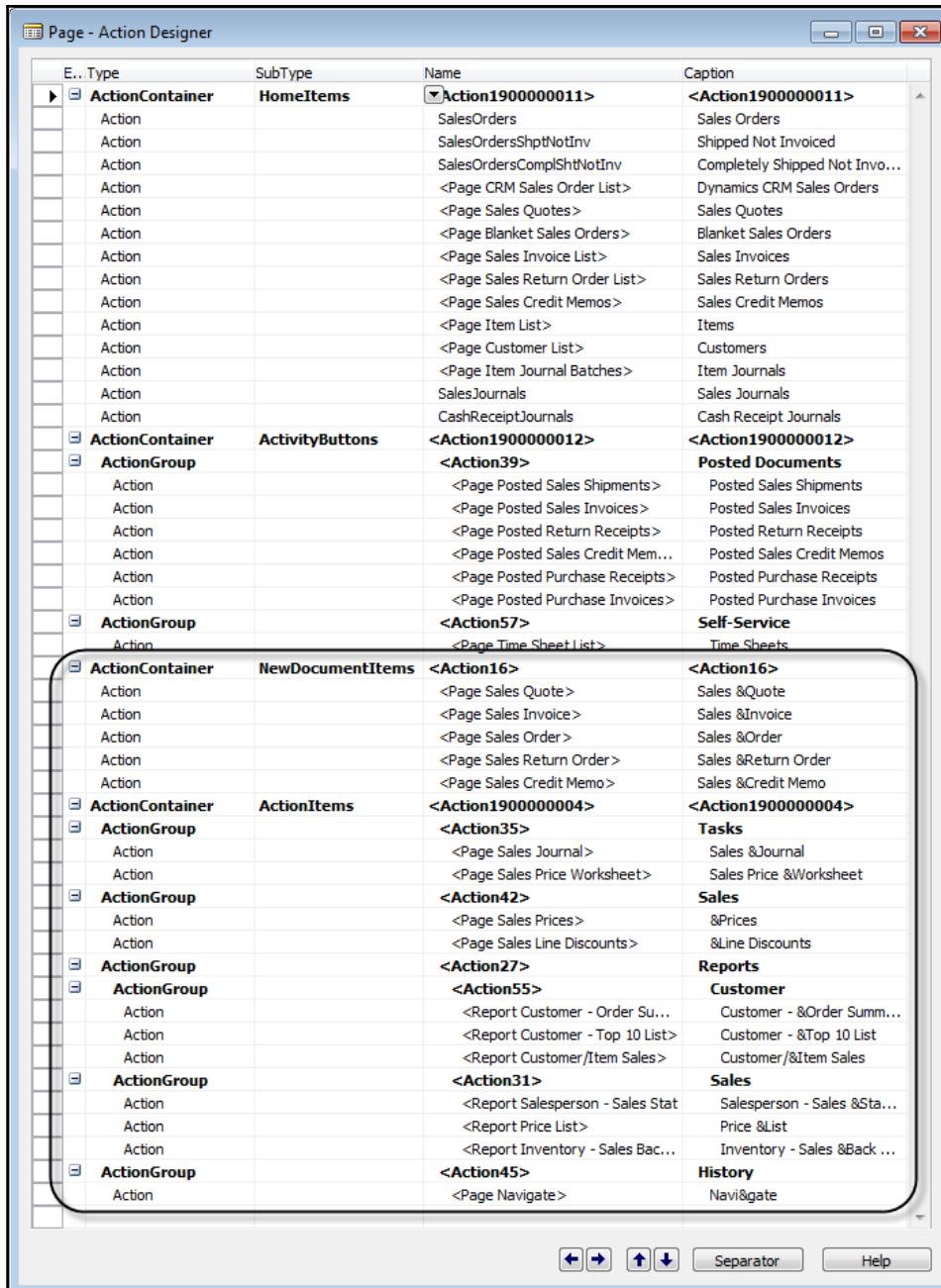
When we compare the available actions in the two ribbon images, we see most of the same actions, but displayed somewhat differently.

If we made the same kind of analysis of some of the other default Role Centers, we would find similar results. When the page is copied to another object number the appearance of the ribbon changes, losing detail. As it turns out, many of the default Role Centers have been manually configured by Microsoft and delivered with the NAV software distribution as part of the effort to show examples of Role Center ribbon design. Thus, we should start with a fresh copy of our Role Centers when designing for our customer so that we know what tailoring has been done and we are in control of the design.

## Action Designer

The actions for a page are defined and maintained in the **Action Designer**. The **Action Designer** is accessed from within the **Page Designer**. Open our new Page 50020 - WDTU Role Center in the **Page Designer**, then either press **Ctrl + Alt + F4** or **View | Page Actions** to open the **Action Designer** to view the current set of actions defined for this page.

For our newly created Page 50020, cloned from Page 9006, the Action Designer contents look like the following screenshot:



The actions enclosed in the rectangle are the ones that are assigned to the ribbon. Whether or not they actually are displayed, how they are displayed, and where they are displayed, are all controlled by a combination of the following factors:

- The structure of the controls within the action list
- The properties of the individual actions
- The customizations/personalizations that have been applied by a developer, administrator, or the user

The first column of each action control is the **Type**. In hierarchical order, the action control entries can be ActionContainer, Action Group, Action, or Separator. The specific **SubType** of each **ActionContainer** entry determines the area, Ribbon, or Navigation Pane, in which the subordinate groups of actions will appear:

The screenshot shows the 'Page - Action Designer' window. It displays a hierarchical list of action controls. The columns are labeled 'E...', 'Type', 'SubType', and 'Name'. A mouse cursor is hovering over the 'Name' column of the first item under 'ActionContainer'.

E...	Type	SubType	Name
▶	ActionContainer	ActivityButtons	Action1900000012>
▶	ActionGroup	NewDocumentItems	Action39>
	Action	ActionItems	<Page Posted Sales Sh
	Action	RelatedInformation	<Page Posted Sales In
	Action	Reports	<Page Posted Return P
	Action	HomeItems	<Page Posted Sales Cr
	Action	ActivityButtons	<Page Posted Purchas
	Action		<Page Posted Purchas
▶	ActionGroup		<Action57>

If the **SubType** is HomeItems or ActivityButtons (Page Control SubTypes that can only be used in Role Center pages), the indented subordinate actions will appear in the Navigation Pane. All the other SubTypes (NewDocumentItems, ActionItems, and Reports) will cause their subordinate actions to appear in the Role Center Ribbon. These three SubTypes are not limited to use in Role Center pages. The **SubType** RelatedInformation is not intended for use in Role Center pages, but only in other page types.

An ActionGroup control provides a grouping of actions that will appear as a category in the ribbon. For actions to appear within the category on the ribbon, those Action controls must follow the ActionGroup and be indented. If an ActionGroup is indented within a parent ActionGroup, it will generate a drop-down list of actions.

The other type of action control is the Separator. In the NAV 2017 Action Ribbon, the separator controls don't appear to do anything.

If we compare the control entries in the preceding Action Designer screenshot to the action icons that display in the screenshot of the unmodified Page 50020 ribbon, we will see the following:

- The action control entries of the `NewDocumentItems` and `ActionItems` `ActionContainers` appear on the **Actions** tab of the ribbon. The `ActionItems` Category is intended for task related functions and which `NewDocumentItems` Category is intended for those actions that cause a new document to be opened.
- All the control entries in the `NewDocumentItemsActionContainer` appear in the New Document Items Category in the Action Ribbon.
- The control entries in the `Tasks` `ActionGroup` and the `Sales` `ActionGroup` appear respectively in the **Tasks** and **Sales** Categories of the ribbon.
- One action, **Refresh**, is a default action that is automatically generated and assigned to the **Page** Category.
- All the control entries in the `Reports` captioned `ActionGroup` are in the **Reports** Category of the Action Ribbon.

## Creating a WDTU Role Center Ribbon

If we were creating a role center to be used in a real production environment, we would likely be defining a new Activities Page, new Cues, a new or modified Cue table, new FactBoxes, and so on. However, since our primary purpose here is learning, we'll take the shortcut of piggybacking on the existing role center and simply add our WDTU actions to the foundation of that role-existing center.

There are several steps to be taken to define our WDTU Role Center Ribbon using the Developer tools. The steps we need to do for our WDTU actions are as follows:

1. Define one or more new ribbon categories for the WDTU actions.
2. Create the WDTU action controls in the Action Designer.
3. Assign the WDTU action controls to the appropriate ribbon categories.
4. Finalize any look and feel items.

Because some of the original Order Processor Role Center ribbon layout disappeared when we cloned Page 9006 to Page 50020, we will also want to recreate that layout. For this part of our ribbon definition effort, because we want to learn more ways to accomplish implementation goals, we will use the Configuration/Personalization tools.

The steps needed to replicate the Page 9006 ribbon layout are as follows:

1. Define the needed ribbon categories.
2. Assign the action controls to the appropriate categories.
3. Finalize any look and feel items.

The normal sequence of defining an Action Ribbon is to complete the work that utilizes development tools and then proceed with the work that can be done by an implementer or system administrator (or even an authorized user). So, we will work on the WDTU portion of the Action Ribbon first, and then follow with the work of replicating the original layout.

Let's add the following functions to the WDTU portion of the Action Ribbon:

- Radio Show List page
- Playlist page
- Radio Show Types page
- Playlist Item Rates page
- Item List (filtered for Playlist Items) page
- Record Labels page

To add an action, access the **Page - Action Designer** screen, go the bottom of the list of existing actions, add a line of **TypeAction**, then open the **Properties** screen for that line. Click on the **RunObject** property **Value** field and select the object to be run, in this case, it is Page Radio Show List. Next, we will define the target ribbon menu caption, icon display size and icon to represent this menu option.

Let's do the same thing for one of our WDTU Maintenance actions, the **Radio Show Types** page. We'll set the **Caption** value to Radio Show Types and the **Image** value to Entry.

## Action Groups / Ribbon Categories

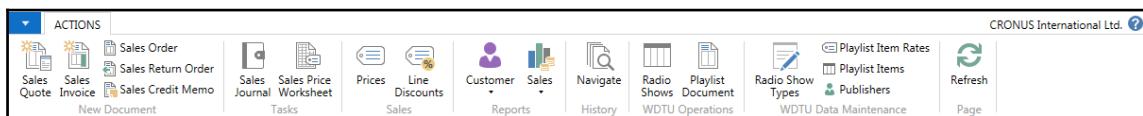
Ribbon Categories are defined and assigning actions to them is through the use of **ActionGroups**. We will make the proper **ActionGroup** entries in **Action Designer** with the appropriate **Action** entries indented under them as shown in the following. We should also choose appropriate images. We'll put the first two items in a category named WDTU Operations and the other four items in a category named WDTU Data Maintenance, as shown in the following screenshot:

**Page - Action Designer**

E.. Type	SubType	Name	Caption
└ ActionContainer	ActionItems	<Action1900000004>	<Action1900000004>
└ ActionGroup		<Action35>	Tasks
Action		<Page Sales Journal>	Sales &Journal
Action		<Page Sales Price Worksheet>	Sales Price &Worksheet
└ ActionGroup		<Action42>	Sales
Action		<Page Sales Prices>	&Prices
Action		<Page Sales Line Discounts>	&Line Discounts
└ ActionGroup		<Action27>	Reports
└ ActionGroup		<Action55>	Customer
Action		<Report Customer - Order...	Customer - &Order Summ...
Action		<Report Customer - Top 1...	Customer - &Top 10 List
Action		<Report Customer/Item S...	Customer/&Item Sales
└ ActionGroup		<Action31>	Sales
Action		<Report Salesperson - Sal...	Salesperson - Sales &Sta...
Action		<Report Price List>	Price &List
Action		<Report Inventory - Sales...	Inventory - Sales &Back ...
└ ActionGroup		<Action45>	History
Action		<Page Navigate>	Navi&gate
└ ActionGroup		<Action17>	WDTU Operations
Action		<Page Radio Show List>	Radio Shows
Action		<Page Playlist Document>	Playlist Document
└ ActionGroup		<Action22>	WDTU Data Maintenance
Action		<Page Radio Show Types>	Radio Show Types
Action		<Page Playlist Item Rates>	Playlist Item Rates
Action		<Page Item List>	Playlist Items
Action		<Page Publishers>	Publishers

**Actions**

The resulting Ribbon looks like the following screenshot:



## Configuration/Personalization

The procedures and interface tools we use to do Configuration are also used by users to do Personalization. Both terms refer to revising the display contents and format of a Role Center as it appears to one or more users. As it says in the Help, *Walkthrough: Configuring the Order Processor Role Center*:



The difference between configuration and personalization is Configuring a Role Center changes the user interface for all users who have that profile, whereas Personalizing a Role Center only changes the user interface for a single user.

We could replace the WDTU Category assignments we just made using ActionGroups by defining Categories and assigning actions using Configuration. The result would look exactly the same to our users. However, instead, let's use Configuration to quickly restore the layout of the actions that were in Role Center Page 9006. We can run page 9006 to see what that layout is (or reference the earlier snapshot of the Page 9006 ribbon).

The following are a couple of important points:

- Configuration is tied to a specific Profile. Other Profiles using the same Role Center page do not see the same Configuration layout.
- Configuration can only be done by the Owner of a Profile. When we created our WDTU Test Profile, we did not assign an owner, so that will have to be done now.
- Profile setup can be accessed in the RTC in **Departments** | **Administration** | **Application Setup** | **Role Tailored Client** | **Profiles**. This can also be found by entering the word Profile in the RTC Search box. The Owner ID for a Profile can be updated there by an Administrator with sufficient Permissions.

A Role Center ribbon is configured by opening the Role Tailored Client in Configuration mode with the focus on the Profile we want to configure. Personalization doesn't require this step. This is done from the DOS Command prompt using a command line essentially similar to the following lines of code:

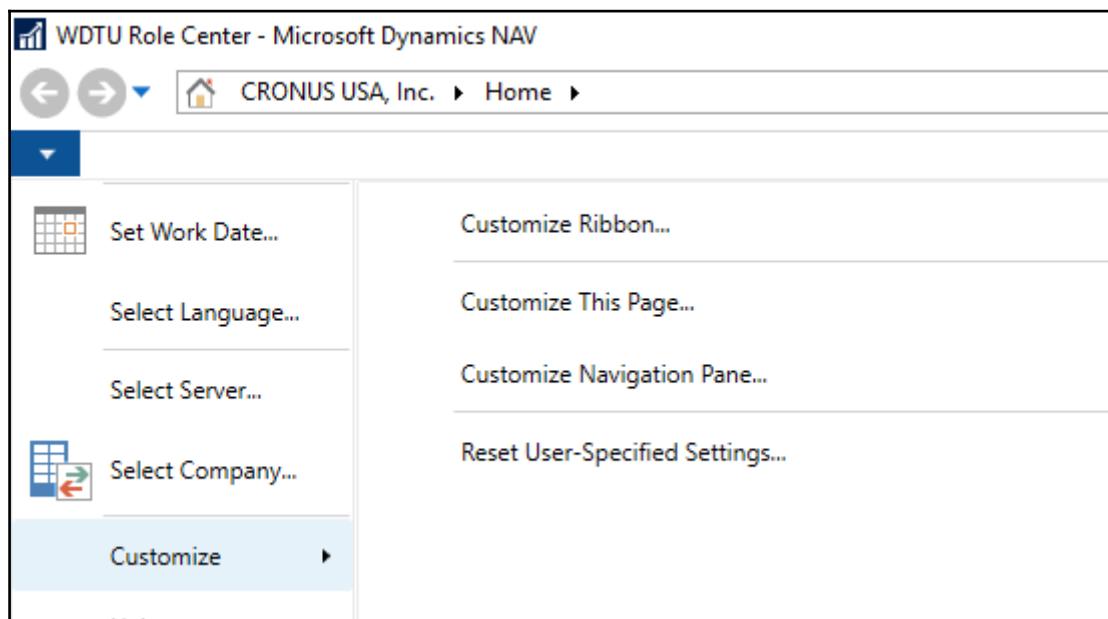
```
"C:\Program Files (x86)\Microsoft Dynamics NAV100  
RoleTailored ClientMicrosoft.Dynamics.Nav.Client.exe"  
-configure -profile:"WDTU Test"
```

For additional information, refer to the Developer Help (in MSDN) *How to: Open the Microsoft Dynamics NAV Windows Client in Configuration Mode*.

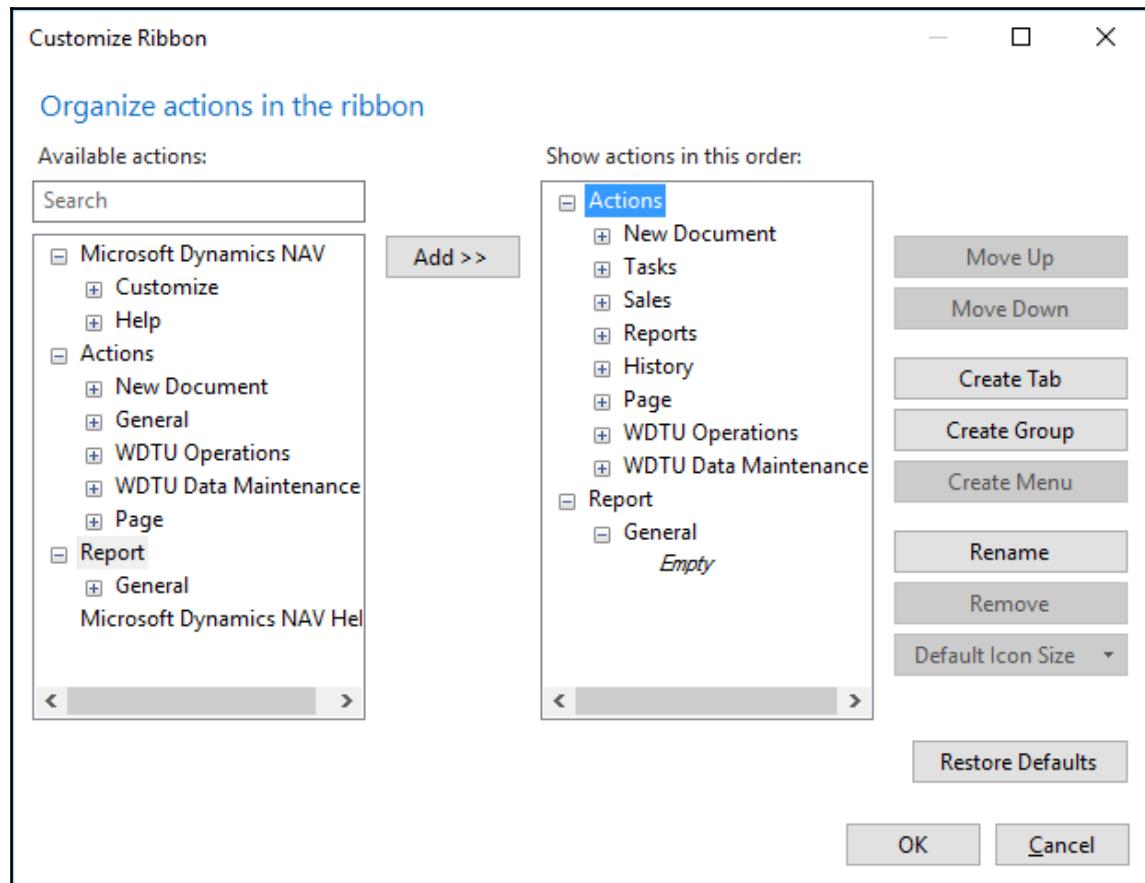


In NAV 2017, there is an option field called **UI Elements Removal** in the NAV Server Administration tool. Depending on the setting, the system administrator can limit the accessible User Interface (UI) elements to be either only those on objects available in the license or only those on objects to which the user has access permission. For more information on this feature, see the Help, **How to Specify When UI Elements are Removed**.

Once the Role Center displays, click on the arrowhead to the right of the Microsoft Dynamics icon (1 in the following screenshot), then on **Customize** option (2 in the following screenshot), followed by the **Customize Ribbon** option (3 in the following screenshot):

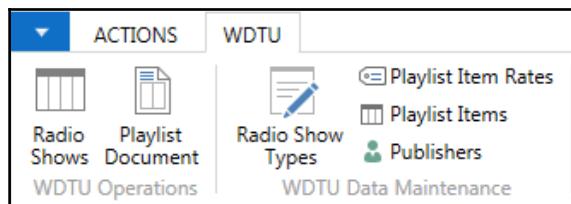


That will take us to the following screen:



As we can see, using the Customize Ribbon screen, we can Create Groups (referred to as Categories elsewhere), add new actions to those available on the ribbon, remove actions (that is, make them not visible), reorganize ribbon entries, even create new tabs, or rename existing items. In summary, everything that we've done so far to customize the Role Center Ribbon can be done through this screen. The big difference is that **Customization** in Personalization mode is specific to a single Profile, while the other modifications will apply to all profiles.

As part of our Personalizing, we might use the **Create Tab**, **Move Up**, and **Move Down** options to rearrange the actions on the ribbon, moving the WDTU actions to their own ribbon tab. When we are done configuring the ribbon for the WDTU Test Profile, the WDTU portion should look like the following screenshot:



## Navigation Pane

The Navigation Pane is the menu list that makes up the leftmost column on a Role Center. A Navigation Pane can have two or more buttons. The required two buttons are Home and Departments.

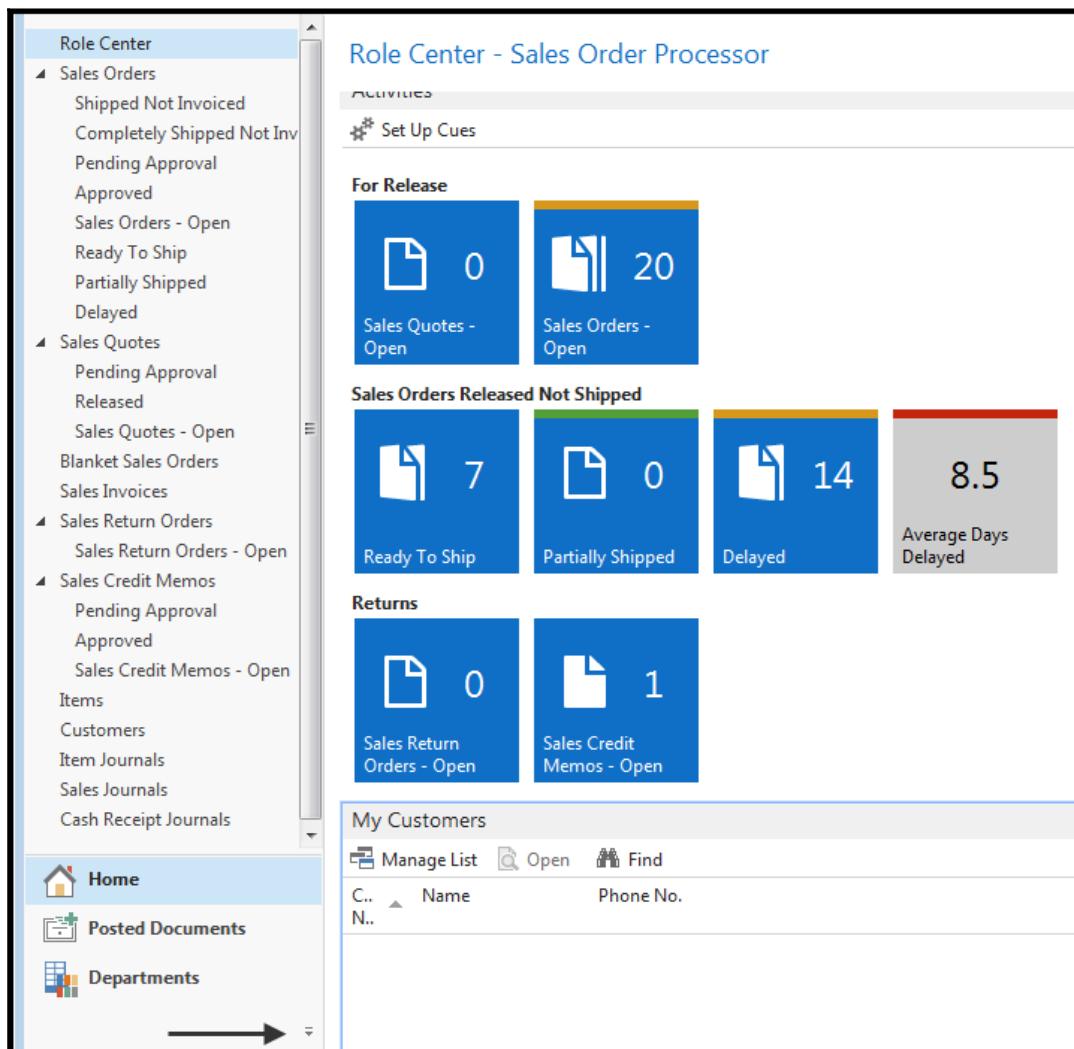
### Navigation Home Button

The basic contents of the Home button for a Role Center are defined in the Action Designer in **ActionContainer** of **SubType** as **HomeItems**:

The screenshot shows the 'Action Designer' window with the title 'Page - Action Designer'. It displays a table with four columns: E.. Type, SubType, Name, and Caption. The 'E.. Type' column contains mostly 'Action' entries, except for one 'ActionContainer' entry. The 'SubType' column shows 'HomeItems' for the first row and 'Action' for the subsequent rows. The 'Name' and 'Caption' columns list various CRM actions such as Sales Orders, Sales Quotes, etc. The 'ActionContainer' row has a single child action named 'SalesOrders'.

E.. Type	SubType	Name	Caption
ActionContainer	HomeItems	<Action1900000011>	<Action1900000011>
Action	Action	SalesOrders	Sales Orders
Action	Action	SalesOrdersShptNotInv	Shipped Not Invoiced
Action	Action	SalesOrdersComplShptNotInv	Completely Shipped Not Invo...
Action	Action	<Page CRM Sales Order List>	Dynamics CRM Sales Orders
Action	Action	<Page Sales Quotes>	Sales Quotes
Action	Action	<Page Blanket Sales Orders>	Blanket Sales Orders
Action	Action	<Page Sales Invoice List>	Sales Invoices
Action	Action	<Page Sales Return Order List>	Sales Return Orders
Action	Action	<Page Sales Credit Memos>	Sales Credit Memos
Action	Action	<Page Item List>	Items
Action	Action	<Page Customer List>	Customers
Action	Action	<Page Item Journal Batches>	Item Journals
Action	Action	SalesJournals	Sales Journals
Action	Action	CashReceiptJournals	Cash Receipt Journals

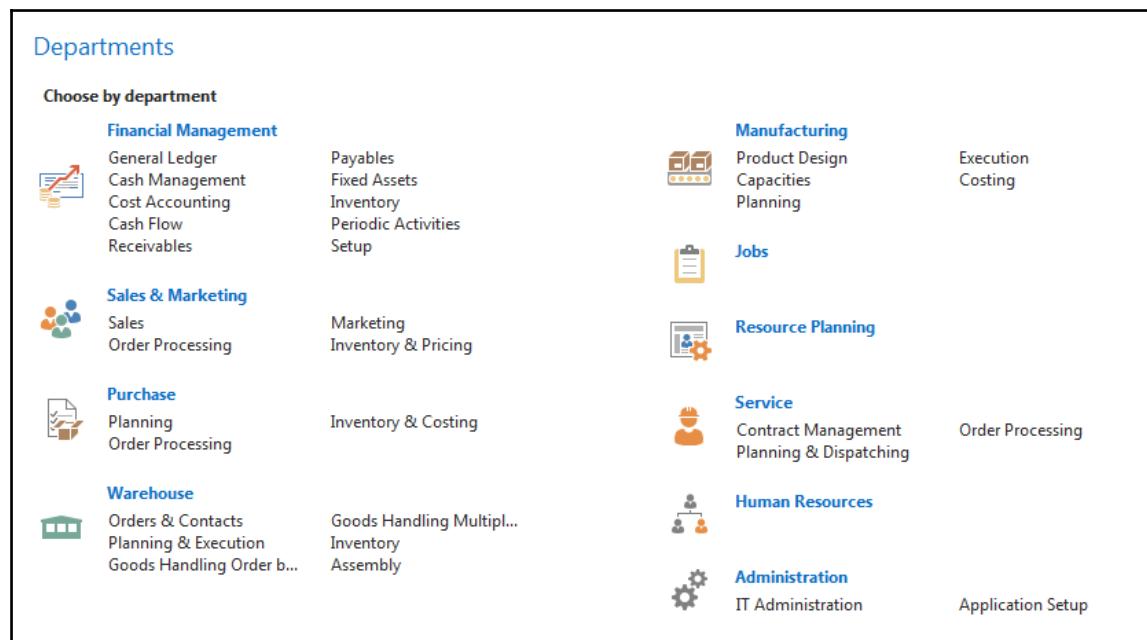
In addition to the action controls defined in the Action Designer, the Navigation Pane Home menu list includes all the Cue entries that appear in the Activities Pane of the Role Center. We can see the combined sets of action options in the following screenshot. Note that there are a number of indented (nested) options within groups such as **Sales Orders**, **Sales Quotes**, and **Sales Credit Memos**. These groups are set up using the same type of Configuration tools that we just used for the ribbon:



We can access the Configuration/Personalization tools either through the arrow next to the drop-down arrowhead to the left of the ribbon (as shown earlier) or through the very tiny icon at the bottom right of the Navigation Pane (highlighted by an arrow in the preceding screenshot).

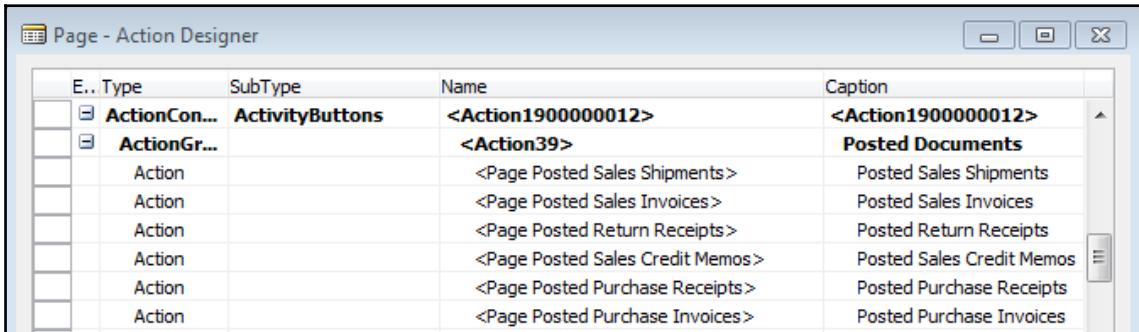
## Navigation Departments Button

The other required Navigation Pane button, the Departments button, has its menu entries generated based on the contents of the MenuSuite object. If we click on the **Departments** button, a screen like the following screenshot will be displayed:



## Other Navigation Buttons

Other Navigation Pane buttons can be defined and populated by means of ActionGroup entries with the ActionContainerActivityButtons in the Action Designer, as shown in the following screenshot:



The screenshot shows the 'Action Designer' window with a title bar 'Page - Action Designer'. The main area is a grid table with four columns: 'E.. Type', 'SubType', 'Name', and 'Caption'. The 'SubType' column contains 'ActivityButtons' for the first two rows and 'Action' for the remaining six rows. The 'Name' column lists various action names, and the 'Caption' column provides the corresponding button labels.

E.. Type	SubType	Name	Caption
	ActionCon...	<Action1900000012>	<Action1900000012>
	ActionGr...	<Action39>	<b>Posted Documents</b>
	Action	<Page Posted Sales Shipments>	Posted Sales Shipments
	Action	<Page Posted Sales Invoices>	Posted Sales Invoices
	Action	<Page Posted Return Receipts>	Posted Return Receipts
	Action	<Page Posted Sales Credit Memos>	Posted Sales Credit Memos
	Action	<Page Posted Purchase Receipts>	Posted Purchase Receipts
	Action	<Page Posted Purchase Invoices>	Posted Purchase Invoices

Navigation Pane buttons can also be added, renamed, repopulated, and made not visible through the **Customize Navigation Pane...** option in the page **Customization** submenu.

## XMLports

**XML (eXtensible Markup Language)**, a structured text format developed to describe data to be shared by dissimilar systems. XML has become a standard for communications between systems. To make handling XML-formatted data simpler and more error resistant, NAV provides XMLports, a data import/export object. In addition to processing XML-formatted data, XMLports can also handle a wide variety of other text file formats, including CSV files, generic flat files, and so on. XML-formatted data is text based, with each piece of information structured in one of two basic formats: Elements or Attributes. An Element is the overall logical unit of information, while an Attribute is a property of an Element. They are formatted as follows:

- <Tag>elementvalue</Tag> (an **Element** format)
- <Tag AttribName="attribute datavalue"> (an **Attribute** format))



Elements can be nested, but must not overlap. Element and Attribute names are case-sensitive. Names cannot start with a number, punctuation character, or the letters xml (upper or lower cased) and cannot contain spaces.

An Attribute value must always be enclosed in single or double quotation marks. Some references suggest that Elements should be used for data and Attributes for metadata. Complex data structures are built up of combinations of these two formats.

For example, let's consider the following code:

```
<Table Name='Sample XML format'>
  <Record>
    <DataItem1>12345</DataItem1>
    <DataItem2>23456</DataItem2>
  </Record>
  <Record>
    <DataItem1>987</DataItem1>
  </Record>
  <Record>
    <DataItem1>22233</DataItem1>
    <DataItem2>7766</DataItem2>
  </Record>
</Table>
```

In this instance, we have a set of data identified as a `Table` with an attribute of `Name` equal to '`SampleXMLformat`', which contains three `Records`, each `Record` containing data in one or two fields named `DataItem1` and `DataItem2`. The data is in a clearly structured text format, so it can be read and processed by any system 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 the sharing and common interpretation of the format between the transmitter and recipient.

XML is a standard format in the sense that the data structure options are clearly defined. It is very flexible, in the sense that the identifying tag names (in `< >` brackets) and the related data structures are totally open ended in how they can be defined and processed. 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.

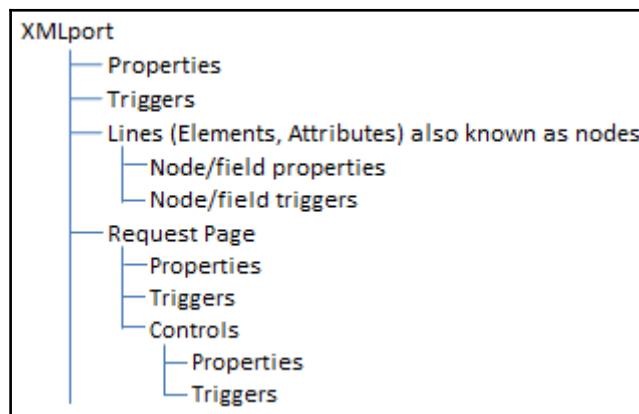
XML data structures can be as simple as a flat file consisting of a set of identically formatted records or as complex as a sales order structure with headers containing a variety of data items, combined with associated detail lines containing their own assortments of data items. An XML data structure can be as complicated as the application requires.

XML standards are maintained by the W3C, whose website is at [www.w3.org](http://www.w3.org). There are many other useful websites for basic XML information.

## 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. This allows the internal logic to be simpler. XMLports utilize a process of looping through and processing data similar to that of Report objects.

The components of XMLports are as follows:



## XMLport properties

XMLport properties are shown in the following screenshot of the Properties of the XMLport object, 9170:

XMLport - Properties	
Property	Value
ID	9170
Name	<b>Profile Import/Export</b>
Caption	Profile Import/Export
CaptionML	<b>ENU=Profile Import/Export</b>
Direction	<Both>
DefaultFieldsValidation	<Yes>
Encoding	<UTF-16>
XMLVersionNo	<1.0>
Format/Evaluate	<b>XML Format/Evaluate</b>
UseDefaultNamespace	<No>
DefaultNamespace	<urn:microsoft-dynamics-nav/xmlports/x91...>
Namespaces	<Undefined>
InlineSchema	<No>
UseLax	<No>
Format	<Xml>
FileName	<Undefined>
UseRequestPage	<Yes>
TransactionType	<UpdateNoLocks>
Permissions	<Undefined>
PreserveWhiteSpace	<No>

Descriptions of the individual properties are as follows:

- **ID:** This is the unique XMLport object number.
- **Name:** This is the name by which this XMLport is referred to within C/AL code.
- **Caption:** This is the name that is displayed for the 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 will set the default for individual field **FieldValidate** properties to **Yes**.

- **Encoding (or TextEncoding):** This defines the character encoding option to be used, **UTF-8**(ASCII compatible) or **UTF-16**(not ASCII compatible) or ISO-8859-2 (for certain European languages written in Latin characters). **UTF-16** is the default. This is inserted into the heading of the XML document.
- The **TextEncoding** option is only available if the **Format** property is Fixed Text or Variable Text. In this case, a character coding option of **MS-DOS** is available and is the default.
- **XMLVersionNo:** This defines to which version of XML the document conforms, **Version1.0** or **1.1**. The default is **Version1.0**. This is inserted into the heading of the XML document.
- **Format/Evaluate:** This can be **C/SIDEFormat/Evaluate** (the default) or **XMLFormat 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. Very limited additional information on this is available in the online **Help** in Temporary Property (XMLports).
- **UseDefaultNamespace** and **DefaultNamespace:** These properties are provided to support compatibility with other systems that require the XML document to be in a specific namespace, such as use of a web service as a reference within Visual Studio. **UseDefaultNamespace** defaults to **No**. A default namespace in the form of **URN (UniformResourceName** or, in this case, a **Namespace Identifier**) concluding with the object number of the XMLport is supplied for the **DefaultNamespace** property. This property is only active if the **Format** property is XML.
- **Namespaces:** This property takes you to a new screen where you can set up multiple namespaces for the XMLPort including a Prefix.
- **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. This property is only active if the **Format** property is XML.
- **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. This property is only active if the **Format** property is XML.

- **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 **VariableText** 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 **FixedText** means that 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. If enabled, these four fields can also be changed programmatically from within C/AL code.
- **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, and 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**). See the Help for this property for more information.
- **RecordSeparator:** This applies to **VariableText** or **FixedText** format external files only. 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. See the Help for this property for more information.
- **TableSeparator:** This applies to **VariableText** or **FixedText** format external files only. 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>>. See the Help for this property for more information.

- **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 has only 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, we can refer to the 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

XMLport has a very limited set of triggers, which are as follows:

- **Documentation()** is for documentation comments
- **OnInitXMLport()** is executed once when the XMLport is loaded before the table views and filters have been set
- **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.

XMLports should not be run directly from a Navigation Pane action command (due to conflicts with NAV UX standards), but can be run either from ribbon actions on a Role Center or other page or by means of an object containing the necessary C/AL code. When running from another object (as opposed to running from an action menu entry), C/AL code calls the XMLport to stream data either to or from an appropriately formatted file (XML document or other text format). This calling code is typically written in a Codeunit, but it can be placed in any object that can contain C/AL code.

The following example code executes an exporting XMLport and saves the resulting file from the NAV service tier to the client machine:

```
XMLfile.CREATE(TemporaryPath + 'RadioShowExport.xml');
XMLfile.CREATEOUTSTREAM(OutStreamObj);
XMLPORT.EXPORT(50000, OutStreamObj);
FromFileName := XMLfile.NAME;
ToFileName := 'RadioShowExport.xml';
XMLfile.CLOSE;

//Need to call DOWNLOAD to move the xml file
//from the service tier to the client machine
DOWNLOAD(FromFileName, 'Downloading File...', 'C:', 'Xml
file (*.xml) | *.xml', ToFileName);

//Make sure to clean up the temporary file from the
//service tier
ERASE(FromFileName);
```

Two text variables (the `from` filename and the `to` filename), a file variable, and an `OutStreamObj` variable are required to support the preceding code. The data variables are defined as shown in the following screenshot:

The screenshot shows the 'XMLPortDriver - C/AL Locals' window. It has tabs for Parameters, Return Value, Variables, and Text Constants, with 'Variables' selected. A table lists four variables: XMLFile (Subtype File), OutStreamObj (Subtype OutStream), FromFileName (Subtype Text, Length 80), and ToFileName (Subtype Variant). The 'FromFileName' row has an asterisk (\*) next to it.

Name	Subtype	DataType	Length
XMLFile		File	
OutStreamObj		OutStream	
FromFileName		Text	80
* ToFileName		Variant	

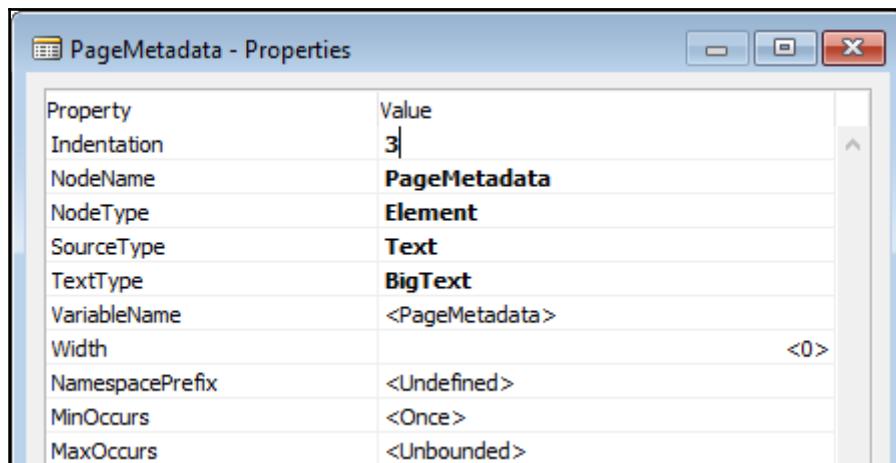
## The XMLport line properties

Different XMLport line properties are active on a line depending on the value of the **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 after the following 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, so, often, we will want to define the Level 0 line to be a simple text element line. This allows the definition of multiple Tables at Indentation level 1.
- **NodeName:** This defines the Node Name that will be used in the XML document to identify the data associated with this position in the XML document. No spaces are allowed in a **NodeName**; we can use underscores, dashes, and periods, but no 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 the 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**, **NamespacePrefix**, **MinOccurs**, and **MaxOccurs** properties are discussed later in this chapter.

## SourceType as Table

The following screenshot shows the properties for **SourceType as Table**:

The screenshot shows a Windows-style dialog box titled "Profile - Properties". It contains a table with two columns: "Property" and "Value". The properties listed are: Indentation (Value: 1), NodeName (Value: Profile), NodeType (Value: Element), SourceType (Value: Table), SourceTable (Value: Profile), VariableName (Value: <Profile>), SourceTableView (Value: <Undefined>), ReqFilterHeading (Value: <>), ReqFilterHeadingML (Value: <>), CalcFields (Value: <Undefined>), ReqFilterFields (Value: <Undefined>), LinkTable (Value: <Undefined>), LinkTableForceInsert (Value: <Yes>), LinkFields (Value: <Undefined>), Temporary (Value: <No>), AutoSave (Value: <Yes>), AutoUpdate (Value: <No>), AutoReplace (Value: <No>), Width (Value: <0>), NamespacePrefix (Value: <Undefined>), MinOccurs (Value: <Once>), and MaxOccurs (Value: <Unbounded>).

Property	Value
Indentation	1
NodeName	Profile
NodeType	Element
SourceType	Table
SourceTable	Profile
VariableName	<Profile>
SourceTableView	<Undefined>
ReqFilterHeading	<>
ReqFilterHeadingML	<>
CalcFields	<Undefined>
ReqFilterFields	<Undefined>
LinkTable	<Undefined>
LinkTableForceInsert	<Yes>
LinkFields	<Undefined>
Temporary	<No>
AutoSave	<Yes>
AutoUpdate	<No>
AutoReplace	<No>
Width	<0>
NamespacePrefix	<Undefined>
MinOccurs	<Once>
MaxOccurs	<Unbounded>

The descriptions of the table-specific properties are as follows:

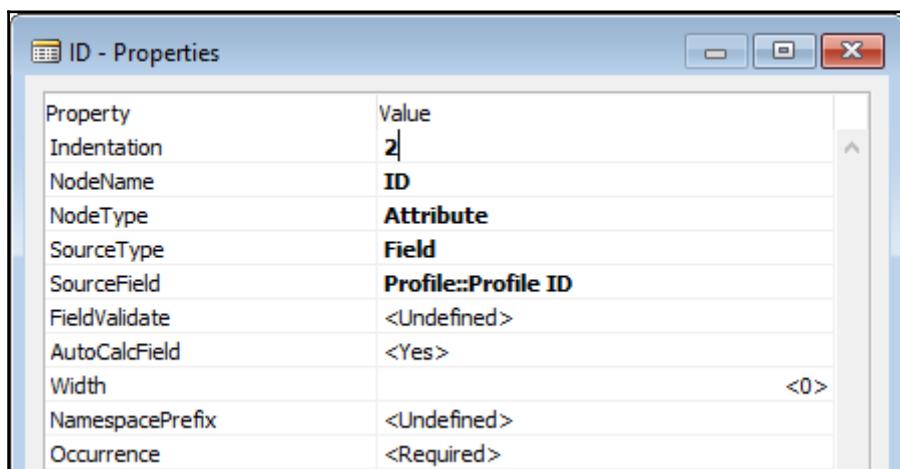
- **SourceTable**: This defines the NAV table being referenced.
- **VariableName**: This defines the name to be used in C/AL code for the NAV table. It is the functional equivalent to the definition of a Global variable.
- **SourceTableView**: This enables the developer to define a view by choosing a 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, we were exporting all of the Purchase Orders for a Vendor, we 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 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.
- **AutoSave**: If set to **Yes** (the default), an imported record will be automatically saved to the table. Either **AutoUpdate** or **AutoReplace** must also be set to **Yes**.
- **AutoUpdate**: If a record exists in the table with a matching primary key, all the data fields are initialized, and then all the data from the incoming record is copied into the table record.

- **AutoReplace:** If a record exists in the table with a matching primary key, the populated data fields in the incoming record are copied into the table record; all the other fields in the target record are left undisturbed. This provides a means to update a table by importing records with a limited number of data fields filled in.

The **Width**, **NamespacePrefix**, **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**, **NamespacePrefix**, **MinOccurs**, and **MaxOccurs** properties will follow in the next section.

## Element or Attribute

An **Element** data item may appear many times, but an **Attribute** data item may only appear no more than once; the occurrence control properties differ based on the **NodeType**.

### NodeType of 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.
- **NamespacePrefix:** When an XMLPort has multiple Namespaces, this property allows you to select a specific one.

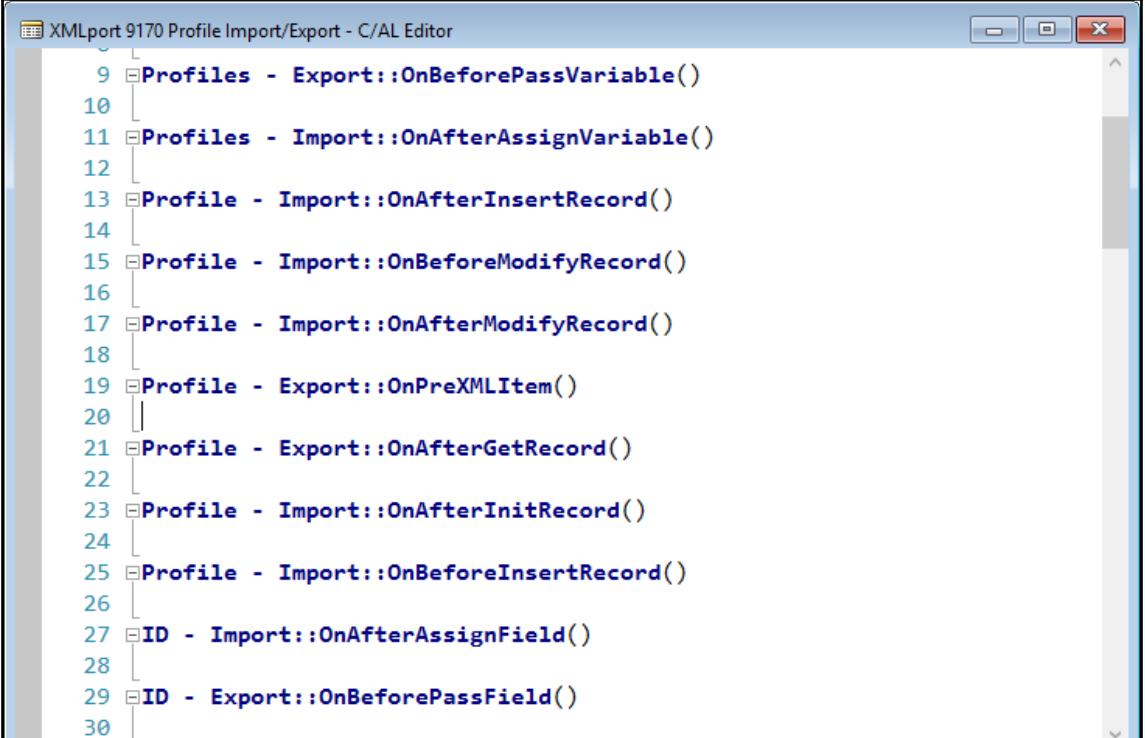
### NodeType of Attribute

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 for the three line Source types: **Profiles -Text**, **Profile -Table**, **ID -Field**:



The screenshot shows the C/AL Editor window titled "XMLport 9170 Profile Import/Export - C/AL Editor". The code editor displays a series of numbered triggers for an XMLport. The triggers are categorized by their scope (Profiles or Profile) and type (Import or Export). The triggers listed are:

- 9 Profiles - Export::OnBeforePassVariable()
- 10 Profiles - Import::OnAfterAssignVariable()
- 11 Profile - Import::OnAfterInsertRecord()
- 12 Profile - Import::OnBeforeModifyRecord()
- 13 Profile - Import::OnAfterModifyRecord()
- 14 Profile - Export::OnPreXMLItem()
- 15 Profile - Export::OnAfterGetRecord()
- 16 Profile - Import::OnAfterInitRecord()
- 17 Profile - Import::OnBeforeInsertRecord()
- 18 ID - Import::OnAfterAssignField()
- 19 ID - Export::OnBeforePassField()

As we can see in the preceding screenshot, there are different XMLport triggers depending on whether **DataType** is Text, Table, or Field.

## **DataType as Text**

The triggers for DataType as Text are as follows:

- **Export::onBeforePassVariable()**, for **Export** only: This trigger is typically used for manipulation of the text variable.
- **Import::OnAfterAssignVariable()**, for **Import** only: This trigger gives us 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 we would put the C/AL code to build and insert records for the permanent database tables.
- **Import::OnBeforeModifyRecord()**, for **Import** only: When AutoSave is Yes, this is used to update the imported data before saving it.
- **Import::OnAfterModifyRecord()**, for **Import** only: When AutoSave is No, this is used to update the data after updating.
- **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.
- **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 we 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.

Any desired options that are to be available to the user as part of the Request Page must be defined in the **RequestOptionsPageDesigner**. 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. As with any other filter setup screen, the user has complete control of what fields are used for filtering and what filters are applied.

## Web services

Web services are an industry standard software interface that allows software applications to interoperate using standard interface specifications along with standard communications services and protocols. When NAV publishes some web services, those functions can be accessed and utilized by properly programmed software residing anywhere on the Web. This 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.

Some benefits of NAV Web Services are as follows:

- Very simple to publish, that is, to expose a web service to a consuming program outside of NAV
- Provides managed access to NAV data while respecting and enforcing NAV rules, logic, and design that already exists
- Uses Windows Authentication and respects NAV data constraints
- Supports **SSL (Secure Socket Layer)**
- Supports both the SOAP interface (cannot access Query objects) and the OData(4)/REST interface (cannot access Codeunit objects)

Disadvantages of NAV Web Services include the following:

- Allowing access to a system from the Web requires a much higher level of security
- The NAV objects that are published generally need to be designed (or, at least, customized) to properly interface with this very different user interface
- Access from the Web complicates the system administrator's job of managing loads on the system

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 as follows:

- What is the degree to which the functionality of the standard RTC interface is needed? A 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 (remember, there is now a full featured web client available for NAV).
- What is 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. These involve additional processing overhead. 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 our NAV system should generally be limited to known users, not the general public. Web services should not be used to provide Internet exposure to the customer's NAV system, but rather for **intranet** access.

Because web services are intended for use by low-intensity users, there are separate license options available with lower costs per user than the full client license. This can make the cost of providing web services-based access quite reasonable, especially if, by doing so, we increase the ability of our NAV users to provide better service to their customers and to realize increased profits.

## Exposing a web service

Three types of NAV objects can be published as Web Services: Pages, Queries, and Codeunits. The essential purposes are as follows:

- Pages provide access to the associated primary table. Use Card Pages for table access unless there is a specific reason to use another page type.
- Codeunits provide access to the functions contained within each Codeunit.
- Queries provide rapid, efficient access to data in a format that is especially compatible with a variety of other Microsoft products, as well as products from other vendors.

An XMLPort can be exposed indirectly as a Codeunit parameter. This provides a very structured way of exposing NAV data through a Web Service. (See AJ Kauffmann's blog series on XMLPorts in Web Services at <http://kauffmann.nl/index.php/2011/01/15/how-to-use-xmlports-in-web-services-1/>). There is an example later in this chapter.

When a Page has no special constraints, either via properties or permissions, there will normally be 11 methods available. They are as follows:

- Create: This creates a single record (similar to a NAV INSERT).
- CreateMultiple: This creates a set of records (passed argument must be an array).
- Read: This reads a single record (similar to a NAV GET).
- ReadMultiple: This reads a filtered set of records, paged. Page size is a parameter.
- Update: This updates a single record (similar to a NAV MODIFY).
- UpdateMultiple: This updates a set of records (passed argument must be an array).
- Delete: This deletes a single record.
- IsUpdated: This checks if the record has been updated since it was read.
- ReadByRecID: This reads a record based on the record ID.
- GetRecIDFromKey: This gets a record ID based on the record key.
- Delete\_<PagePartName> (PagePartRecordKey): This deletes a single record that is exposed by a page part of **TypePage**, such as the **Sales Order Subform** Page Part of the **Sales Order** page.

Whatever constraints have been set in the page that we have published will be inherited by the associated web services. For example, if we publish a page that has the **Editable** property set to **No**, then only the Read, ReadMultiple, and IsUpdated methods will be available as web services. The other five methods will be suppressed by virtue of the **Editable = No** property.

A codeunit that has been published as a web service has its functions made available to for access. A query published as a web service provides access to a service metadata (EDMX) document or an AtomPub Document. To learn more about using queries published as web services, refer to the information published in the *Developer and IT Pro Help* section on MSDN.

## Publishing a web service

Publishing a web service is one of the easiest things we will ever do in NAV. However, as stated earlier, this doesn't mean that we will be able to simply publish existing objects without creating versions specifically tailored for use with Web Services. However, for the moment, let's just go through the basic publishing process.

The point of access is the Departments menu through **NavigationPane** | **Departments** | **Administration** | **ITAdministration** | **General** | **WebServices**. The **Web Services** page displays, as shown in the following screenshot.

The first column allows us to specify whether the object is a Page, Codeunit, or Query. This is followed by the Object ID and then the **ServiceName**. Finally, the **Published** flag must be checked. At this point, the web services for that object are published:

Object Type	Object ID	Object Name	Service Name	All Ten...	Publ...	OData V4 URL	OData URL	SOAP URL
Page	50001	Radio Show Card	Radio_Show_Card	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	http://desktop...	http://desktop...	http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/C...
Codeunit	50001	Rating Webservice	Radio_Show_Management	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Not applicable	Not applicable	http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/C...
Page	89	Jobs	Job List	<input type="checkbox"/>	<input checked="" type="checkbox"/>	http://desktop...	http://desktop...	http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/C...
Page	1007	Job Planning Lines	Job Planning Lines	<input type="checkbox"/>	<input checked="" type="checkbox"/>	http://desktop...	http://desktop...	http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/C...

## Enabling web services

Prior to using web services, we must enable them from the NAV Administration application. In NAV Administration, we can see the checkboxes for enabling. We can either enable SOAP Services or OData Services or both, as shown in the following screenshot:

DynamicsNAV100 - (Running)

General

Database

Client Services 7046

SOAP Services

Enable SOAP Services:  Port: 7047

Enable SSL:  SOAP Base URL: http://DE...

Max Message Size: 1024

OData Services

Enable add in annotations:  Enable V4 Endpoint:

Enable OData Services:  Max Page Size: 1000

Enable SSL:  OData Base URL: http://DE...

Enable V3 Endpoint:  Port: 7048

## Determining what was published

Once an object has been published, we may want to see exactly what is available as a web service. As web services are intended to be accessed from the Web, in the address bar of our browser, we will enter the following (all as one string):

```
http://<Server>:<WebServicePort>/<ServerInstance>/WS/  
<CompanyName>/services
```

Example URL addresses are as follows:

```
http://localhost:7047/DynamicsNAV/WS/Services  
http://Arthur:7047/DynamicsNAV/WS/CRONUS International Ltd  
/Services
```

The company name is optional and case sensitive.



When the correct address string is entered, our browser will display a screen similar to the following screenshot. This screenshot is in an XML format of a data structure called **WSDL (WebServicesDescriptionLanguage)**:

```

<?xml version="1.0"?>
- <discovery xmlns="http://schemas.xmlsoap.org/discovery/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./SystemService"/>
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./Page/PowerBifinance/Show_Management"/>
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./Page/powerbifinance"/>
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./Page/Radio_Show_Card"/>
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./Page/SalesOrder"/>
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./Page/Job_List"/>
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./Page/Job_Planning_Lines"/>
  <contractRef xmlns="http://schemas.xmlsoap.org/discovery/sci/" ref="http://desktop-ghc1ao6:7047/DynamicsNAV100/WS/CRONUS USA%2C Inc./Page/Job_Task_Lines"/>
</discovery>
```

In this case, we can see that we have two NAV SOAP Services available:  
Codeunit/Radio\_Show\_Management and Page/Radio\_Show\_Card.

To see the methods, which are NAV 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 can change the URL used earlier to replace the word `Services` with `Codeunit/Radio_Show_Management`. We must also include the company name in the URL that lists the methods WSDL.

To see the OData services, change the URL to one in the form, as follows:

```
http://<Server>:<WebServicePort>/<ServerInstance>/OData
```

From that entry in our browser, we get information about what's available as OData. OData is structured like XML, but OData provides the full metadata (structural definition) of the associated data in the same file as the data. This allows OData to be consumed without the requirement of a lot of back and forth technical preplanning communications:

```

<?xml version="1.0" encoding="UTF-8"?>
- <service xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom" xml:base="http://ghc1ao6:7048/DynamicsNAV100/OData/">
  - <workspace>
    <atom:title>Default</atom:title>
    - <collection href="powerbifinance">
      <atom:title>powerbifinance</atom:title>
      </collection>
    - <collection href="Radio_Show_Card">
      <atom:title>Radio_Show_Card</atom:title>
      </collection>
    - <collection href="SalesOrder">
      <atom:title>SalesOrder</atom:title>
      </collection>
```

The actual consumption (meaning "use of") of a web service is also fairly simple, but that process occurs outside of NAV in any of a wide variety of off-the-shelf or custom applications that are not part of this book's focus. Examples are readily available in Help, the MSDN library, the NAV forums, and elsewhere.

Tools that can be used to consume NAV Web Services include, among others, Microsoft Power BI, 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.

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 2017 should be invoked for the application to work properly. In many cases, we 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 we want to provide just two or three functions from a Codeunit that contains many additional functions. Or we may 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, perhaps in a codeunit that can be published as web services (generally recognized as a best practice).



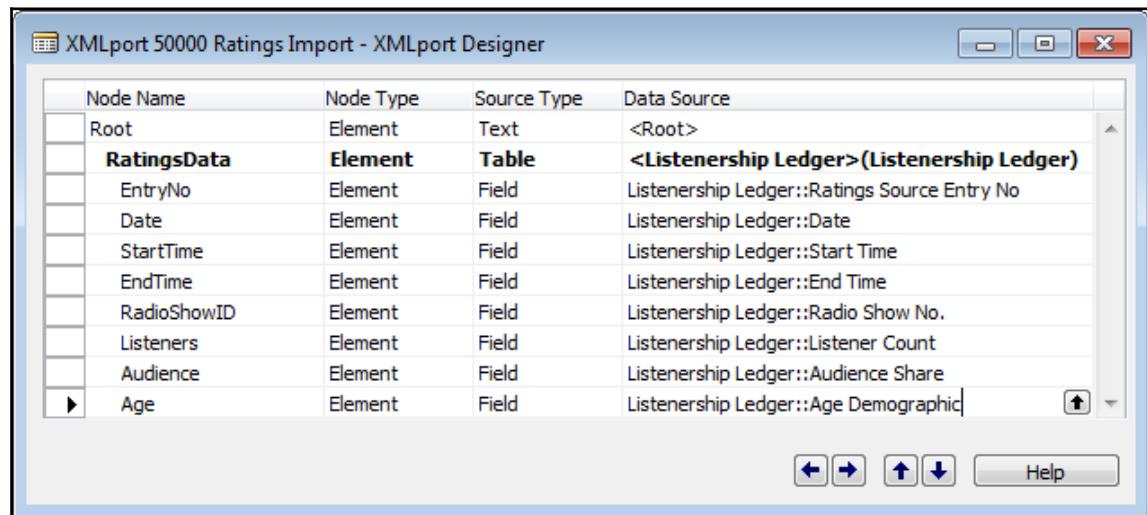
Use of web services carries with it issues that must be dealt with in any production environment. In addition to delivering the required application functionality, there are security, access, and communications issues that need to be addressed. It is recommended that a NAV Web Service may not be directly exposed to external users, but NAV data may be secured by limiting access through the use of custom, functionally limited, external software interfaces. While outside the scope of this text, proper attention to data security is critical to the implementation of a good quality solution.

## XMLport - Web Services Integration example for WDTU

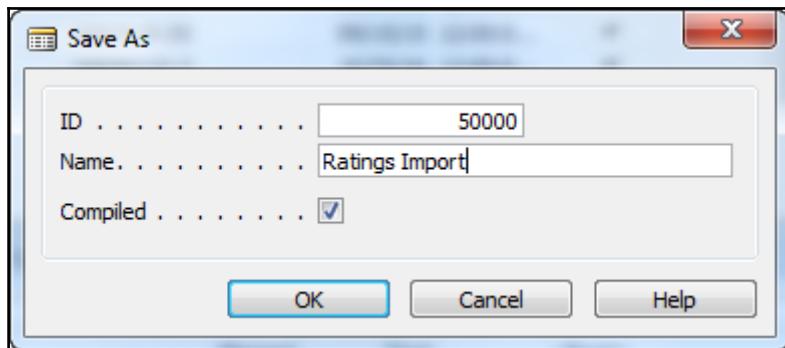
WDTU subscribes to a service that compiles listenership data. This data is provided to subscribers in the form of XML files. The agency that provides the service has agreed to push that XML data directly to a web service exposed by our NAV 2017 system. This approach will allow WDTU to have access to the latest listenership data as soon as it is released by the agency.

WDTU must provide access to the XMLport that fits the incoming XML file format. The handshaking response expected by the agency computer from our web service is a fixed XML file with one element (Station ID) and an attribute of said element (Frequency).

The first step is to build our XMLport. We access the XMLport designer through the **Object Designer** | **XMLport button** | **New button**. Define the new XMLport lines as shown in the following screenshot:



After we have the lines entered, we will click on *Alt + F* and then **Save As....**. Fill in the **Save As** screen as shown and click on **OK** to save and compile the XMLport without exiting the XMLport Designer:

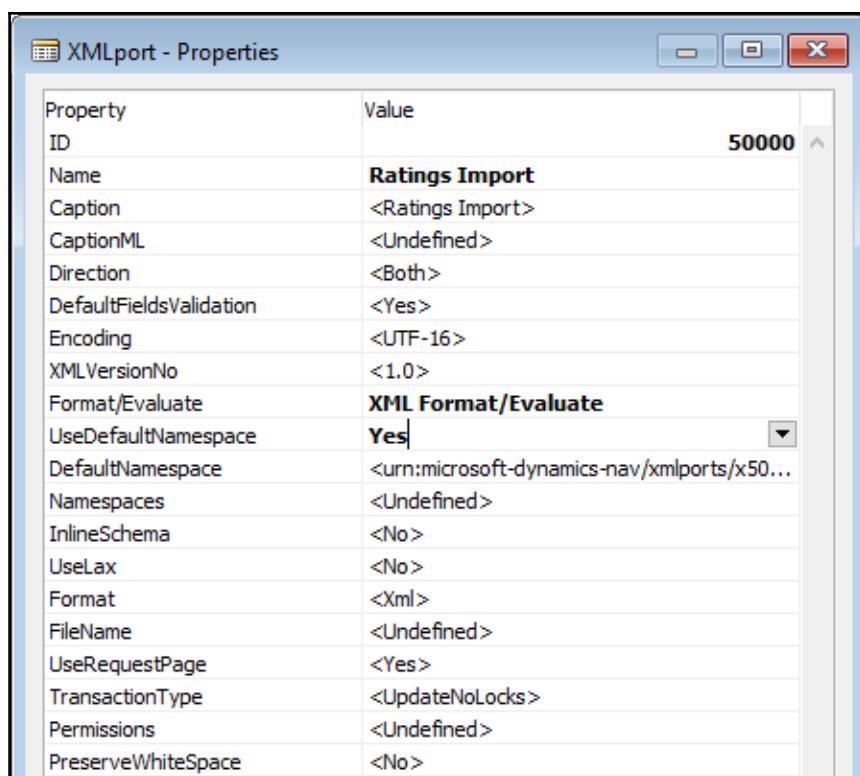


Highlight the blank line at the bottom of the XMLport Designer screen and click on *Shift + F4*, the Properties icon, or right-click and then click on **Properties** to display the XMLport properties screen.

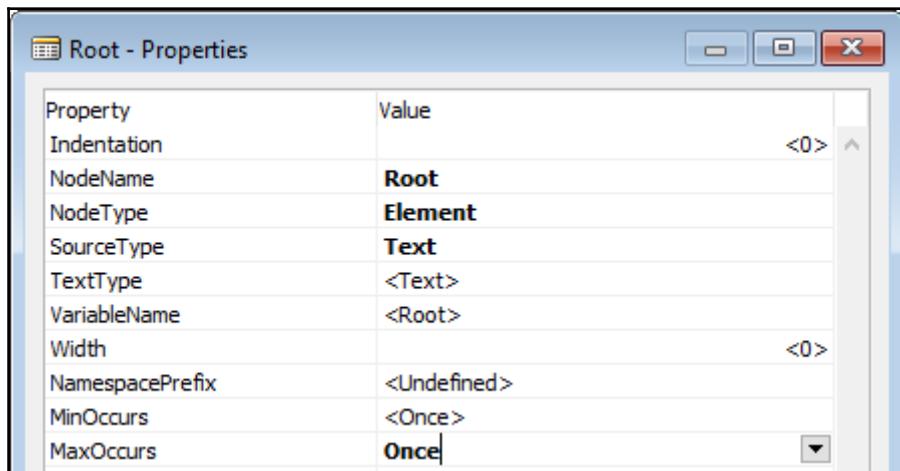
Set the **Format/Evaluate** property to `XML Format/Evaluate`. This allows Visual Studio to automatically understand the data types, such as integer, decimal, and so on, involved. Set **UseDefaultNamespace** to `Yes`, and the **DefaultNamespace** to `urn:Microsoft-dynamics-nav/xmlports/x50000`, which is the default format, or `urn:Microsoft-dynamics-nav/xmlports/RatingsImport`.

Even though we are using the XMLport as an import only object, make sure the **Direction** property stays at `<Both>`. When the value is set to either `Import` or `Export`, it is not possible to use the XMLport as a **Var** (by reference) parameter in the codeunit function, which we will expose as a web service.

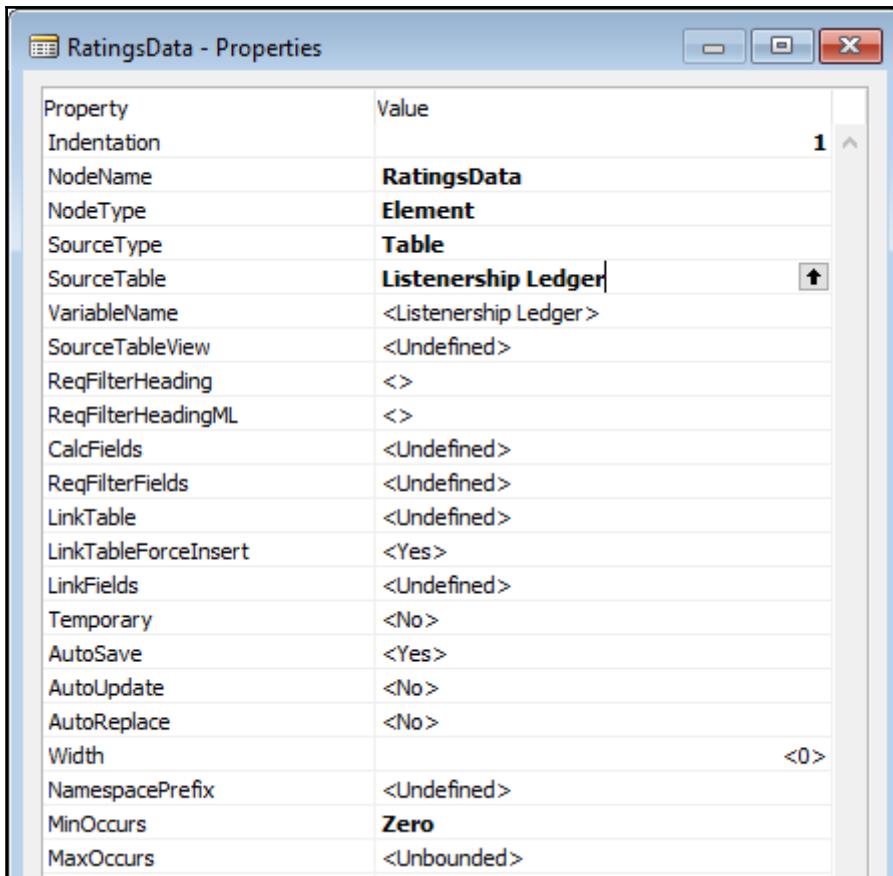
Following is the XMLport 50000 Properties screen with those changes in place:



After we close the **Properties** screen for the XMLport, we can highlight the **Root Element** line and display its properties. Set the **MaxOccurs** property to **Once**, as shown in the following screenshot:

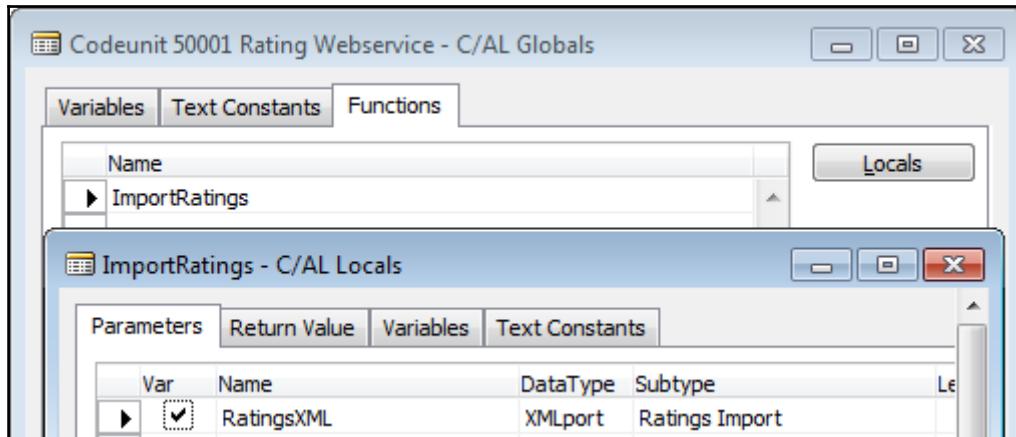


Close the **Root Properties**, highlight the **RatingsData Table Element**, and access its **Properties** screen. Set **MinOccurs** to **Zero** and make sure **MaxOccurs** remains at the default value of **<Unbounded>**, as shown in the following screenshot. Once that is done, close the **RatingsData -Properties** screen. As our XMLport matches the incoming data format from the listenership ratings agency, no C/AL code is necessary in this XMLport. Exit the **XMLport Designer** and save and compile **XMLport 50000**:

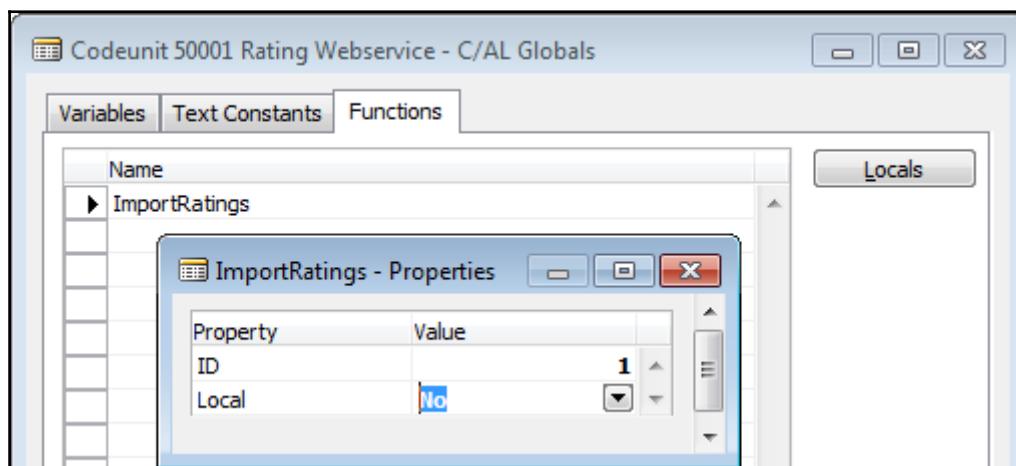


Now that we have our XMLport constructed, it's time to build the codeunit that will be published as a web service. Go to **Object Designer** | **Codeunit button** | **New button**. Then click on the menu option **View** | **C/AL Globals** | **Functions** tab. Enter the new **Function** name of **ImportRatings** and click on the **Locals** button.

In the **C/AL Locals** screen, enter the single parameter, RatingsXML, **Type** XMLport, and a **SubType** of Ratings Import. Make sure the **Var** column on the left is checked. The **C/AL Locals** screen should then look like the following screenshot:

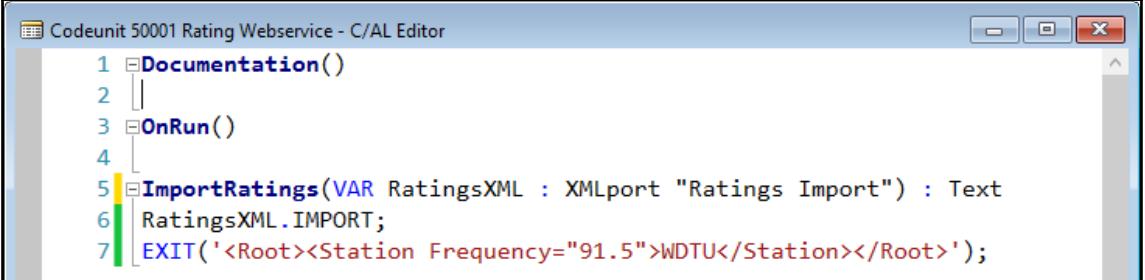


Now, click on the **Return Value** tab and set the **Return Type** value to **Text** and the **Length** value to 250. Exit the **C/AL Locals** and **C/AL Globals** screens, returning to the **Codeunit C/AL Editor** screen. Finally, we will highlight our new function in the **Functions** tab, and set the **Local** property to **No** so that we can access this function from Web Services, as shown in the following screenshot:



Before proceeding further, let's save our work. Just as in the **XMLport Designer**, we can save our work without exiting the Designer by clicking on **File | Save As...**, and then entering the designer object number (**50001**) and name (**Ratings Webservice**).

We only need two lines of C/AL code in the codeunit. The first line's task is to import the XML utilizing the XMLport parameter to cause the XMLport to process. The second code line's purpose is to send the required text response back to the external system, with the response formatted as XML data. In the **C/AL Editor**, that code looks as follows:



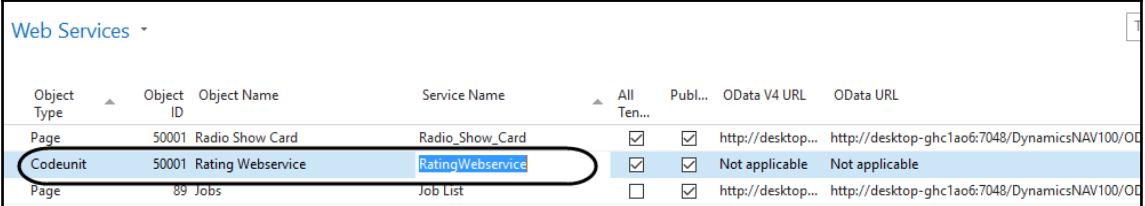
```

Codeunit 50001 Rating Webservice - C/AL Editor
1 Documentation()
2
3 OnRun()
4
5 ImportRatings(VAR RatingsXML : XMLport "Ratings Import") : Text
6 RatingsXML.IMPORT;
7 EXIT('<Root><Station Frequency="91.5">WDTU</Station></Root>');

```

Exit **Codeunit 5001** and compile and save it. Now, we will publish the codeunit that we just created.

Open the Role Tailored Client, navigate to **Departments | Administration | IT Administration | General | Web Services** (or just search for Web Services using the **Search** box), and invoke the **Web Services** page. Fill in **Object Type** of **Codeunit**, **Object ID** of **50001**, **Service Name** of **WDTU Ratings**, and check **Published**:



Object Type	Object ID	Object Name	Service Name	All Ten...	Publis...	ODATA V4 URL	ODATA URL
Page	50001	Radio Show Card	Radio_Show_Card	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	http://desktop...	http://desktop-ghc1ao6:7048/DynamicsNAV100/OD
Codeunit	50001	Rating Webservice	RatingWebservice	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Not applicable	Not applicable
Page	89	Jobs	Job List	<input type="checkbox"/>	<input checked="" type="checkbox"/>	http://desktop...	http://desktop-ghc1ao6:7048/DynamicsNAV100/OD
Page	1007	L1 PL	L1 PL	<input type="checkbox"/>	<input type="checkbox"/>	L1 PL	L1 PL - 6.7048/D... - NAV100/OD

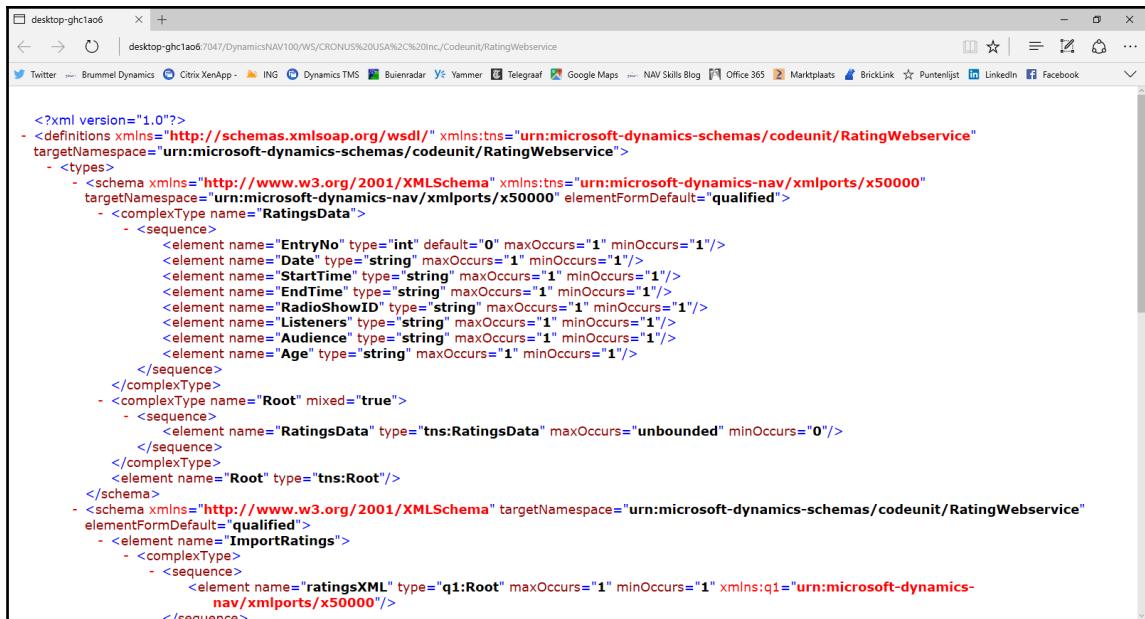
Now, to test what we've done, we need to open a browser and enter a URL in the following format:

```
http://<Server>:<SOAPWebServicePort>/<ServerInstance>/WS/
<CompanyName>/services
```

For example, more specifically, like the following URL:

[http://localhost:7047/DynamicsNAV100/WS/CRONUS International  
Ltd/Codeunit/WDTURatings](http://localhost:7047/DynamicsNAV100/WS/CRONUS International Ltd/Codeunit/WDTURatings)

Instead, for testing purposes, we could just click on the Web icon on the right-hand end of our new entry in the **Web Services** screen, as shown in the preceding screenshot. The result in our browser screen should look like the following screenshot, showing that we can connect with our web service and that our XMLport contains all the fields for the data we plan to import:



The screenshot shows a Microsoft Edge browser window with the title bar "desktop-ghc1a06" and the address bar "desktop-ghc1a06:7047/DynamicsNAV100/WS/CRONUS%20USA%2C%20Inc/Codeunit/RatingWebservice". The main content area displays the XML schema definition for the RatingWebservice. The XML code includes definitions for types, complex types, and sequences, defining fields such as EntryNo, Date, StartTime, EndTime, RadioShowID, Listeners, Audience, and Age.

```
<?xml version="1.0"?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="urn:microsoft-dynamics-schemas/codeunit/RatingWebservice"
targetNamespace="urn:microsoft-dynamics-schemas/codeunit/RatingWebservice">
- <types>
- <schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:tns="urn:microsoft-dynamics-nav/xmlports/x50000" elementFormDefault="qualified">
- <complexType name="RatingsData">
- <sequence>
<element name="EntryNo" type="int" default="0" maxOccurs="1" minOccurs="1"/>
<element name="Date" type="string" maxOccurs="1" minOccurs="1"/>
<element name="StartTime" type="string" maxOccurs="1" minOccurs="1"/>
<element name="EndTime" type="string" maxOccurs="1" minOccurs="1"/>
<element name="RadioShowID" type="string" maxOccurs="1" minOccurs="1"/>
<element name="Listeners" type="string" maxOccurs="1" minOccurs="1"/>
<element name="Audience" type="string" maxOccurs="1" minOccurs="1"/>
<element name="Age" type="string" maxOccurs="1" minOccurs="1"/>
</sequence>
</complexType>
- <complexType name="Root" mixed="true">
- <sequence>
<element name="RatingsData" type="tns:RatingsData" maxOccurs="unbounded" minOccurs="0"/>
</sequence>
</complexType>
<element name="Root" type="tns:Root"/>
</schema>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:microsoft-dynamics-schemas/codeunit/RatingWebservice"
elementFormDefault="qualified">
- <element name="ImportRatings">
- <complexType>
- <sequence>
<element name="ratingsXML" type="q1:Root" maxOccurs="1" minOccurs="1" xmlns:q1="urn:microsoft-dynamics-
nav/xmlports/x50000"/>
</sequence>
```

## Review questions

1. Users cannot delete or modify database data through Web Services. True or False?
2. 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
3. Software external to NAV that accesses NAV Web Services must be .NET compatible. True or False?
4. Action Ribbons can be modified in which of the following ways. Choose three:
  - a) C/Side changes in the Action Designer
  - b) Dynamic C/AL code Configuration
  - c) Implementer/Administrator Configuration
  - d) User Personalization
  - e) Application of a profile template
5. XMLports cannot contain C/AL code. All data manipulation must occur outside of the XMLport object. True or False?
6. The default **PromotedActionCategoriesML** includes which two of the following?
  - a) Page
  - b) Report
  - c) Standard
  - d) New
7. Web services are an industry-standard interface defined by the World Wide Web Consortium. True or False?

8. Role Centers can have several components. Choose two:
  - a) Activity Pane
  - b) Browser Part
  - c) Cues
  - d) Report Review Pane
9. Web Services are a good tool for publishing NAV to be used as a public retail sales system on the Web. True or False?
10. An action can appear in multiple places in a Role Center, for example, in the Home button of the Navigation Pane, in the Action Ribbon, and in a Cue Group. True or False?
11. Which of the following object types can be published as Web Services? Choose two:
  - a) Reports
  - b) Queries
  - c) XMLPorts
  - d) Pages
12. A Cue may be a shortcut to a filtered list supported by a field in a Cue table. True or False?
13. New System Parts can be created by an NAV developer using C/SIDE tools. True or False?
14. All new implementations should create new Role Centers based on the results of detailed analysis of the intended users' work roles, only using the standard Role Centers as templates. True or False?
15. All Web Service data interfaces require the use of XML data files. True or False?
16. Role Center Cues can be tied to FlowFields, Normal fields, or Queries. True or False?
17. Actions in a ribbon are Promoted so that they can be displayed in color. True or False?

18. Role Center components include which of the following? Choose two:
  - a) Cues
  - b) Microsoft Office parts
  - c) Web Services
  - d) Page parts
19. Once a Role Center layout has been defined by the Developer, it cannot be changed by the users. True or False?
20. In the Role Tailored Client, XMLports can only be used to process XML formatted files and cannot process other text file formats. True or False?

## **Summary**

In this chapter, we reviewed some of the more advanced NAV 2017 tools and techniques. By now, we should have a strong admiration for the power and flexibility of NAV 2017. Many of these subject areas require more study and hands-on practice by you. We spent a lot of time on Role Center construction because that is the heart of the Role Tailored Experience. Much of what you learn about Role Center design and construction can be applied across the board in role tailoring other components. We went through XMLports and Web Services, and then showed how the two capabilities can be combined to provide a simple, but powerful, method of interfacing with external systems. By now, you should be almost ready to begin your own development project. In the next chapter, we will cover the debugger, extensibility (adding non-C/AL controls to pages), and additional topics.

# 9

## Successful Conclusions

*"The expected never happens; it is the unexpected always."*

- John Maynard Keynes

*"The most powerful designs are always the result of a continuous process of simplification and refinement."*

- Kevin Mullet

Each new version of NAV includes new features and capabilities. **Microsoft Dynamics NAV 2017** is no exception. It is our responsibility to understand how to apply these new features, use them intelligently in our designs, and develop with both their strengths and limitations in mind. The new features in NAV 2017 include a new Job Scheduler and a new way to display SQL server locking information. Our goal in the end is not only to provide workmanlike results but, if possible, to delight our users with the quality of the experience and the ease to use our products.

In this chapter, we will cover the following topics:

- Reviewing NAV objects that contain functions we can use directly or as templates in our solutions
- Reviewing some of the NAV C/SIDE tools that help us debug our solutions
- Learning ways we can enhance our solutions using external controls integrated into NAV
- Discussing design, development, and delivery issues that should be addressed in our projects

## Creating new C/AL routines

Now that we have a good overall picture of how we enable users to access the tools we create, we are ready to start creating our own NAV C/AL routines. 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 we create should be visually and logically compatible with what already exists. If we think of our new code as a guest being hosted by the original system, we 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; we can take advantage of the features and components of the standard product to address some of our application requirements. There are at least two types of existing NAV C/AL code of which we should make use whenever appropriate.

One group is the callable functions that are used liberally throughout NAV. Once we know about these, we can use them in our logic whenever they fit. There is no documentation for most of these functions, so we must either learn about them here or by doing our homework and studying NAV code. The second group includes the many code snippets we can copy when we 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 serve as models, code that must be modified to fit the situation by changing variable names, adding or removing constraints, and so on.

In addition to the directly usable C/AL code, we should also make liberal use of the NAV Design Patterns Repository located at  
<https://community.dynamics.com/nav/w/designpatterns/105.nav-design-patterns-repository.aspx>.

NAV Design Patterns provide common definitions of how certain types of functions are implemented in NAV. Some of the Pattern examples include the following:

- Copy Document
- Create Data from Templates
- No. Series
- Single-Record (Setup) Table
- Master Data

There are many other patterns, and new pattern definitions are frequently added.

## Callable functions

Most of the callable functions in NAV are designed to handle a very specific set of data or conditions and have no general-purpose use, for example, the routines to update Check Ledger entries during a posting process are likely to apply only to that specific function. If we are making modifications to a particular application area within NAV, we may find functions that we can utilize, either as is or as models for our new functions.

There are also many functions within NAV that are relatively general purpose. They either act on data that is common in many different situations, such as dates and addresses, or they perform processing tasks that are common to many situations, such as providing access to an external file. We will review a few such functions in detail, then list a number of others worth studying. If nothing else, these functions are useful as guides for showing how such functions are used in NAV. The various parameters in these explanations are named to assist with your learning, not named the same as in the NAV code (though all structures, data types, and other technical specifications match the NAV code).

If we are using one of these functions, we must take care to clearly understand how it operates. In each case, we should study the function and test with it before assuming we adequately understand how it works. There is little or no documentation for most of these functions, so understanding their proper use is totally up to us. If we need a customization, that must be done by making a copy of the target function and then modifying the copy.

## Codeunit 358 - DateFilterCalc

This codeunit is a good example of how well designed and well written code has long term utility. Except for code changes required by NAV structural changes, this codeunit has changed very little since it originated in NAV (Navision) V3.00 in 2001. That doesn't mean it is out of date; it means it was well thought out and complete from its beginning.

Codeunit 358 contains two functions we can use in our code to create filters based on the **Accounting Period Calendar**. The first is `CreateFiscalYearFilter`. If we are calling this from an object that has Codeunit 358 defined as a Global variable named `DateFilterCalc`, our call would use the following syntax:

```
DateFilterCalc.CreateFiscalYearFilter  
(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:

```
DateFilterCalc.CreateAccountingPeriodFilter  
(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, we can see how NAV calls these functions. Page 152 - Vendor Statistics, Page 223 - Resource Statistics, and a number of other Master table statistics pages also use this set of functions:

```
13  IF OnAfterGetRecord()  
14  IF CurrentDate <> WORKDATE THEN BEGIN  
15    CurrentDate := WORKDATE;  
16    DateFilterCalc.CreateAccountingPeriodFilter(CustDateFilter[1],CustDateName[1],CurrentDate,0);  
17    DateFilterCalc.CreateFiscalYearFilter(CustDateFilter[2],CustDateName[2],CurrentDate,0);  
18    DateFilterCalc.CreateFiscalYearFilter(CustDateFilter[3],CustDateName[3],CurrentDate,-1);  
19 END;
```

As shown 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 Customers Statistics page:

```
23 FOR i := 1 TO 4 DO BEGIN  
24   SETFILTER("Date Filter",CustDateFilter[i]);  
25   CALCFIELDS(  
26     "Sales (LCY)","Profit (LCY)","Inv. Discounts (LCY)","Inv. Amounts (LCY)","Pmt. Discounts (LCY)",  
27     "Pmt. Disc. Tolerance (LCY)","Pmt. Tolerance (LCY)",  
28     "Fin. Charge Memo Amounts (LCY)","Cr. Memo Amounts (LCY)","Payments (LCY)",  
29     "Reminder Amounts (LCY)","Refunds (LCY)","Other Amounts (LCY)");
```

When one of these functions is called, the `Filter` and `Name` parameters are updated within the function so we can use them 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` we supply.

The returned filter is supplied back in the format of a range filter string, '`startdate..enddate`' (for example, `01/01/16..12/31/16`). If we call `CreateFiscalYearFilter`, the `Filter` parameter will be for the range of a fiscal year, as defined by the system's `AccountingPeriod` table. If we call `CreateAccountingPeriodFilter`, the `Filter` parameter 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 (go 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**

The description of the `FindDate` function is:

- Calling Parameters of (`SearchString` (text, length 3), `Calendar` (Date table), `PeriodType` (Option, integer))
- Returns `DateFound` Boolean

The calling syntax for the `FindDate` function is:

```
BooleanVariable := 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 virtual Date table based on the parameters passed. The search starts with an initial record in the `Calendar` table. If we pass in a record that has already been initialized by positioning the table at a date, that will be the base date, otherwise the Work Date will be used.

The `PeriodType` is an Option's field with the option value choices of day, week, month, quarter, year, and accounting period. For ease of coding, we could call the function with the integer equivalent (0, 1, 2, 3, 4, 5), or set up our 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.

The `SearchString` allows us to pass in a logical control string containing `=`, `>`, `<`, `<=`, `>=`, and so on. `FindDate` will find the first date starting with the initialized `Calendar` date that satisfies the `SearchString` logic instruction and fits the `PeriodType` defined. For example, if the `PeriodType` is `day`, and the date `01/25/16` is used along with the `SearchString` of `>`, then the date `01/26/16` will be returned in the `Calendar`.

## **NextDate function**

The description of the `NextDate` function is:

- Calling Parameters (`NextStep` (integer), `Calendar` (Date table), `PeriodType` (Option, integer))
- Returns Integer

The calling syntax for the `NextDate` function is:

```
IntegerVariable := NextDate(NextStep, CalendarRec, PeriodType)
```

`NextDate` will find the next date record in the `Calendar` table that satisfies the calling parameters. The `Calendar` and `PeriodType` calling parameters for `NextDate` have the same definition as they do for the `FindDate` function. However, for this function to be really useful, the `Calendar` must be initialized before calling `NextDate`. Otherwise, the function will calculate the appropriate next date from day 0. The `NextStep` parameter allows us to define the number of periods of `PeriodType` to move, so as to obtain the desired next date. For example, if we start with a `Calendar` table positioned on `01/25/16`, a `PeriodType` of `quarter` (that is 3), and a `NextStep` of 2, the `NextDate` will move forward two quarters and return with `Calendar` focused on Quarter, `7/1/16` to `9/30/16`.

## **CreatePeriodFormat function**

The description of the `CreatePeriodFormat` function is:

- Calling Parameters (`PeriodType` (Option, integer), `Date` (date))
- Returns - Text, length 10

The calling syntax for the `CreatePeriodFormat` function is:

```
FormattedDate := CreatePeriodFormat(PeriodType, DateData)
```

`CreatePeriodFormat` allows us to supply a date and specify which of its format options we want through the `PeriodType`. The function's return value is a ten-character formatted text value, for example, `mm/dd/yy` or `ww/yyyy`, `mon/yyyy`, `qtr/yyyy`, or `yyyy`.

## Codeunit 365 - Format Address

The functions in the `Format Address` codeunit do the obvious--they format addresses in a variety of situations. 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 we 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 such tasks.

In addition, NAV provides setup options for multiple formats of City - Post Code - County - Country combinations. The `Format Address` functions format addresses according to what was chosen in the setup or was defined in the Countries/Regions page for different Postal areas.

There are over 60 data-specific functions in the `Format Address` codeunit. Each data-specific function allows us 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, plus a parameter of a one-dimensional Text array with eight elements of length up to 90 characters. Each function extracts the address data from its specific master record format and stores it in the array. The function then passes that data to a general-purpose function, which does the actual work of resequencing, according to the various setup rules, and compressing the data by removing blank lines.

The following code examples are of the function call format for the functions of Company and the SalesShip-to addresses. In each case, `AddressArray` is Text, Length 90, and one-dimensional with eight elements:

```
"Format Address".Company(AddressArray,CompanyRec);  
"Format Address". SalesHeaderSellTo (AddressArray,SalesHeaderRec);
```

The function's processed result is returned in the `AddressArray` parameter.

In addition to the data-specific functions in the `FormatAddress` codeunit, we can also directly utilize the more general-purpose functions contained there. If we add a new address structure as part of an enhancement, we may want to create our own data-specific address formatting function in our custom codeunit. However, we should design our function to call the general purpose functions that already exist (and are already debugged).

The primary general-purpose address formatting function (the one we 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, CountryCode)
```

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 data type `Code`, length 20, and length 10, respectively.

Our data is passed into the function in the individual `Address` fields. The results are passed back in the `AddressArray` parameter for us to use.

There are two other functions in the `FormatAddress` codeunit that are often 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.

On the same website as the Dynamics Community NAV Patterns, there is a section entitled chapter, *Recipes - The NAV C/AL Cookbook*. One of those recipes, *Integration of Addresses*, applies to the preceding section on address formatting and has discussion about this topic at <https://community.dynamics.com/nav/w/designpatterns/234.address-integration.aspx>.

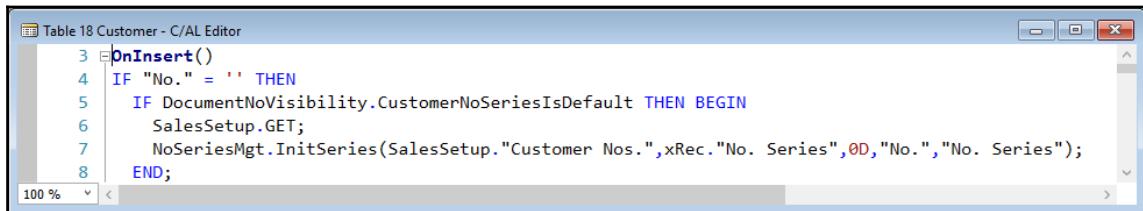
## Codeunit 396 - NoSeriesManagement

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 identification number assigned to each one. This unique identification 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)
```

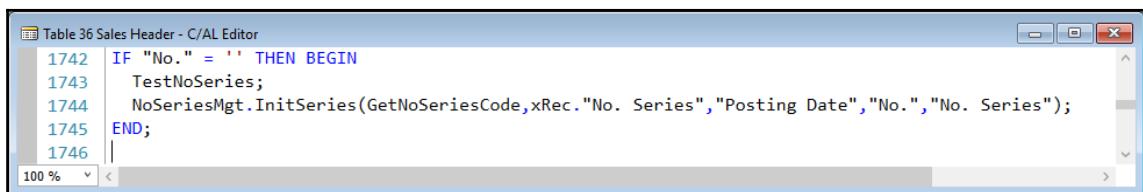
The parameter `WhichNumberSeriesToUse` is generally defined on a Numbers Tab in the Setup record for the applicable application area. The `LastDataRecNumberSeriesCode` tells the function what Number Series was used for the previous record in this table. The `SeriesDateToApply` parameter allows the function to assign ID numbers in a date-dependent fashion. The `NumberToUse` and the `NumberSeriesUsed` are return parameters.

The following screenshot shows an example for **Table 18 - Customer**:



```
Table 18 Customer - C/AL Editor
3 | 3 | OnInsert()
4 | 4 | IF "No." = '' THEN
5 | 5 |   IF DocumentNoVisibility.CustomerNoSeriesIsDefault THEN BEGIN
6 | 6 |     SalesSetup.GET;
7 | 7 |     NoSeriesMgt.InitSeries(SalesSetup."Customer Nos.",xRec."No. Series",0D,"No.","No. Series");
8 | 8 |   END;
100 % < >
```

The following screenshot shows a second example for **Table 36 - Sales Header**. In this case, the call to `NoSeriesMgt` has been placed in a local function:



```
Table 36 Sales Header - C/AL Editor
1742 | 1742 | IF "No." = '' THEN BEGIN
1743 | 1743 |   TestNoSeries;
1744 | 1744 |   NoSeriesMgt.InitSeries(GetNoSeriesCode,xRec."No. Series","Posting Date","No.","No. Series");
1745 | 1745 | END;
1746 | 1746 |
100 % < >
```

With the exception of `GetNextNo` (used in assigning unique identifying numbers to each of a series of transactions) and, possibly, `TestManual` (used to test if manual numbering is allowed), we 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.

There is also an NAV Pattern defined describing the use of number series in NAV. It is entitled *No. Series* and can be found at <https://community.dynamics.com/nav/w/designpatterns/74.no-series>.

## Function models to review and use

It is very helpful when creating new code to have a model that works which we can study (or clone). This is especially true in NAV, where there is little or no development documentation available for many of the different functions we would like to use. One of the more challenging aspects of learning to develop in the NAV environment is learning how to handle issues in the **NAV way**. Learning the NAV way is very beneficial, because then our code works better, is easier to maintain and easier to upgrade. There is no better place to learn the strengths and subtle features of the product than to study the code written by the developers who are part of the inner circle of NAV creation.



If there is a choice, don't add custom functions to the standard NAV Codeunits. Well segregated customizations in clearly identified custom objects make both maintenance and upgrades easier. When we build functions modeled on NAV functions, the new code should be in a customer licensed codeunit.

A list of objects follows that contain functions we may find useful to use in our code or as models. We find these useful to study how it's done in NAV ("it" obviously varies, depending on the function's purpose):

- **Codeunit 1 - Application Management:** This is a library of utility functions widely used in the system
- **Codeunits 11, 12, 13, 21, 22, 23, 80, 81, 82, 90, 91, and 92:** These are the Posting sequences for Sales, Purchases, General Ledger, Item Ledger; these control the posting of journal data into the various ledgers
- **Codeunit 228 - Test Report-Print:** Functions for printing Test Reports for user review prior to Posting data
- Codeunit 229 - Print Documents: Functions for printing Document formatted reports
- **Codeunits 397 and 400 - Mail:** Functions for interfacing with Outlook and SMTP mail
- **Codeunit 408 - Dimension Management:** Don't write your own; use these
- **Codeunit 419 - File Management:** Functions including BLOB tasks, file uploading, and downloading between server and client systems.
- **Codeunits 802:** Online Map interfacing
- **Codeunit 5054- Word Management:** Interfaces to Microsoft Word
- **Codeunit 5063 - Archive Management:** Storing copies of processed documents
- **Codeunits 5300 through 5313:** More Outlook interfacing

- **Codeunits 5813 through 5819:** Undo functions
- **Codeunit 6224:** XML DOM Management for XML structure handling
- **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 over 150 codeunits with the word **Management** (or the abbreviation **Mgt**) as part of their description name (filter the Codeunit Names using `*Management* | *Mgt*` - don't forget that such a filter is case sensitive). Each of these codeunits contains functions whose purpose is the management of some specific aspect of NAV data. Many are specific to a very narrow range of data. Some are more general because they contain functions we can reuse in another application area (for example, Codeunit 396 - `NoSeriesManagement`).

When we are working on an enhancement in a particular functional area, it is extremely important to check the Management codeunits utilized in that area. We may be able to use some existing standard functions directly. This will have the benefit of reducing the code we have to create and debug. Of course, when a new version is released, we will have to check to see if the functions on which we relied have changed in a way that affects our code.

If we can't use the existing material as is, we may find functions we can use as models for tasks in the area of our 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 process flow in the standard NAV system.

## Multi-language system

The NAV system is designed as a multi-language system, meaning it can interface with users in many languages. 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. Because 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, it has no effect on the data in the database. The data is not multi-language unless we provide that functionality, by means of our own enhancements or data structure.



Microsoft's automatic translation does not support Unicode languages such as Chinese and Japanese out of the box but it can be (and has been) done. See Alex Chow's blog at <http://www.dynamicsnavconsultant.com/2014/06/chinese-language-pack-dynamics-nav/navision/>.

Use the following URL to read about the NAV Pattern, Multilanguage Application Data, which describes how data and UI elements can be multilanguage enabled:

<https://community.dynamics.com/nav/w/designpatterns/77.multilanguage-application-data>.

The basic elements that support the multi-language feature include the following:

- Multi-Language Captioning properties (for example, **CaptionML**) supporting definitions of alternative language captions for all the fields, action labels, titles, and so on
- The Application Management codeunit logic that allows language choice at login
- The `fin.stx` files supplied by NAV, which are language specific and contain texts used by NAV executables, such as the Windows Client and Development Environment (`fin.stx` cannot be modified, except by Microsoft)
- The Text Constants property, `ConstantValueML`, supporting definition of alternative language messages

Before embarking on an effort to create multi-language enabled modifications, review all the available documentation on the topic (search Help on the word **language**). It's wise to do some small scale testing to ensure that you understand what is required, and that your approach will work (such testing is required for any significant enhancement).

## 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, we can create the order in US dollars and accept payment for the invoice in Euros or Bitcoin or Yen. For this reason, where there are money values, they are generally stored in the local currency (referenced as **LCY**), as defined in setup. 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 the handling of money fields are as follows:

```
754 | GetSalesHeader;
755 | IF SalesHeader."Currency Code" <> '' THEN BEGIN
756 |     Currency.TESTFIELD("Unit-Amount Rounding Precision");
757 |     "Unit Cost" := 
758 |         ROUND(
759 |             CurrExchRate.ExchangeAmtLCYToFCY(
760 |                 GetDate,SalesHeader."Currency Code",
761 |                 "Unit Cost (LCY)",SalesHeader."Currency Factor"),
762 |                 Currency."Unit-Amount Rounding Precision")
763 | END ELSE
764 |     "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 as shown in the following code snippet screenshot:

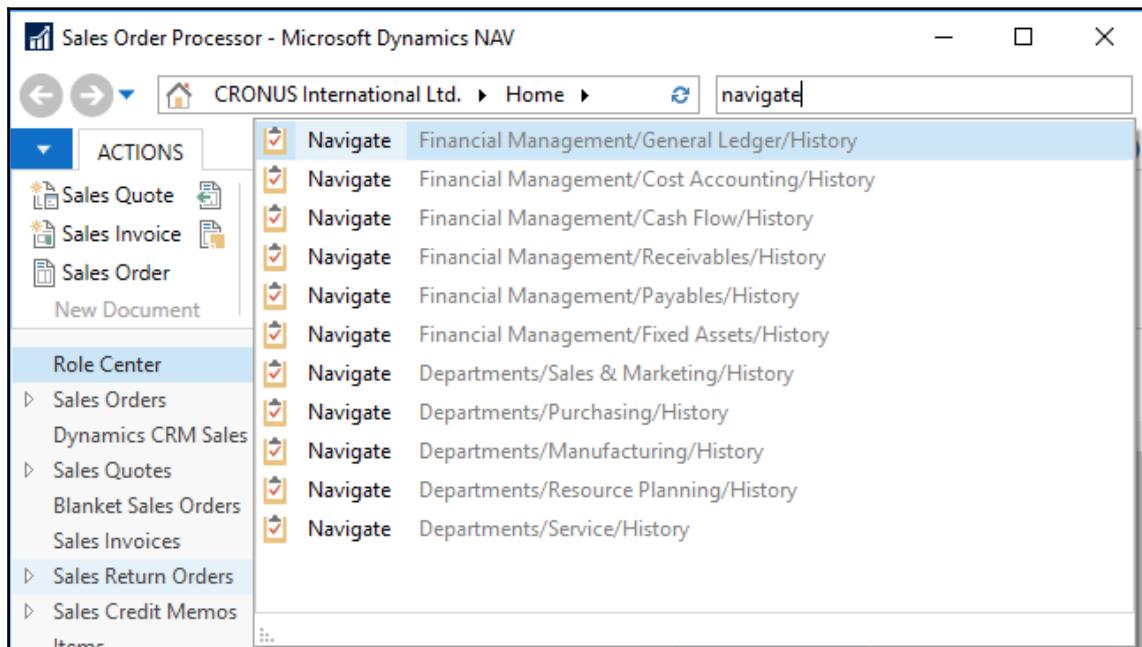
```
775 | "Line Discount Amount" :=
776 |     ROUND(
777 |         ROUND(Quantity * "Unit Price",Currency."Amount Rounding Precision") *
778 |             "Line Discount %" / 100,Currency."Amount Rounding Precision");
```

As we can see, before creating any modification that has money fields, we must familiarize ourselves with the NAV currency conversion feature, the code that supports it, and the related setups. A good place to start is the C/AL code in Table 37 - Sales Line, Table 39 - Purchase Line, and Table 330 - Currency Exchange Rate.

## Navigate

Navigate is an underappreciated tool for both the user and the developer. Our focus here is on its value to the developer. Navigate (Page 344) finds and displays the counts 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. This is a handy tool for the developer as it can show the results of posting activity and provide the tools to drill into the detail of all those results. If we add new transactions and ledgers as part of an enhancement, our Navigate function should be enhanced to cover them too.

Navigate can be called from the **Navigate** action, which appears in a number of places in the **Departments Tasks** menu and Role Center ribbons. From anywhere within NAV, the easiest way to find Navigate is to type the word into the **Search** box (see the next screenshot):



If we invoke the **Navigate** page using the menu action item, we must enter the **PostingDate** and **DocumentNumber** for the entries we wish to find. Alternately, we 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 **PostingDate + DocumentNo.** option is 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, we can just highlight a record and click on **Navigate** to search for all the related entries. In the following example, a Posted Invoice is highlighted:

Posted Sales Invoices							Type to filter (F3)	No.	▼	→
No.	Sell-to Customer...	Sell-to Customer Name	Currency Code	Amount	Amount Including VAT	Location Code	No filters applied			
103001	10000	The Cannon Group PLC	EUR	7,438.50	8,182.35	BLUE				
103002	20000	Selangorian Ltd.	EUR	6,337.98	6,971.78					
103003	30000	John Haddock Insurance Co.	EUR	5,454.00	5,999.40					
103005	49525252	Beef House	EUR	1,449.70	1,449.70	GREEN				
103006	49525252	Beef House	EUR	7,248.48	7,248.48	GREEN				
103007	49525252	Beef House	EUR	1,039.85	1,039.85	GREEN				

After clicking the **Navigate** action, the page will pop up filled in, as shown in the following screenshot:

Edit - Navigate - Selected - Posted Sales Invoice

CRONUS International Lt... ?

HOME ACTIONS

Show Related Entries Find Print... Find by Document Find by Business Contact Find by Item Reference Refresh Find

Process Find By Page

**Document**

Document No.: 103001 Posting Date: 22-01-18

Related Entries	No. of Entries
Posted Sales Invoice	1
G/L Entry	5
VAT Entry	2
Cust. Ledger Entry	1
Detailed Cust. Ledg. Entry	1
Res. Ledger Entry	2

**Source**

Document Type: Posted Sales Invoice Source No.: 10000  
Source Type: Customer Source Name: The Cannon Group PLC

OK

Had we accessed the **Navigate** page through one of the menu entries we found through **Search** (*Ctrl + F3*), we would have filled in the **DocumentNo.** and **PostingDate** fields and clicked on **Find**. As we can see here, the **Navigate** page shows a list of related, posted entries (one of which will include the entry we highlighted when we invoked the **Navigate** function). If we highlight one of the lines in the **Related Entries** list, then click on the **Show Related Entries** icon at the top of the page; we will see an appropriately formatted display of the chosen entries.

If we highlight the **G/L Entry** table entries and click on **Show**, we will see a result like the following screenshot. Note that all the **G/L Entry** are displayed for the same **Posting Date** and **Document No.**, matching those specified at the top of the **Navigate** page:

The screenshot shows the Microsoft Dynamics NAV interface for 'Posted Sales Invoices'. The main title bar says 'Posted Sales Invoices - Microsoft Dynamics NAV'. The ribbon has 'HOME' selected. The left navigation pane shows 'General Ledger Entries' under 'Postings' and 'Source Documents'. The main area displays a grid of 'General Ledger Entries' with columns: Post. Date, Document Type, Document No., G/L Acco..., Description, Gen. Posting..., Gen. Bus. Posting ..., Gen. Prod. Posting ..., Amount, Bal. Accou.., Bal. Accou.., and Entry No. A filter bar at the top right shows 'Filter: 103001 > 22-01-18'. Below the grid, there's a 'Incoming Document Files' section with a table showing file details: Name, Type, and a preview area. At the bottom, there's a footer with 'CRONUS International Ltd.' and a timestamp 'donderdag 25 januari 2018 F11ZG72:LUC VAN VUGT'.

Post. Date	Document Type	Document No.	G/L Acco...	Description	Gen. Posting...	Gen. Bus. Posting ...	Gen. Prod. Posting ...	Amount	Bal. Accou..	Bal. Accou..	Entry No.
22-1-2018	Invoice	103001	6910	Invoice 103001	Sale	DOMESTIC	SERVICES	391,50	G/L Account		2727
22-1-2018	Invoice	103001	6410	Invoice 103001	Sale	DOMESTIC	SERVICES	-7.830,00	G/L Account		2729
22-1-2018	Invoice	103001	5611	Invoice 103001				-783,00	G/L Account		2730
22-1-2018	Invoice	103001	5611	Invoice 103001				39,15	G/L Account		2728
22-1-2018	Invoice	103001	2310	Invoice 103001				8.182,35	G/L Account		2731

## Modifying for Navigate

If our modification creates a new table that will contain posted data and the records contain both **Document No.** and **Posting Date** fields, we can include this new table in the **Navigate** function.

The C/AL Code for Navigate functionality based on **PostingDate + DocumentNo.** is found in the `FindRecords` and `FindExtRecords` functions of Page 344 - Navigate. The following screenshot shows the segment of the Navigate CASE statement code for the `CheckLedgerEntry` table:

```
596 IF CheckLedgEntry.READPERMISSION THEN BEGIN  
597   CheckLedgEntry.RESET;  
598   CheckLedgEntry.SETCURRENTKEY("Document No.", "Posting Date");  
599   CheckLedgEntry.SETFILTER("Document No.", DocNoFilter);  
600   CheckLedgEntry.SETFILTER("Posting Date", PostingDateFilter);  
601   InsertIntoDocEntry(  
602     DATABASE::"Check Ledger Entry", 0, CheckLedgEntry.TABLECAPTION, CheckLedgEntry.COUNT);  
603 END;
```

The code checks `READPERMISSION`. If that permission 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 we wish to add a new table to the `Navigate` function, we must replicate this functionality for our new table.

In addition, we 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, modeling on the applicable section of code in this function (that is, choose the code set that best fits our new tables). Making a change like this, when appropriate, will not only provide a powerful tool for our users, but also provide a powerful tool for us as developers.

## Debugging in NAV 2017

In general, the processes and tools we use for debugging can serve multiple purposes. The most immediate purpose is always that of identifying the causes of errors and then resolving those errors. There are two categories of production errors, which may also occur during development, and NAV 2017's **Debugger** module is very well suited to addressing both of these. The NAV debugger smoothly integrates developing in the Development Environment and testing in the **Role Tailored Client (RTC)**.

The first category is the type that causes an error condition that terminates processing. In this case, the immediate goal is to find the cause and fix it as quickly as possible. The debugger is an excellent tool for this purpose. The second category is the type that, while running to completion successfully, gives erroneous results.

We often find that debugging techniques can be used to help us better understand how NAV processes work. We may be working on the design of (or determination of the need for) a modification, or we may simply want to learn more about how a particular function is used or outcome is accomplished in the standard NAV routines. It would be more appropriate to call these efforts analysis or self-education rather than debugging, even though the processes we use to dissect the code and view what it's doing are very similar. In the course of these efforts, less sophisticated approaches are sometimes useful in understanding what's going on. We'll quickly review some of those alternate approaches before studying use of the NAV 2017 Debugger.

## Text Exports of Objects

Using a developer license, we are allowed to export objects into text files where we can use a text editor to examine or even manipulate the result. Let's take a look at an object that is exported into text and imported into a favorite text editor. We will use one of the tables that are part of our WDTU development, the Playlist Item Rate table, 50004, as shown in the following screenshot:

```
OBJECT Table 50004 Playlist Item Rate
{
    OBJECT-PROPERTIES
    {
        Date=04/10/15;
        Time=11:46:50 AM;
        Modified=Yes;
        Version List=C002;
    }
    PROPERTIES
    {
        LookupPageID=Page50005;
        DrillDownPageID=Page50005;
    }
    FIELDS
    {
        { 10 ; ;Type ;Option ;TableRelation=IF (Type=CONST(Vendor)) Vendor.No.
          ELSE IF (Type=CONST(Customer)) Customer.No. ;
          OptionCaptionML=ENU=Vendor,Customer;
          OptionString=Vendor,Customer;
          Description=Vendor,Customer }

        { 20 ; ;No. ;Code20 }
        { 30 ; ;Item No. ;Code20 }
        { 40 ; ;Start Date ;Date }
        { 50 ; ;End Date ;Date }
        { 60 ; ;Rate Amount ;Decimal }
        { 70 ; ;Publisher Code ;Code20 ;Description=TableRelation:Publisher.Code }

    }
    KEYS
    {
        { ;Type ;Clustered=Yes }
    }
    FIELDGROUPS
    {
    }
    CODE
    {
    }
    BEGIN
    END.
}
```

The general structure of all exported objects is similar, with differences that we would expect for the different objects. This particular table contains no C/AL-coded logic, as those statements would be visible in the text listing. We can see by looking at this table object that we could easily search for instances of the `Code` string throughout the text export of the entire system, but it would be more difficult to look for references to the Playlist Item Rates because it is only referenced by page ID, `Page50005`. While we can find the instances of `Code` with our text editor, it would be quite difficult to differentiate those instances that relate to the Playlist Item Rate table from those for any other table. This includes those that have nothing to do with our WDTU system enhancement, as well as those simply defined in an object as Global Variables.

If we are determined to use a text editor to find all instances of "Playlist Item Rate", "Rate Amount", we can do the following:

- Rename the field in question to something unique. C/SIDE will rename all the references to this field to this new name.
- Export all the sources to text followed by using our text editor (or even Microsoft Word) to find the new, unique name.
- Either return the field in the database to the original name or work in a temporary copy of the database, which we will then discard. Otherwise, we will have quite a mess.

One task that needs to be done occasionally is to renumber an object or to change an internal object reference that refers to a no longer existing element. The C/SIDE editor may not let us do that easily or, in some cases, not at all. In such a case, the best answer may be to export the object into text, make the change there, and then import it back in as modified.



Be careful. When we import a text object, C/SIDE does not check to see if the incoming object is valid. C/SIDE does that checking when we import a compiled `.fob` object.

If we do object renumbering, we should use the functionality built into **Mergetool** (available at [www.mergetool.com](http://www.mergetool.com)). Many years ago, Mergetool became the recommended upgrade support tool for NAV and, in many instances, it's still the best answer.

There are occasions when it is very helpful to simply view an object flattened out in text format. In a report or XMLport where we may have combinations of logic and properties, the only way to see everything at once is in text format. We can use any text editor we like, Notepad, Word, or one of the visual programming editors; the exported object is just text. We need to cope with the fact that when we export a large number of objects in one pass, they all end up in the same text file. This 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. You can achieve this by either using the `Split-NAVApplicationObjectFile` PowerShell cmdlet or one of the 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).

## Dialog function debugging techniques

Sometimes, the simpler methods are more productive than the more sophisticated tools, because we 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. However, we should remember that none of these techniques conform to the **Testing Best Practices** in the Help topic: *Testing the Application*. These should only be used when a quick one-time approach is needed, or when recommended testing practices simply won't easily provide the information needed and one of these techniques will do so.

## Debugging with MESSAGE and CONFIRM

The simplest debug method is to insert MESSAGE statements at key points in our logic. This is very simple and, if structured properly, provides us with a simple trace of the code logic path. We can number our messages to differentiate them and display any data (in small amounts) as part of a message like the following:

```
MESSAGE ('This is Test 4 for %1',Customer."No.");
```

A big disadvantage is that MESSAGE statements do not display until processing either terminates or is interrupted for user interaction. Also, if you create a situation that generates hundreds of messages, you will find it quite painful to click through them individually at process termination.

If we force a user interaction at some point, then our accumulated messages will appear prior to the interaction. The simplest way to force user interaction is to issue a CONFIRM message in the format as follows:

```
IF CONFIRM ('Test 1', TRUE) THEN;
```

If we 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), we could use CONFIRM statements for all the messages. The operator must then respond to each one before our program will move on but, sometimes, that is what we want. However, if we make the mistake of creating the situation where hundreds of messages are generated, the operator will have to respond to each one individually in what could be a very time consuming and inefficient process.

## Debugging with DIALOG

Another tool that is useful for progress tracking is the DIALOG function. DIALOG is usually set up to display a window with a small number of variable values. As processing progresses, the values are displayed in real time. The following are a few ways we can use this:

- Simply tracking progress of processing through a volume of data. This is the same reason we would provide a DIALOG display for the benefit of the user. The act of displaying slows down processing somewhat, so we 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 we may gauge the relative speed of progress through several steps.
- We might have a six-step process to analyze. We could define six tracking variables and display all of them in the DIALOG. We would initialize each variable with values dependent on what we are tracking, such as A1, B2000, C300000, and so on. At each process step, update and display the current state of one or all of the variables. This can be a very helpful guide for how our process is operating. To slow things down, we could put a SLEEP (100) or SLEEP (500) after the DIALOG statement (the number is milliseconds of delay).

## Debugging with text output

We 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. We need to either do this in single user mode or make it multiuser by including the `USER ID` on every entry.

This technique allows us 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. We can even timestamp our output records so that we can use this method to look for processing speed problems.

Following the test run, we can analyze the results of our test more quickly, than if we were using displayed information. We 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 we identify the variables or code segments of most interest. We can combine this approach with the following approach using the `ERROR` statement, if we 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, we need to test several times using the same data, but the test changes the data. If we have a small database, we can always back up the database and start with a fresh copy each time. However, that can be inefficient and, if the database is large, impractical. If we are using the built-in NAV Test functions, we can roll back any database changes so the tests are totally repeatable. Another alternative is to conclude our test with an `ERROR` function to test and retest with exactly the same data.

The `ERROR` function forces a runtime 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 our debugging information is provided by using the Debugger or any of the `DIALOG` functions just mentioned prior to the execution of the `ERROR` function. If we are using `MESSAGE` to generate debugging information, we 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 our testing validation is dependent on checking results using `Navigate` or our test is a multistep process, such as order entry, review, and posting. In this latter case, only use of the built-in Test functions (creating Test Runner Codeunits, and so on.) will be adequate. However, in some situations, use of the `ERROR` function is a very handy technique for repeating a test with minimal effort.

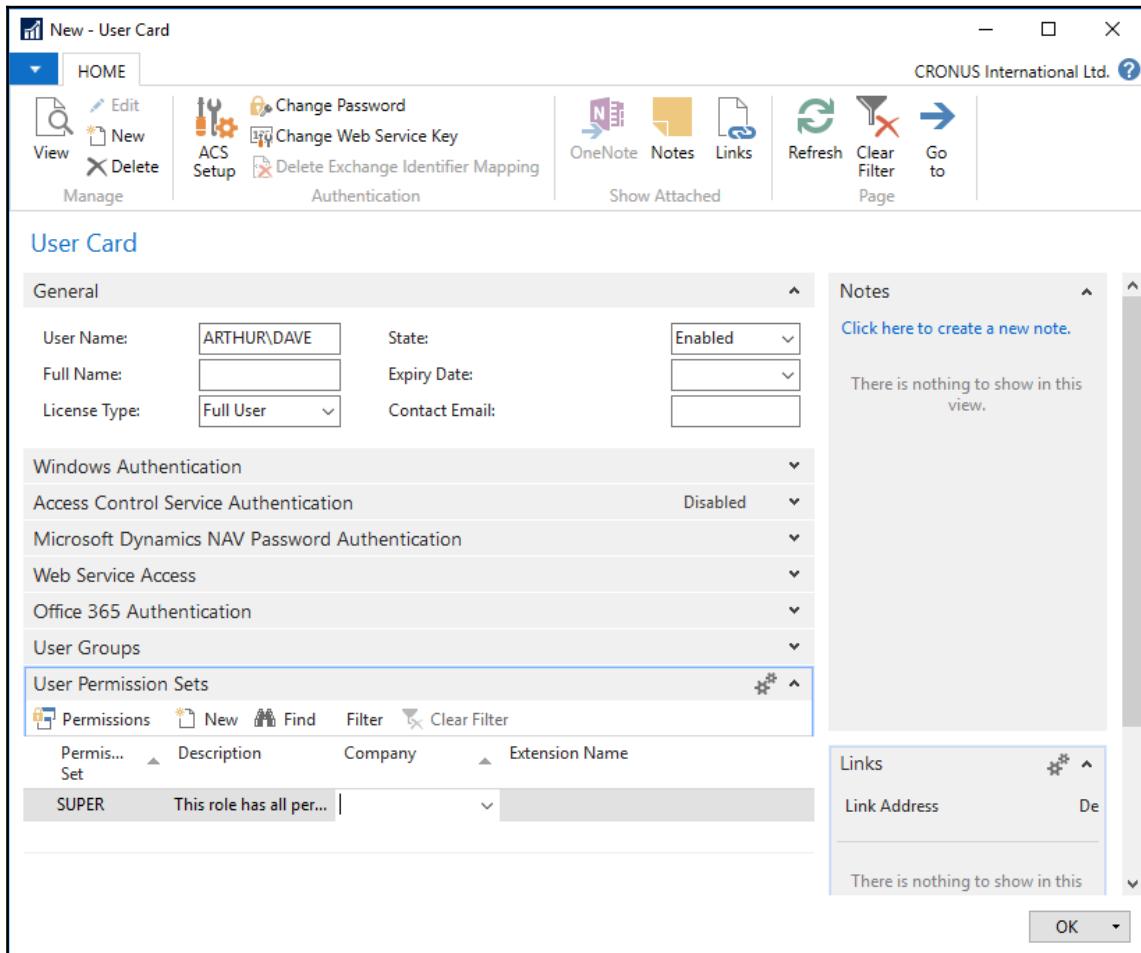
When testing just the posting of an item, it often works well to place the test-concluding `ERROR` function just before the point in the applicable `Posting` codeunit where the process would otherwise be completed successfully. In order for the `Rollback` function to be effective, we must make sure there aren't any `COMMIT` statements included in the range of the code being tested.

## The NAV 2017 Debugger

As defined in the **Debugging Help** (search on NAV2017 Help Debugging - you should study this section), debugging is the process of finding and correcting errors. NAV 2017 has a powerful built-in debugger. The user interface for the NAV 2017 Debugger is written in C/AL. The Debugger objects can be identified by filtering in the Object Designer, All objects, on `*Debug*`. Reviewing the structure of the Debugger objects in C/SIDE may help better understand its inner workings.

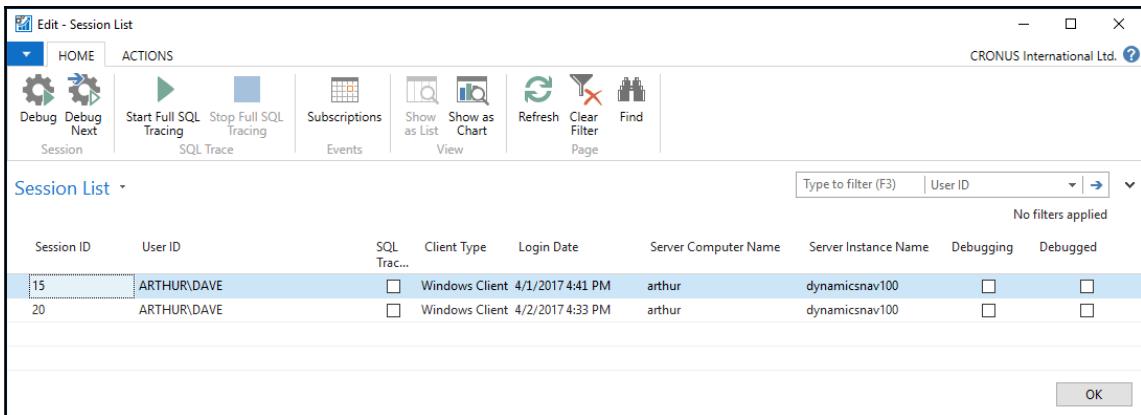
The Debugger can be activated in multiple different ways including from within the Development Environment, from within the RTC, from a command line and by means of a C/AL function. The latter two options attach to a session at the same time as they activate. The best choice for activation method depends on the specific situation and the debugging technique being utilized by the developer.

Only a user who has SUPER permissions for all companies is allowed to activate the Debugger. The **user permissions** setup should have an empty **Company** field, as we see in the circled space in the following screenshot:



## Activating the Debugger

Activating the Debugger from the Development Environment is a simple matter of clicking on **Tools** | **Debugger** | **Debug Sessions** (or *Shift + Ctrl + F11*). The initial page that displays when the Debugger is activated will look like the following screenshot (typically, with each session having a different User ID):



The screenshot shows the 'Edit - Session List' window. At the top, there are tabs for 'HOME' and 'ACTIONS'. Under 'ACTIONS', there are buttons for 'Session' (Debug, Debug Next), 'SQL Trace' (Start Full SQL Tracing, Stop Full SQL Tracing, SQL Trace), 'Events' (Subscriptions, Show as List, Show as Chart, View), and 'Find' (Refresh, Clear Filter, Page). A search bar at the top right contains 'Type to filter (F3)' and 'User ID'. Below the search bar, it says 'No filters applied'. The main area is titled 'Session List' with a dropdown arrow. It displays a table with columns: Session ID, User ID, SQL Trac..., Client Type, Login Date, Server Computer Name, Server Instance Name, Debugging, and Debugged. Two rows are shown:

Session ID	User ID	SQL Trac...	Client Type	Login Date	Server Computer Name	Server Instance Name	Debugging	Debugged
15	ARTHUR\DAVE	<input type="checkbox"/>	Windows Client	4/1/2017 4:41 PM	arthur	dynamicsnav100	<input type="checkbox"/>	<input type="checkbox"/>
20	ARTHUR\DAVE	<input type="checkbox"/>	Windows Client	4/2/2017 4:33 PM	arthur	dynamicsnav100	<input type="checkbox"/>	<input type="checkbox"/>

An 'OK' button is located at the bottom right of the table.

Note that only one session per service tier at a time can be debugged.



If we activate the Debugger by means of any method that does not specify a session, this same screen will appear. The Debugger can also be activated from within the RTC as follows:

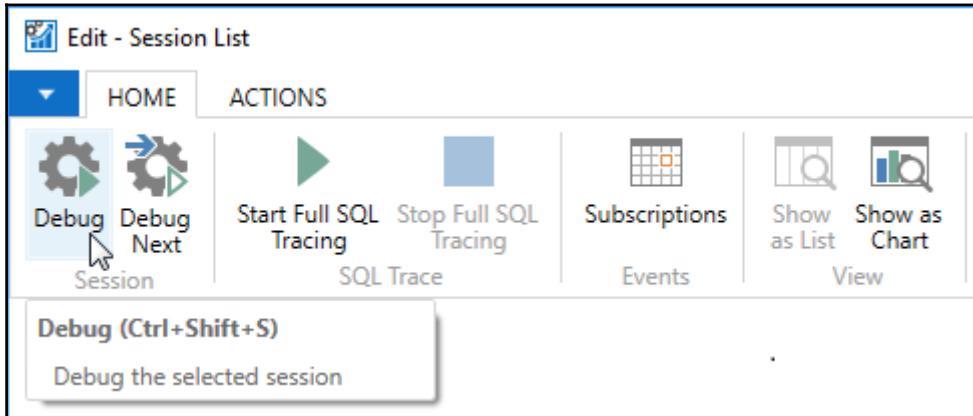
1. Enter **Sessions** in the Search box.
2. Select the link displayed (**Administration/IT Administration/General**).
3. In the **General** section, click on **Sessions**.

We can also get to this same point by clicking on the **Departments** button in the Navigation pane. Then click on **IT Administration | General**, followed by **Sessions** in the **Tasks** section.

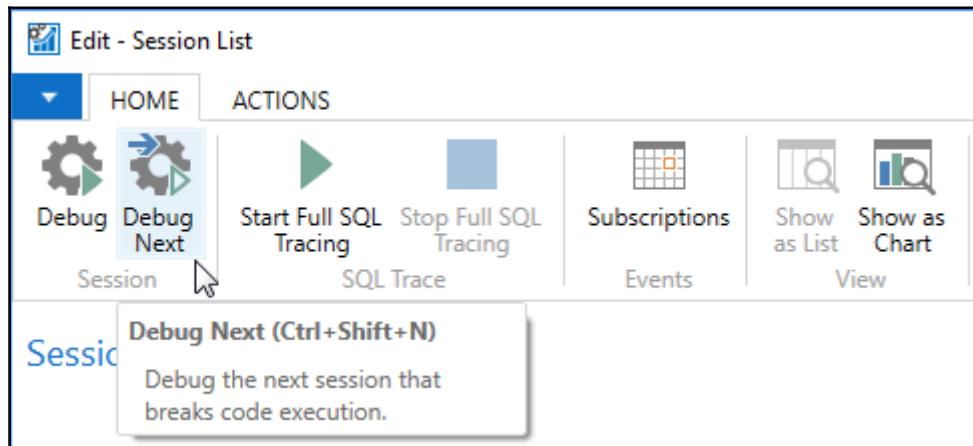
However we activate it, the Debugger runs as a separate independent session that can be attached to an operating session. The Help *Activating the Debugger* describes activating the Debugger to debug a Web Service. The Help *Configuring NAS Services* has information about using the Debugger with an NAS Service.

## Attaching the Debugger to a Session

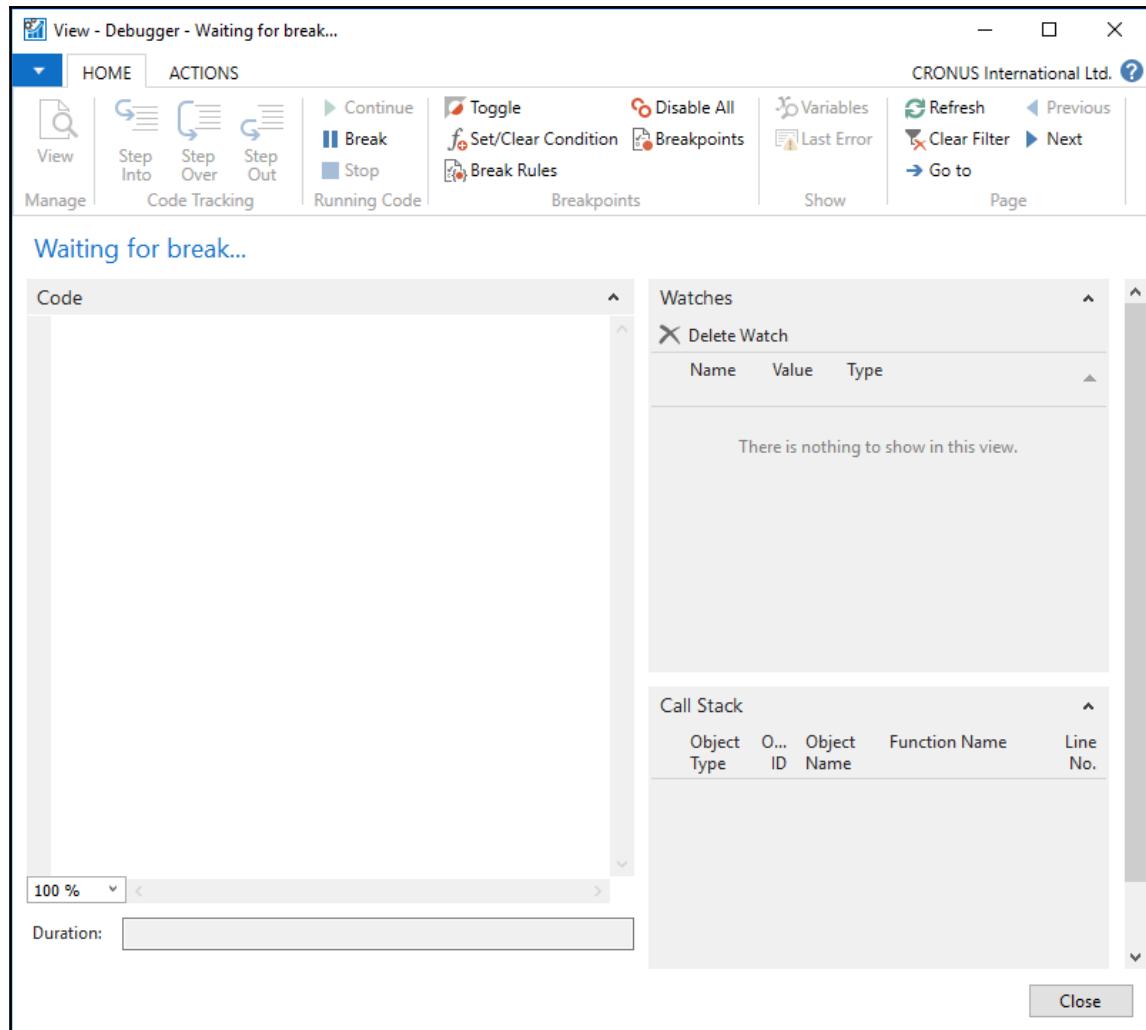
From the **Edit-Session list** screen, we have two options to attach the Debugger session to a session. One way is to highlight a Session and then click on the **Debug** icon, as shown in the following screenshot:



The other way is to click on the **Debug Next** icon, then initiate a new Session, as shown in the following screenshot. The Debugger will be attached to the new Session:



When we click on **Debug Next**, an empty **View - Debugger** page will open, awaiting the event that will cause a break in processing and the subsequent display of Code detail (**Code**), Watched variables (**Watches**), and the **Call Stack**:



## Creating Break Events

Once the Debugger is activated and attached to a session, a break event must occur to cause the debug trace and associated data display to begin. Break events include (but are not limited to) the following:

- An error occurs that would cause object execution to be terminated
- A previously set **Breakpoint** is reached during processing
- The record is read when the **Break on Record Changes** **Break Rule** is active

- The **Break** icon in the Running Code group is clicked in the ribbon of the **View - Debugger** page
- A **Breakpoint Condition**, which has been set in the Breakpoints group in the ribbon of the **View -Debugger** page, is satisfied during processing

Of the preceding events, the two most common methods of starting up a debug trace are the first two, an error or reaching a previously set breakpoint. If, for example, an error condition is discovered in an operating object, the debugging process can be initiated by:

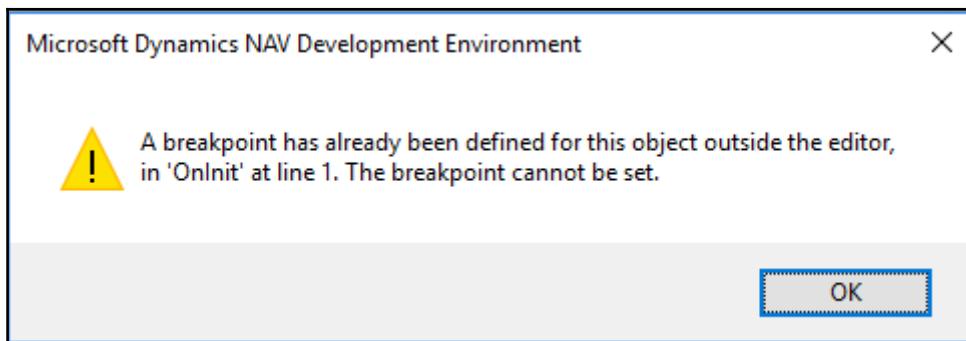
- Activating the debugger
- Attaching the debugger session to the session where the error will occur
- Running the process where the error occurs

When the error occurs, the page parts (**Code**, **Watches**, and **Call Stack**) in the debug window will be populated, and we can proceed to investigate variable values, review code, and so forth.

Breakpoints are stopping points in an object that was set by the developer. Breakpoints can be set in a variety of ways, including in the Development Environment, in the **View-Debugger Code** page, and in the **Edit - Debugger Breakpoint List**.

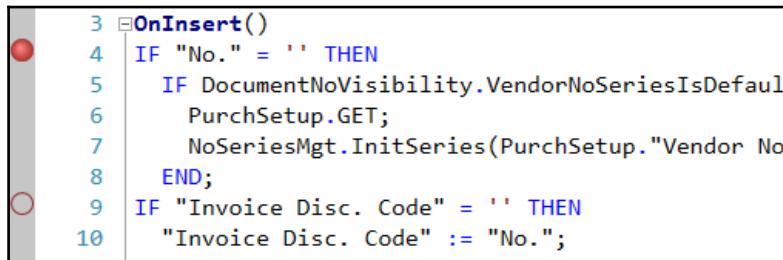
While the latter two locations to set breakpoints may be very useful while we are in the middle of a debugging session, those breakpoints only display while the Debugger is active. Once we exit the debugging session, those breakpoints that were set in the Debugger will disappear from view, while the breakpoints that were set from within the applicable C/SIDE Designer will remain visible and available for use until removed by a developer.

The result may be somewhat confusing because we can only see all of the breakpoints when we are in the Debugger. If we try to set a breakpoint in the Development Environment and a breakpoint has already been set on that line of code while in the Debugger, we will get the following error message:



For this reason, it may be a better practice to set all our planned testing breakpoints in the Development Environment. When we set breakpoints within the Debugger, we should clear them before ending our test session. Otherwise, we may later run into breakpoints we didn't remember existed and that we can't see in the Designers.

Active breakpoints are represented in code by a filled-in circle. Disabled breakpoints are represented by an empty circle. Examples are shown in the following code screenshot:

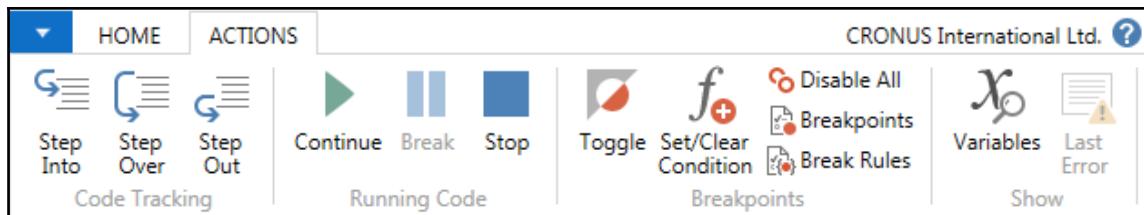


The screenshot shows a portion of C/AL code in a code editor. Line 3 contains an active breakpoint (filled red circle) and a comment OnInsert(). Lines 4 through 10 show standard C/AL logic for handling vendor numbers and invoice discounts. Line 4 has an empty circle, indicating it is a disabled breakpoint.

```
3  ⚡OnInsert()
4  IF "No." = '' THEN
5    IF DocumentNoVisibility.VendorNoSeriesIsDefault
6      PurchSetup.GET;
7      NoSeriesMgt.InitSeries(PurchSetup."Vendor No."
8    END;
9  IF "Invoice Disc. Code" = '' THEN
10   "Invoice Disc. Code" := "No.;"
```

## The Debugger window

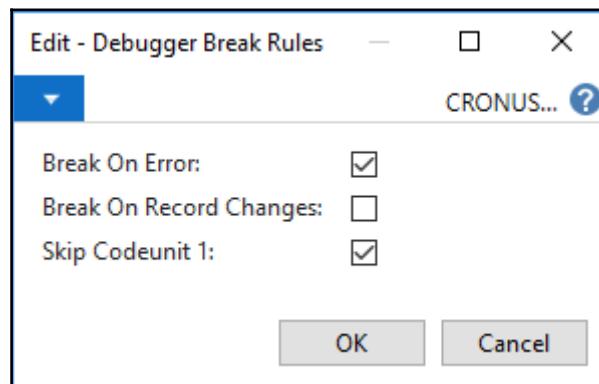
When viewing C/AL code in a Designer, breakpoints can be set, disabled, or removed by pressing the *F9* key. When viewing C/AL code in the Code window of the Debugger, breakpoints can only be set or removed by pressing the *F9* key or clicking on the Toggle icon. Other Debugger breakpoint controls are shown in the following screenshot:



### Ribbon Actions:

- **Continue:** This continues processing until the next break.
- **Step Out:** This is designed to complete the current function without stopping, and then break.
- **Step Over:** This is designed to execute a function without stopping, then break.

- **Step Into:** This is designed to trace into a function.
- **Break:** This breaks at the next statement.
- **Stop:** This stops the current activity but leaves the debugging session active.
- **Toggle:** This sets or clears a breakpoint on the current line.
- **Set/Clear Condition:** This sets or clears a conditional (based on C/AL expression) breakpoint at the current line.
- **Disable All:** This disables all checkpoints in the attached session.
- **Break Rules:** This displays the following screen:



The **Break Rules** options work as follows:

- **Break On Error** default is on
  - If **Break On Record Changes** is on when a debug session is attached to an operating session, the debugging will start immediately
  - **Skip Codeunit 1** default is on, allowing all the Codeunit 1 processing to normally be processed without tracing
- **Breakpoints:** This displays a list of the active breakpoints and provides action options to enable, disable, or delete breakpoints individually or in total.
- **Variables:** This displays the Debugger Variable List where we can examine the status of all variables that are in scope. Additional variables can be added to the Watch list here, as shown in the following screenshot:

The screenshot shows the 'View - Debugger Variable List' window. At the top, there's a ribbon with 'HOME' selected and 'ACTIONS' tab. Below the ribbon are buttons for 'Add Watch' (highlighted with a cursor), 'Show as List', 'Show as Chart', 'Refresh', 'Clear Filter', and 'Find'. A tooltip for 'Add Watch (Ctrl+Ins)' says 'Add the selected variable to the watch list.' On the right, it says 'No filters applied'. The main area is a table with columns 'Name', 'Value', and 'Type'. It lists variables under '<Globals>': 'Rec' (Table Customer (18)) with value '01121212'; 'xRec' (Table Customer (18)) with value '' (empty string); 'CustomizedCalEntry' (Table Customized Calendar Entry ...) with value '<Uninitialized>'; and 'CustomizedCalendar' (Table Customized Calendar Chan...) with value '<Uninitialized>'. A 'Close' button is at the bottom right.

Variables can be removed from the Watch list in the Debugger Watches page part

- **Last Error:** This displays the last error message shown by the session being debugged.



Have you noticed the **Duration** field at the bottom of the **Code** pane? This displays the execution time between the previous breakpoint, or the starting of the debugger, and the current breakpoint.

There are quite a number of valuable Help sections on use of the Debugger, including these and many others:

- *Debugging*
- *Debugger Keyboard Shortcuts*
- *Breakpoints (this one is especially good)*
- *Closing the Debugger*
- *How to: Add Variables to the Watches FactBox*

- *How to: Debug a Background Session*
- *How to: Manage Breakpoints from the Development Environment*
- *How to: Set Conditional Breakpoints*
- *Walkthrough: Debugging the Microsoft Dynamics NAV Windows Client*

## Changing code while debugging

While a debugger session is active, we can open the object being debugged in an appropriate Designer, change the object, save, and recompile it. The revised object will immediately be available to other sessions on the system. However, the version of the object that is being executed and in view in the debugger is the old version of the object, not the changed one. Furthermore, if we refresh the view of the code in the **Debugger Code** window, the new version will be displayed while the old version continues to be executed, leaving potential for significant confusion. Therefore, it's best not to change an object and continue to debug it without starting a new session.

## C/SIDE Test-Driven Development

NAV 2017 includes the enhanced C/AL Testability feature set designed to support C/AL test-driven development. **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.

Advantages of test-driven development include the following:

- Designing testing processes in conjunction with functional design
- Finding bugs early
- Preventing bugs reaching production
- Enabling regression testing which prevents changes from introducing new bugs into previously validated routines

The C/AL Testability feature provides test specific types of Codeunits—Test Codeunits and Test Running Codeunits. Test Codeunits contain Test methods, UI handlers, and C/AL code to support Test methods including the `AssertError` function. Test Runner Codeunits are used to invoke Test Codeunits, for 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. The two triggers are:

- `OnBeforeTestRun`: This is called before each test. It allows defining, via a Boolean, whether or not the test should be executed.
- `OnAfterTestRun`: This is called when each test completes and the test results are available. It 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 following:

- The ability to run suites of application tests both in automated mode and in regression tests:
  - **Automated** means that a defined series of tests could be run and the results recorded, all without user intervention.
  - **Regression testing** 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 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.
- The **TestIsolation** property of TestRunner Codeunits allow tests to be run, then all database changes are rolled back so that no changes are Committed. After a test series in this mode, the database is the same after the test as it was before the test.
- 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 is supported. **Positive testing** looks for a specific result, a correct answer. **Negative testing** checks 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 Test, Handler, 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 handlers that are designed to handle User Interface events without the need for a user to intervene:

- **MessageHandler:** This handles the MESSAGE statement
- **ConfirmHandler:** This handles CONFIRM dialogs
- **StrMenuHandler:** This handles STRMENU menu dialogs
- **PageHandler:** This handles Pages that are not run modally
- **ModalPageHandler:** This handles Pages that are run modally
- **ReportHandler:** This handles Reports
- **RequestPageHandler:** This handles the Request Page of a specific Report
- **FilterPageHandler:** This handles Filter Pages generated by a FilterPageBuilder data type
- **HyperlinkHandler:** This handles hyperlinks that are passed to the HYPERLINK function
- **SendNotificationHandler:** This handles the new NAV 2017 Notifications
- **RecallNotificationHandler:** This handles the recall of the new NAV 2017 Notifications

The C/SIDE Test Driven Development approach should proceed along the following steps:

1. Define an application function specification.
2. Define the application technical specification.
3. Define the testing technical specification including both Positive and Negative tests.
4. Develop Test and Test Running codeunits (frequently, only one or a few Test Running codeunits will be required).
5. Develop Application objects.
6. As soon as feasible, run Application object tests by means of the Test Running codeunits, and log test results for historical and analytical purposes.

7. Continue the development-testing cycle, updating the tests and the application as appropriate throughout the process.
8. 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.

With the latest product media, Microsoft released a full set of approximately 16.0000 regression tests written by Microsoft for NAV 2017 using the NAV Testability tools. These are the tests that the NAV product developers used to validate their work. The number of tests that apply to a specific situation depends on the local version and specific features involved. Make sure your license is updated too.

Included are the regression tests and various tools to manage and execute tests built on top of the testability features released for Microsoft Dynamics NAV. Also included is a coverage tool and guidance documentation to create our own tests and integrate those with the Microsoft provided tests. This allows us to do full regression testing for large modifications and ISV solutions.

## Other interfaces

Some NAV systems must communicate with other software or even with hardware. Sometimes, that communication is Inside-Out (that is, initiated by NAV) and 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 will need at least a basic understanding of the interface tools that are part of NAV.



Because of NAV's unique data structures and the critical business logic embedded therein, it is very risky for an external system to access NAV data directly via SQL Server without using C/AL-based routines as an intermediary.

NAV has a number of methods of interfacing with the world outside its database. We will review those very briefly here. To learn more about these, we should begin by reviewing the applicable material in the online Developer and IT Pro Help material, plus any documentation available with the software distribution. We should also study sample code, especially that in the standard system as represented by the **Cronus Demonstration Database**. And, of course, we should take advantage of any other resources available, including the NAV-oriented Internet forums and blogs.

## 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.

Only non-visual controls are supported through this interface; this doesn't mean we can't figure out a work-around, just that they aren't supported by Microsoft. The Client Add-in feature, discussed later in this chapter, provides visual interface capability through another integration approach.

We cannot use an Automation Controller defined COM component as a control on an NAV Page object. Only client-side automation objects are supported. This is because the NAV server tier operates in 64-bit mode and many COM objects are not compatible with 64-bit operating systems. Instead of server-side automation objects, use Microsoft .NET interoperability functionality (for more information, search for Help on Interoperability).

Some common uses of Automation Controller interfaces are as follows:

- Populating Word template documents to create more attractive communications with customers, vendors, and prospects (for example, past due notices, purchase orders, promotional letters, and so on)
- Moving data to Excel spreadsheets for manipulation (for example, last year's sales data to create this year's projections)
- Moving data to and from Excel spreadsheets for manipulation (for example, last year's financial results out and next year's budgets back in)
- Using 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

It will also be helpful to review the information on this topic in the following Help sections:

- *COM Overview*
- *Best Practices for Using Automation With the Microsoft Dynamics NAV Windows Client*
- *Automation Data Type*
- *Using COM Technologies in Microsoft Dynamics NAV*

## Linked Data Sources

The two table properties, **LinkedObject** and **LinkedInTransaction**, are available for NAV tables. Use of these properties in the prescribed fashion allows data access, including views, in linked server data sources such as Excel, Access, other instances of SQL Server, and even an Oracle database. For additional information, see the Help section, *Using Linked Objects and Accessing Objects in Other Databases or on Linked Servers*. This is one way to integrate NAV with external applications in a way that is seamless for the users.

## NAV Application Server

Microsoft Dynamics NAV Application Server (NAS) is a middle-tier server component that executes business logic without a user interface or user interaction. In NAV 2017, NAS is one of the client services that runs in the Microsoft Dynamics NAV Server.

The NAS is essentially an automated user client. Because NAS is effectively a non-UI version of the standard NAV client module, it can access all of NAV's business rules.

Error messages that are generated by an NAS process are logged in the Event Viewer.

NAS operates essentially the same as any other NAV Windows client. If the NAS setup to run the **Task Scheduler** (formerly called Job Queue), it processes requests in the queue one at a time, in the same manner as the GUI client. Therefore, as developers, we need to limit the number of concurrent calls to an NAS instance as the queue should remain short to allow timely communications between interfaces. In NAV 2017, multiple background sessions can be started from client sessions. This provides opportunities for the creative designer/developer to utilize NAV automation.

## Client Add-ins

The NAV 2017 Client Add-in API (also known as Client Extensibility) provides the capability to extend the RTC for Windows, Web, or Tablet through the integration of external non-NAV controls. The Client Add-in API uses .NET interfaces as the binding mechanism between a control add-in and the NAV framework. Different interfaces and base classes are available to use, or a custom interface can be created. Controls can be designed to raise events that call on the `OnControlAddin` trigger on the page field control that hosts the add-in. They can also add events and methods that can be called from within C/AL.

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 native NAV controls. The following are a few simple examples of how Client Add-ins might be used to extend RTC UI behavior:

- A NAV text control that looks normal but offers a special behavior, but when the user double-clicks on it, the field's contents will display in a popup screen accompanied by other related information or even a graphical display.
- A dashboard made up of several dials or gauges showing the percentage of chosen resources relative to target limits or goals. The dials are defined to support click and drill into the underlying NAV detail data.
- An integrated sales call mapping function displays 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, showing the number of entries at each stage, and supporting filtering to display selected sets of entries.
- Entry and storage of a written document signature on a touch screen.

## **Client Add-in construction**

Some Client Add-ins will be created, packaged, and distributed by ISV Partners who specialize in an application area. When enhancing a system for a customer's specific application, we may decide to create a special purpose add-in.

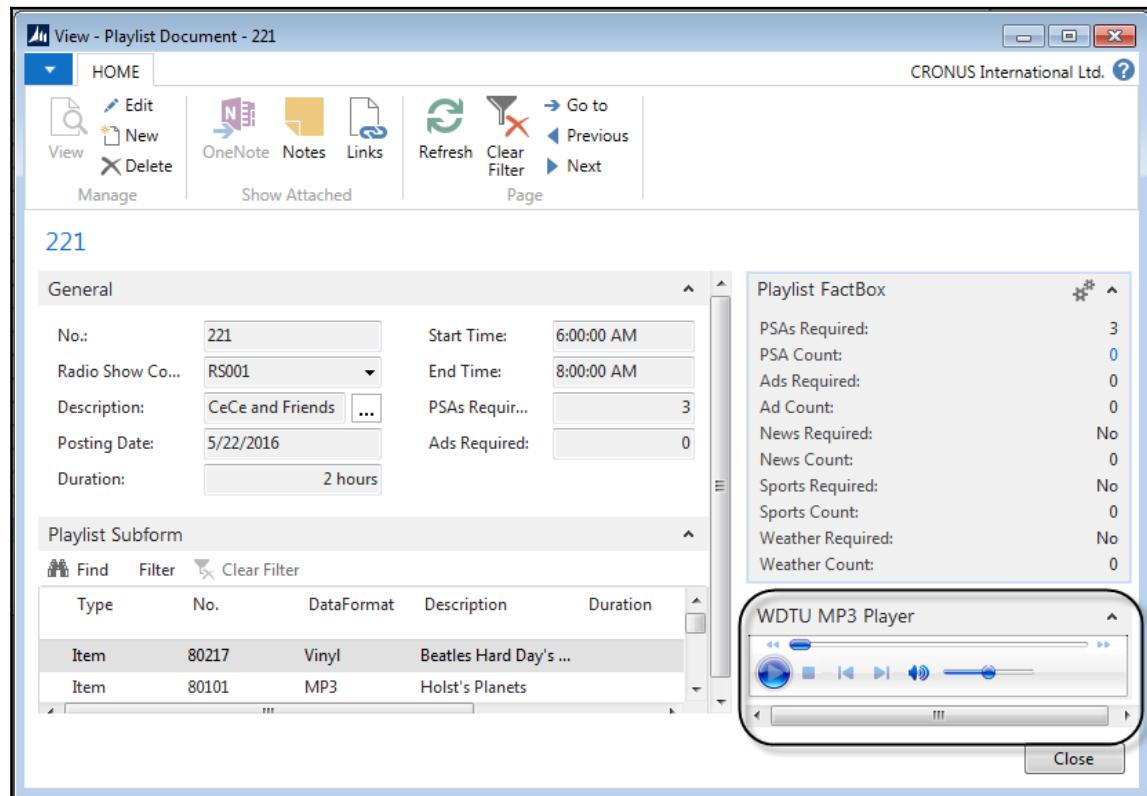
As with any API, there is a defined approach that we must use to create a Client Add-in to interface with the NAV Windows RTC. So long as the code within the add-in is well-behaved .NET code, we have a great deal of flexibility in the structure of the code within the add-in. The control can be one we create, a standard **WinForms** control or one that we've acquired from a third party.

Once we have the .NET control we'll use for our application, we will need to build the add-in structure that envelopes the control. The most logical toolsets to build add-ins are current versions of Visual Studio or one of the free downloadable tools, such as Visual Studio Community for C#. When building an add-in, we must make sure that we are using a compatible version of .NET Framework. The Developer and IT Pro Help in NAV 2017 contains many Help sections covering a wide variety of topics relating to Client Add-ins. Here's a partial list of such sections:

- *Binding a Windows Client Control Add-in to the Database*
- *Client Extensibility API Overview*
- *Developing Windows Client Control Add-ins*
- *Exposing Events and Calling Respective C/AL Triggers from a Windows Client Control Add-in*
- *Exposing Methods and Properties in a Windows Client Control Add-in*
- *Extending the Windows Client Using Control Add-ins*
- *How to Create a Window Client Control Add-in*
- *How to Determine the Public Key Token of the Windows Client Control Add-in*
- *How to Install a Windows Client Control Add-in Assembly*
- *How to Register a Windows Client Control Add-in*
- *Installing and Configuring Windows Client Control Add-ins on Pages*
- *Walkthrough: Creating and Using a Window Client Control Add-in*
- *Windows Client Control Add-in Overview*

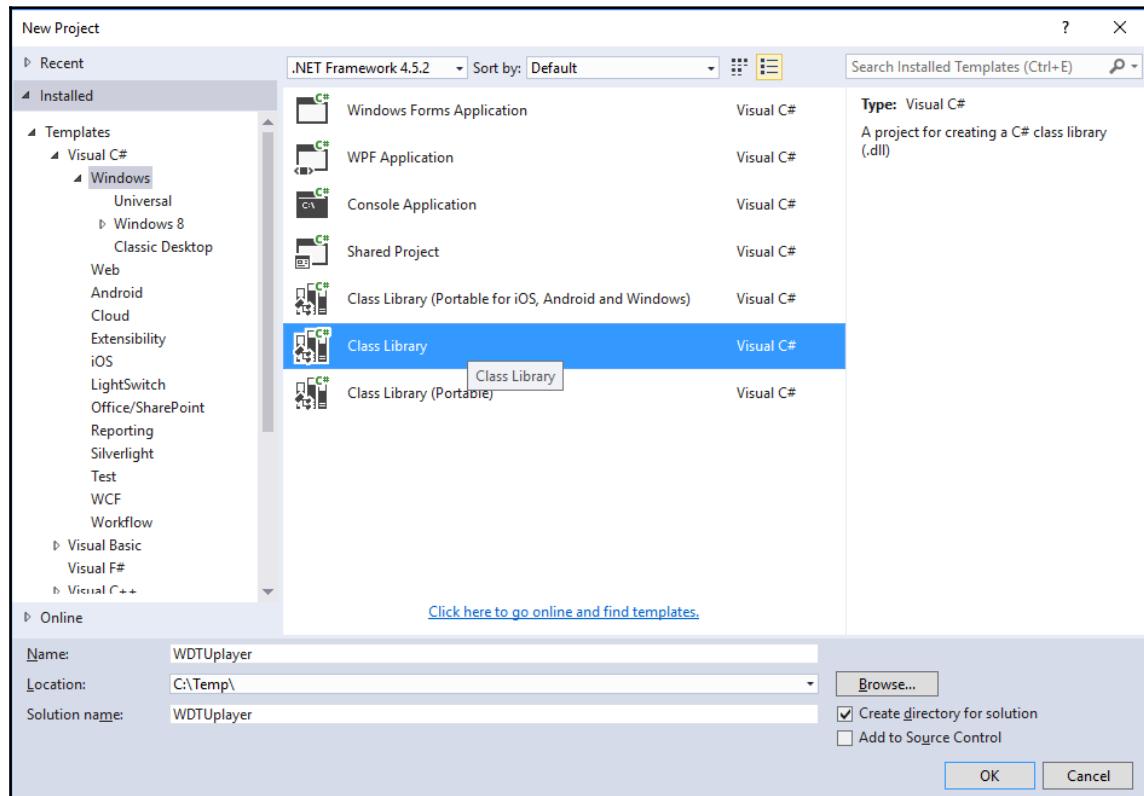
## WDTU Client Add-in

Let's create a Client Add-in for WDTU. We want to add an MP3 player to the Playlist page to allow the user to preview songs on the Playlist. The following screenshot shows what it will look like when we're done (our MP3 player FactBox is circled):

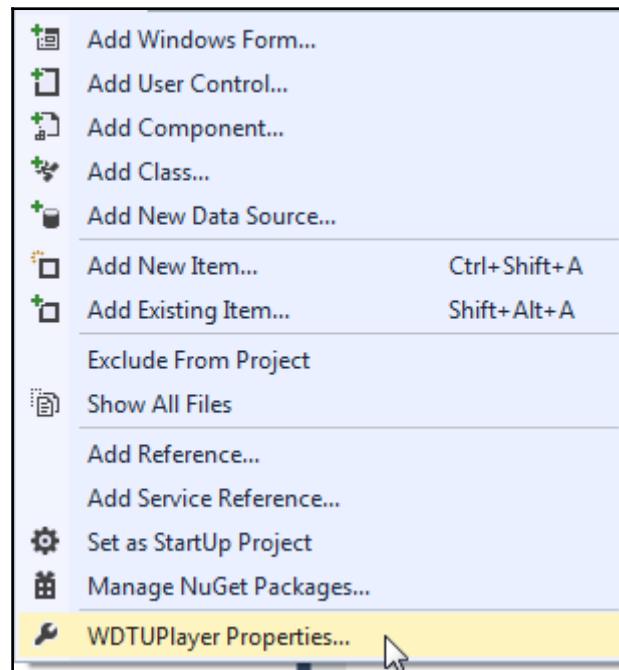


To accomplish this, we will create a .NET assembly (.dll) utilizing the Windows Media Player.

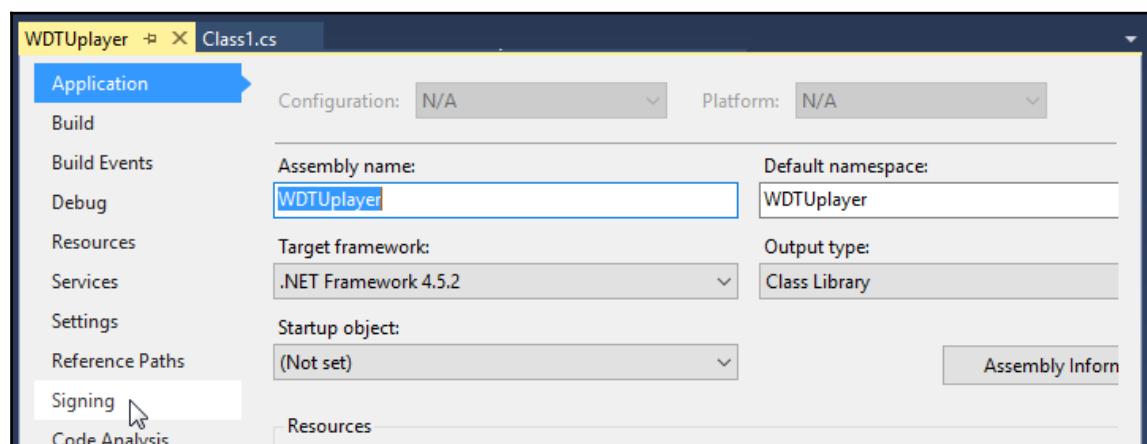
To start, we will open Visual Studio 2015 and create a New Project. For a template, select **Visual C# - Windows Class Library** as shown in the following screenshot. Make sure the .NET Framework selected is 4.5.2 or higher. Name the solution WDTUpLayer and place in a directory we can access later (we'll use C:\Temp). Check the **Create directory for solution** option:



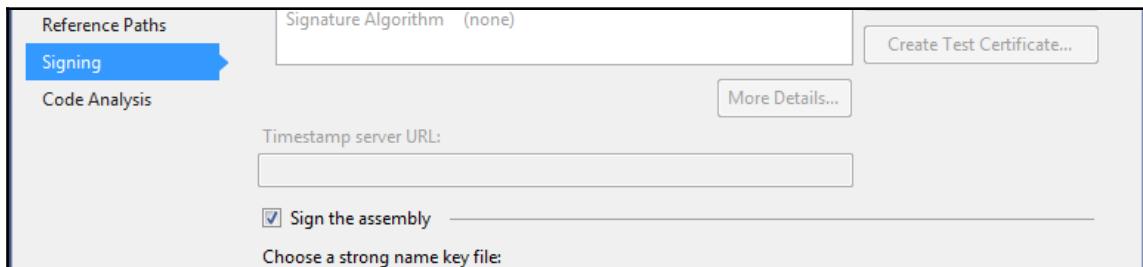
In order to access the .dll in NAV, we will need to create a **Strong Key Name (SNK)**. To do this in Visual Studio, go to the menu for **Project** and select **WDTUPlayerProperties**, as shown in the following screenshot:



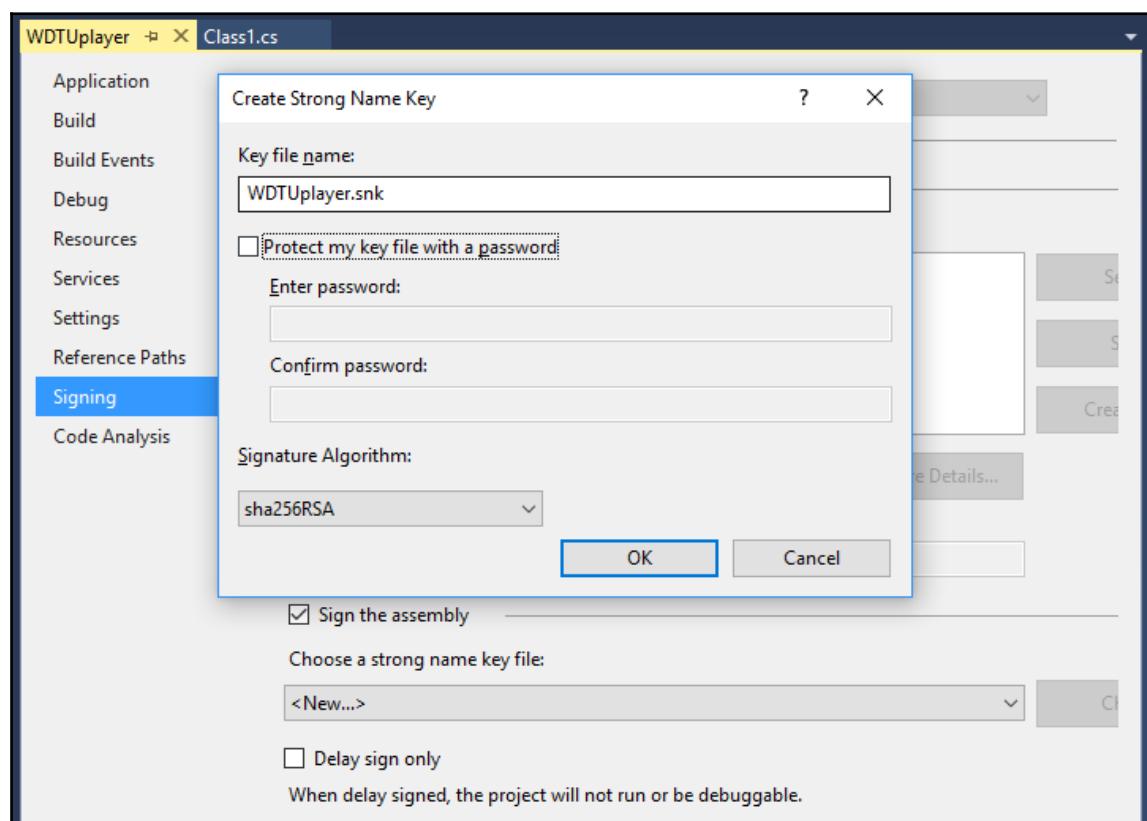
Select the **Signing** option on the left, as shown in the following screenshot:



Then check the **Sign the Assembly** option, as shown in the following screenshot:



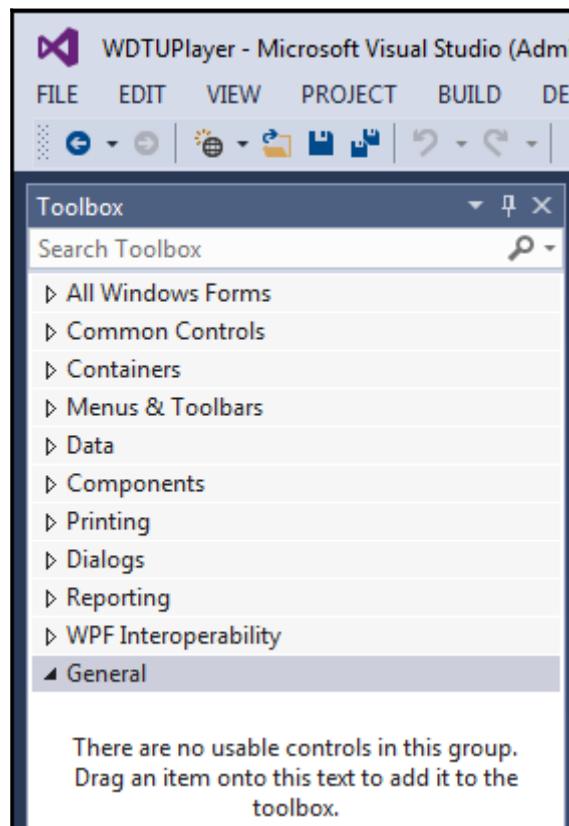
Select <New...> by clicking the drop-down arrow for the box **Choose a strong name key file** and fill in a file name, such as `WDTUpLayer.snk`. Creating the SNK before adding any controls or other objects to the project allows those additional elements to inherit the SNK. See the following screenshot:



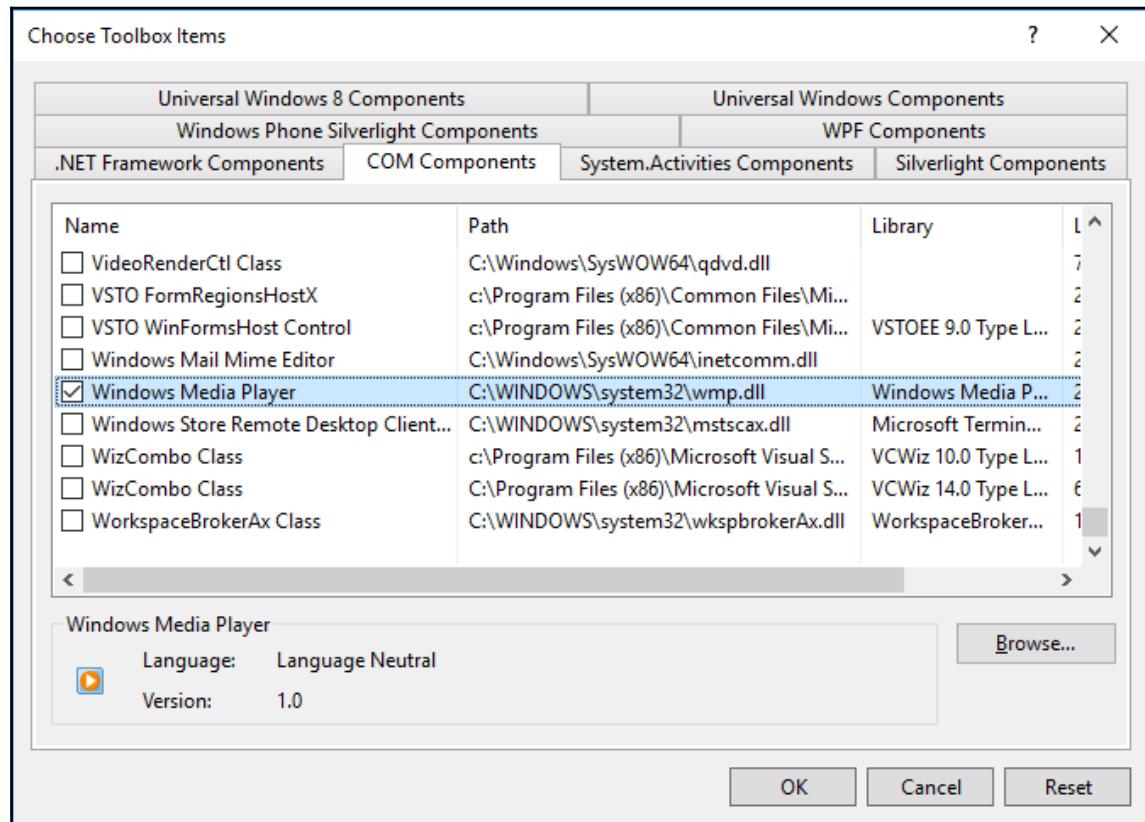
Go to the **Project** menu option and select **Add New Item** (*Ctrl + Shift + A*), then select **Windows Form**. The new **Windows Form** object will be added to our Form design layout screen. Right-click on the form and select **Properties** (or navigate through **View | Properties**). Review the form properties and set the ones defined in the following table:

Property	Value
FormBorderStyle	None
Text	WDTU MP3 Player
Locked	True
AutoSizeMode	None
AutoSize	True
Padding	0,0,0,0
Size	276,47
MaximumSize	0,0
MinimumSize	0,0
MaximizeBox	False
MinimizeBox	False

Next, we will need to make the Windows Media Player (a COM object) available in the Toolbox. Windows Media Player must be installed on our machine (it typically already is). Click on **View | Toolbox** (or *Ctrl + W + X*) and scroll to the **General** group, as shown here:

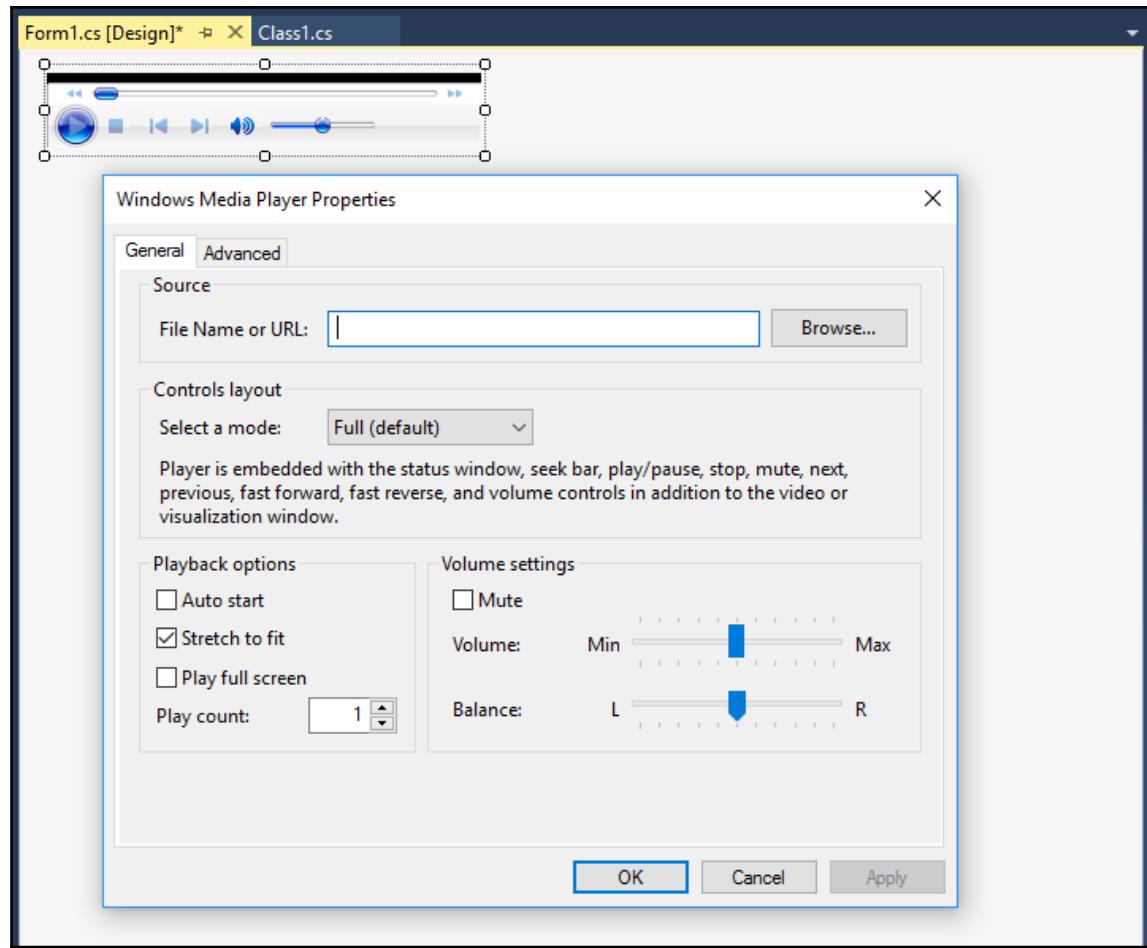


Right-click on **General** and select the **Choose Items** option. In the **Choose Toolbox Items** form, select the **COM Components** tab and scroll down to **Windows Media Player**. Check the box and click on **OK** as shown in the following screenshot:



The **Windows Media Player** control will now be available in the **Toolbox | General** tab.

Select the **Windows Media Player** control and drop it into the form that we added earlier. Right-click on the **Windows Media Player** control and then click on **Properties**. Uncheck **Auto Start** and check **Stretch to Fit**, as shown in the following screenshot:

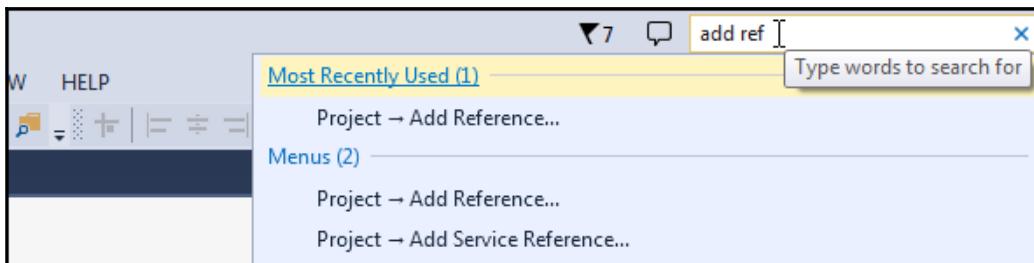


Close the **Windows Media Player Properties** window, click on **View | Properties Window** (or *Ctrl + W + P*), and set properties, as shown in the following table:

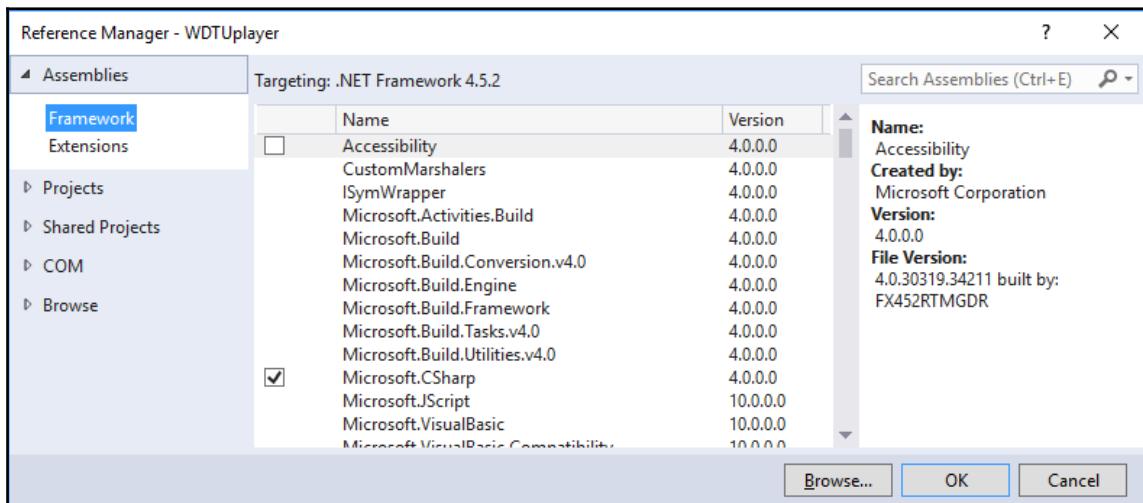
PROPERTY	VALUE
Name	WDTU_MP3_Player
Locked	True
Anchor	Top, Bottom, Left, Right
Location	0,0
Margin	0,0,0,0

Size	275, 45
fullScreen	False
stretchToFit	False
windowlessVideo	False

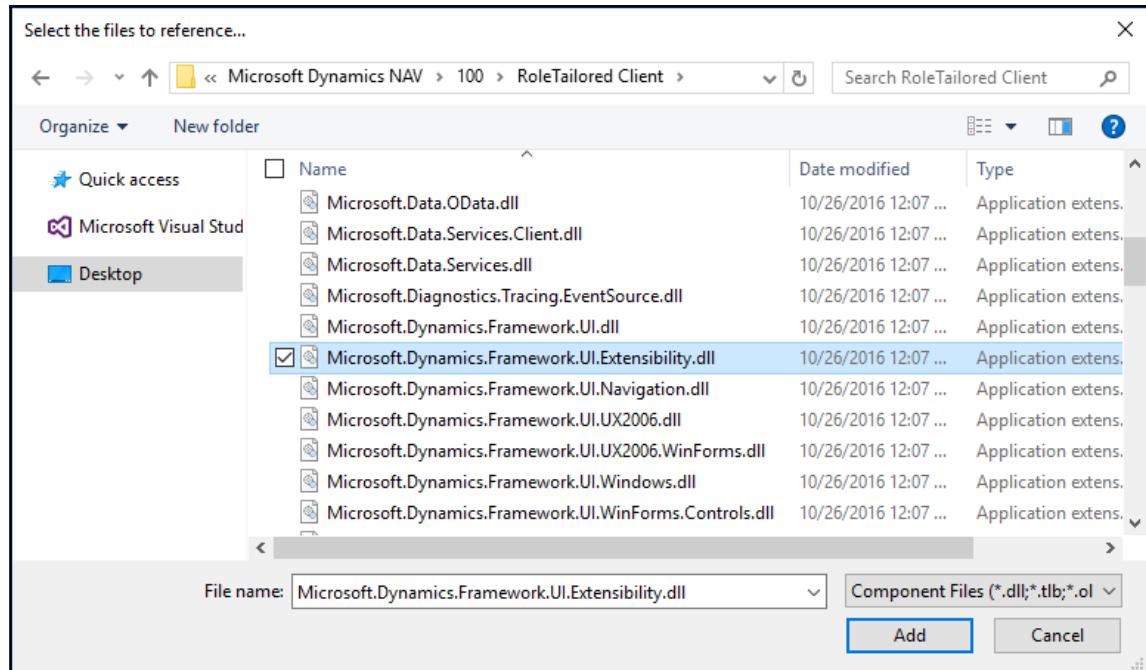
In the **Quick Launch box**, at the top right of the screen, type in Add Reference to search for that function. In the following image, only "add ref" was typed before the **Project - Add Reference** link was displayed):



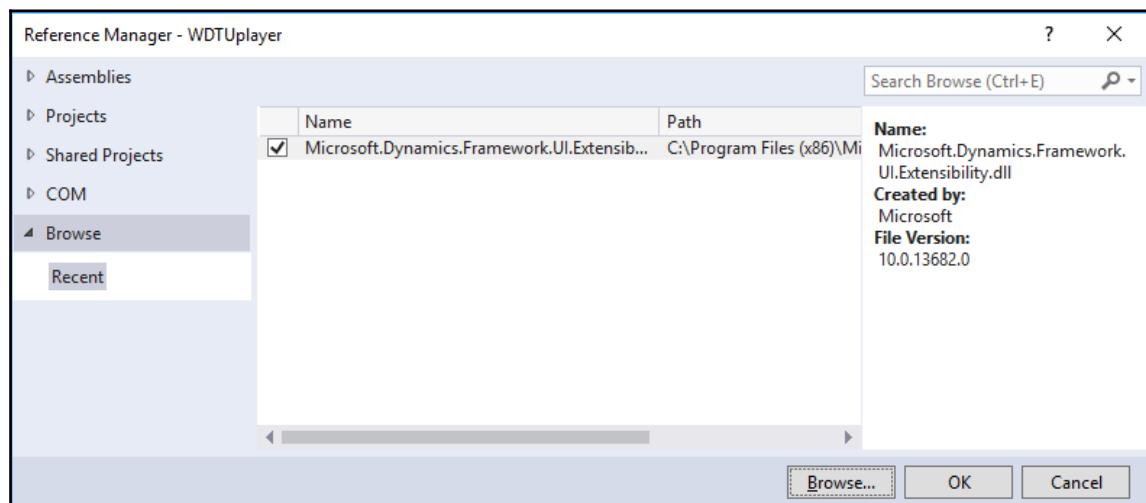
Click on **Project - Add Reference** to display the following screen:



Click on the **Browse** button. Find **Microsoft.Dynamics.Framework.UI.Exensibility**; it is usually found in C:\Program Files (x86)\Microsoft Dynamics NAV100\RoleTailored Client, as shown in the following screenshot:



Then **Add** it as a Reference, and click on **OK** as shown in the following screenshot:



To get to the code behind the `Form1.cs`, in the Solution Designer window, right-click on the form to select **View Code** or double-click on the form image in the main window. Once we are viewing the code, we should make it look like the following:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;using System.Windows.Forms;
namespace WDTUpLayer
{
    public partial class Form1 : Form
    {
        //string to put value from NAV into
        string MoviePath;
        public Form1()
        {
            //Make sure not a null string -
            //will error if not initialized
            MoviePath = "";
            InitializeComponent();
        }

        //Function called from Class to set URL path string
        public void SetMoviePath(string pMoviePath)
        {
            MoviePath = pMoviePath;
        }
        //Default function for NAVMediaPlayer control on form
        //where the URL can be dynamically set from NAV
        private void WDTU_MP3_Player_Enter(object sender, EventArgs e)
        {
            WDTU_MP3_Player.settings.autoStart = false;
            WDTU_MP3_Player.URL = MoviePath;
        }
    }
}
```

In the `Class1.cs` object, the code needs to be created as follows:

```
using System;
using System.Collections.Generic;using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Dynamics.Framework.UI.Extensibility;
```

```
using Microsoft.Dynamics.Framework.UI.Extensibility.WinForms;
using System.Windows.Forms;
using System.Drawing;
using System.ComponentModel;
namespace WDTUplayer
{
    [ControlAddInExport("Cronus.DynamicsNAV.WDTU_MP3")]
    [Description("WDTU MP3 Player")]
    public class WDTU : StringControlAddInBase
    {
        //Create form instance to pass value from NAV
        WDTUplayer.Form1 WMPForm = new WDTUplayer.Form1();
        //Initialize the form
        protected override Control CreateControl()
        {
            WMPForm.TopLevel = false;
            WMPForm.Visible = true;
            return WMPForm;
        }
        // This is the function that receives from NAV (set)
        // and sends back to NAV (get) the string value from
        // the SourceExp property on the NAV page field with
        // the ControlAddIn assigned to it
        public override string Value
        {
            get
            {
                return base.Value;
            }
            set
            {
                base.Value = value;
                WMPForm.SetMoviePath(base.Value);
                //Function in form
            }
        }
    }
}
```

Note the statement defining `public override string Value`. This is the override of the `Value` property in the `Class1.cs` object that retrieves the value passed from NAV (set) and passes the value from the .NET assembly back to NAV (get). Because we defined the `WMPForm` variable as the `Form` object, we can pass the value retrieved from NAV (the field value linked in the CardPart page we will create to hold our Client Add-in).

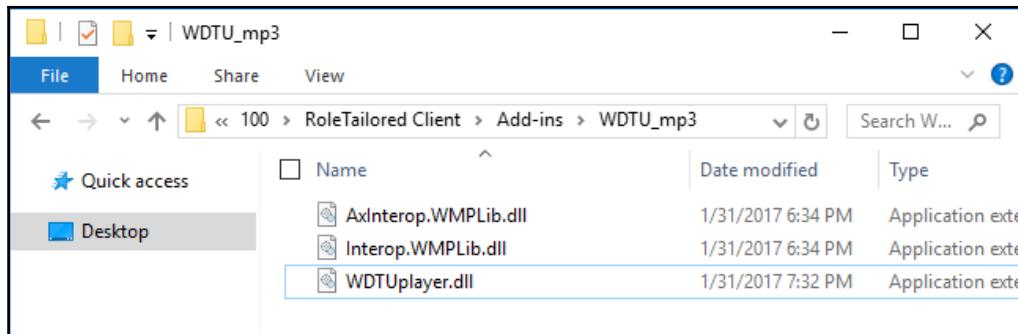
Save all the objects and go to the **Build** menu and select **Build WDTUPlayer**. Locate and copy the following files from our Visual Studio Project folder:

- WDTUplayer.dll
- Interop.WMPLib.dll
- AxInterop.WMPLib.dll

Client Add-ins are each placed in their own directory within the directory (this is the default location):

```
C:\Program Files (x86)\Microsoft Dynamics NAV\100\RoleTailored Client\Add-ins
```

We'll create directory `WDTU_mp3` in that location and place our three files there. Depending on the development computer's setup, additional files may also have to be copied to the `WDTU_mp3` directory, as shown in the following screenshot:



From the **StartMenu** | **All Programs** | **Visual Studio 2015** | **Visual Studio Tools**, run **Developer Command Prompt for VS2015**. When the Visual Studio Command Prompt screen is displayed, enter the following command as shown in the following screenshot:

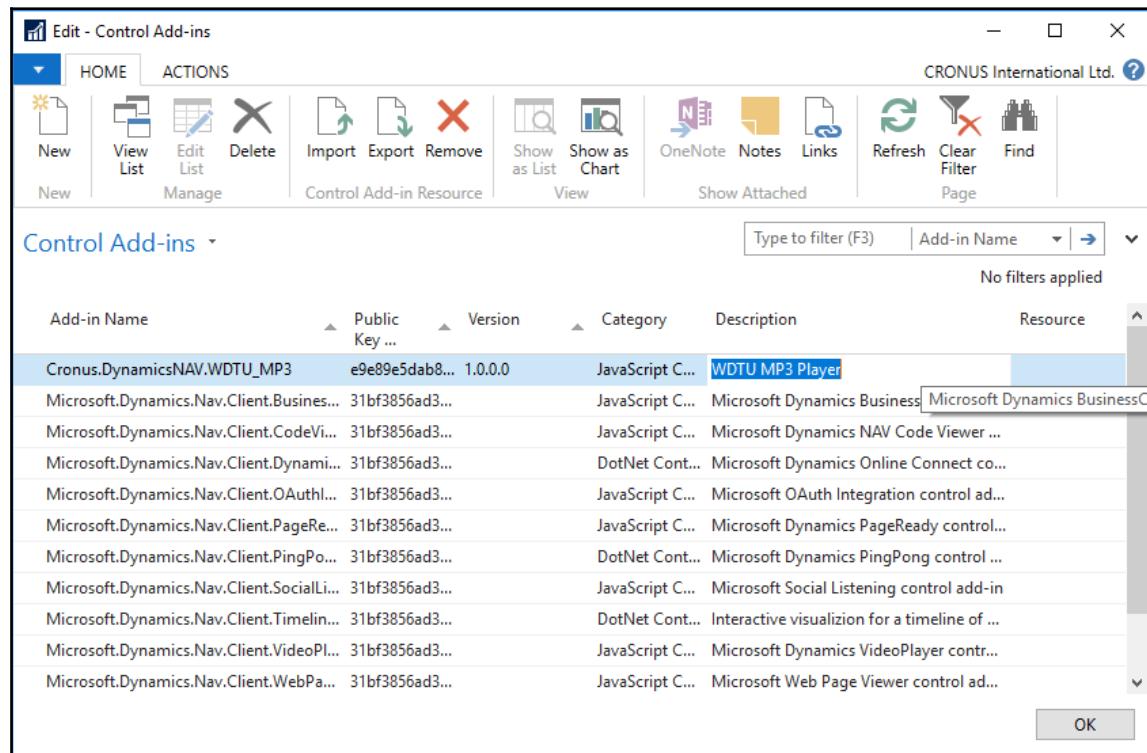
```
sn -T "C:\Temp\WDTUplayer\WDTUplayer\bin\Debug\WDTUplayer.dll"
```

```
Developer Command Prompt for VS2015
C:\Program Files (x86)\Microsoft Visual Studio 14.0>sn -T "C:\Temp\WDTUplayer\WDTUplayer\bin\Debug\WDTUplayer.dll"
Microsoft (R) .NET Framework Strong Name Utility Version 4.0.30319.0
Copyright (c) Microsoft Corporation. All rights reserved.

Public key token is e9e89e5dab8b51dc
C:\Program Files (x86)\Microsoft Visual Studio 14.0>
```

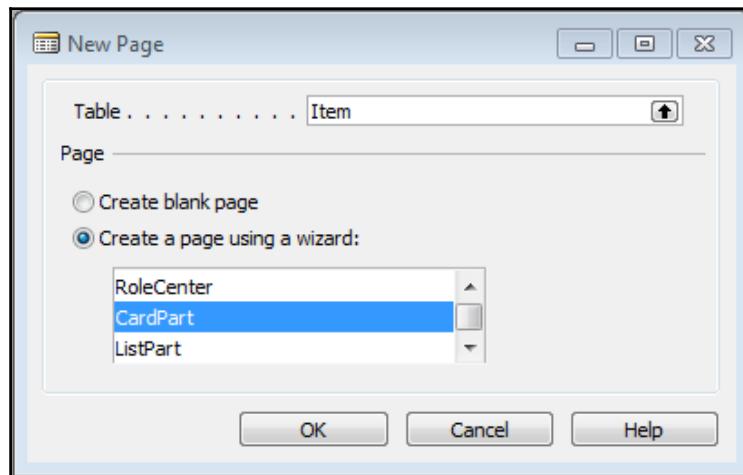
The **Public key token** in this screenshot is **e9e89e5dab8b51dc** (yours likely will be different).

Exit the Visual Studio Command screen and open the NAV 2017 Windows Client and type **Control Add-ins** in the **Search box** to find and open the **Control Add-ins** page. Click on **New** and enter the control **Add-in Name**, **Public Key Token**, **Version** (we will decide what version), and a **Description** as shown in the following screenshot:

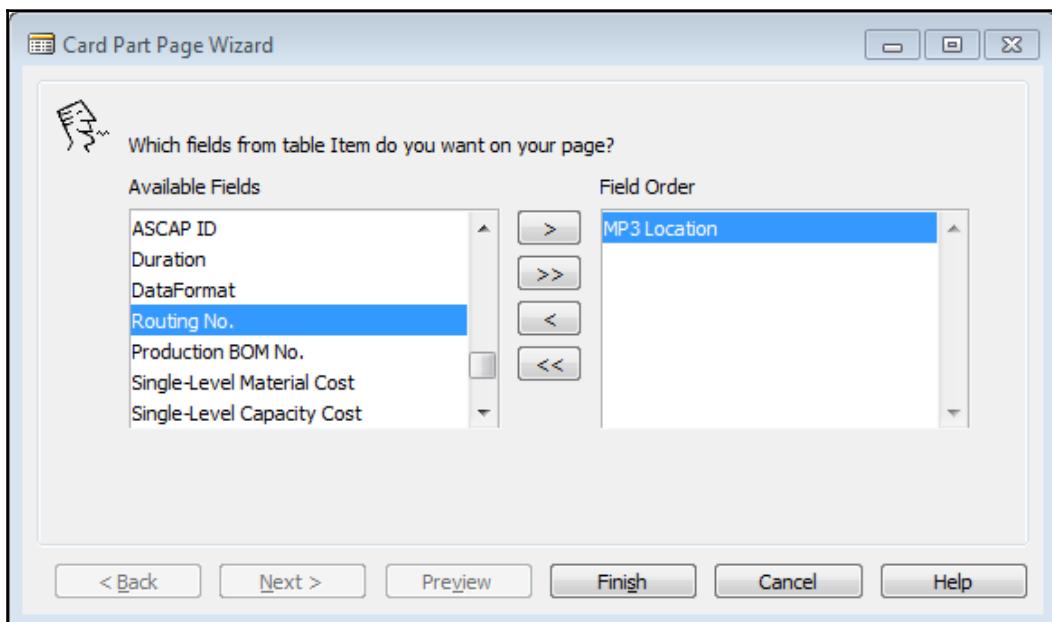


Now that our client's add-in is registered in NAV, we will create a **CardPart** to display a control containing our new Media Player Add-in control, which links the .NET .dll element to a field in NAV.

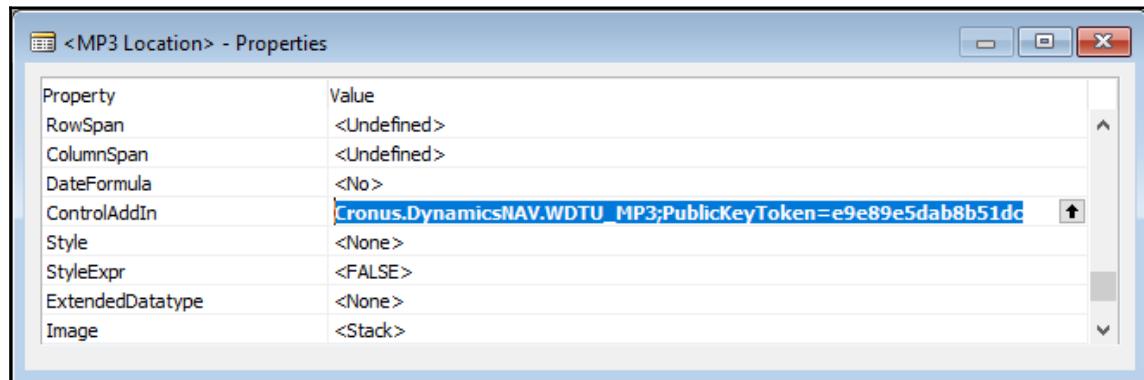
Open the Development Environment and click on **Page | New** in the Object Designer, enter **Item** for the table, **Create a page using a wizard:** and **CardPart** for the page type as shown in the following screenshot:



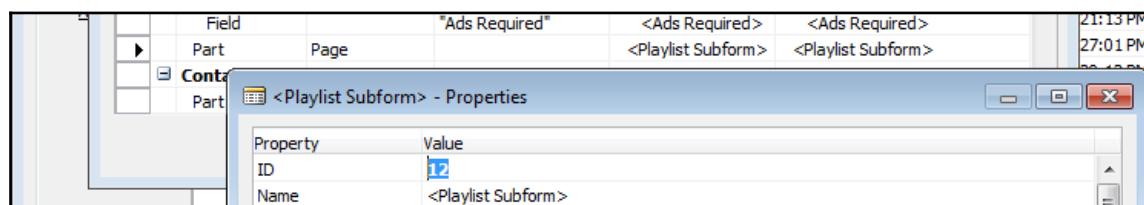
On the next tab, only select a single field, **MP3 Location**. This is the value we wish to pass to the .NET assembly. See the following screenshot:



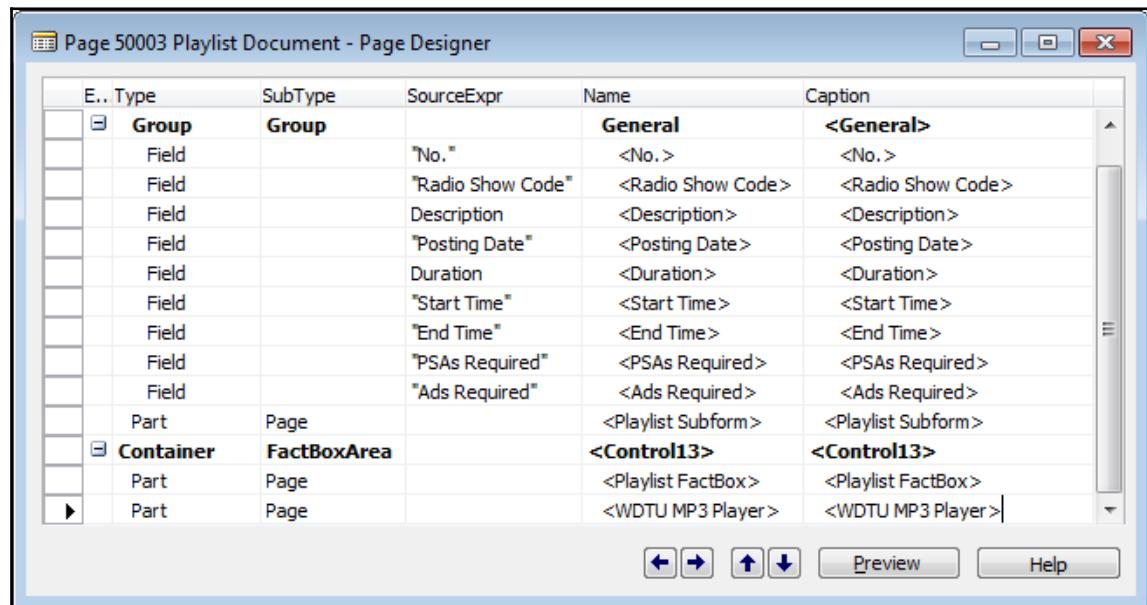
Click on **Finish** and save the Page as 50011 – WDTU MP3 Player. Click on the field **MP3 Location** and view the properties which will display the following screenshot. Scroll down to **ControlAddIn** and perform a lookup to select the **Cronus.DynamicsNAV.WDTU\_MP3**. The public key token will also be populated. Set the **ShowCaption** property to **No**. **Save** and close the page:



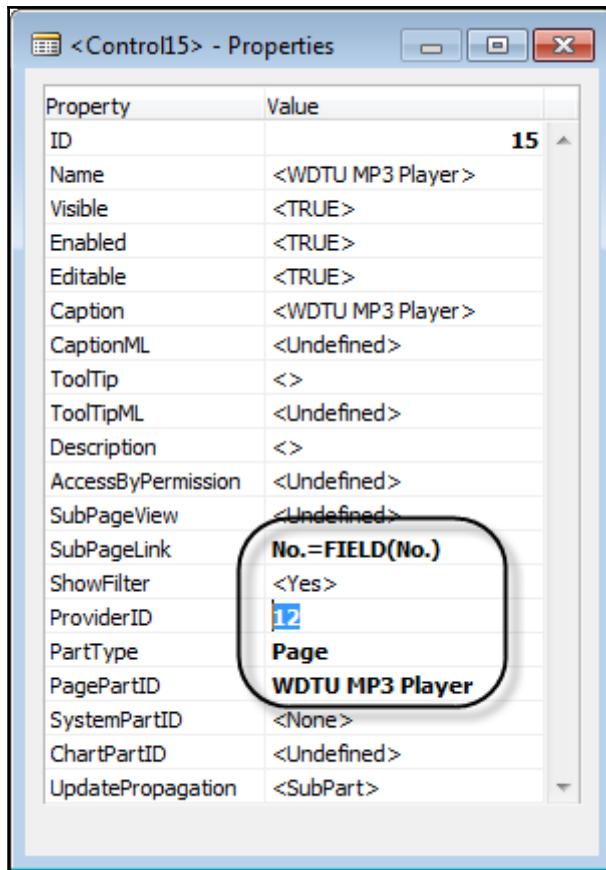
Design **Page 50003 - Playlist**. Open the properties of the subform control for the **Playlist Subform** and copy the control ID, as shown in the following screenshot:



We need to assign this to the WDTU MP3 Player **CardPart** to link the **CardPart** to the **SubForm** lines as shown in the following screenshot. In the **Page Designer** for Playlist page, add the **CardPart** as a **FactBox** under the **PlaylistFactbox**:



Display the properties for this new Page Part control and fill in the properties as follows:



Now, we will test our work. Of course, in order to test, we will need to have test data set up to match what we've designed in our software. We must have Items that represent MP3 files and which contain the location of those MP3 files. Plus, the MP3 files must exist in the defined location. We should set up at least a couple of test items and associated MP3 files so we can enjoy the results of our work.

Once we have a minimal amount of test data set up, **Run the Playlist page (50003)**. If all is well, the **WDTU MP3 player** should play the MP3 from the defined file location for each item in the playlist subpage. If it doesn't work, then this will be an opportunity to experiment some more with the NAV 2017 Debugger.

## Client Add-in comments

In order to take advantage of the Client Add-in capabilities, we will need to develop minimal Visual Studio development skills and some .NET 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 Dynamics NAV system. This feature allows ISVs to create and sell libraries of new controls and new control-based micro-applications. It allows vertically focused partners to create versions of NAV that are much more tailored to their specific industries. This feature allows the integration of third-party products, software, and hardware, at an entirely new level.

The Client Add-in feature is very powerful. If you learn how to use it, you will have another flexible tool in your kit, and your users will benefit.

## Creating an Extension

In NAV 2017 we can publish our changes as **Extensions**. Using extensions allows us to create customizations without making changes to the source code shipped by Microsoft. This makes future upgrades less challenging for our customers by eliminating the code merge portion of an upgrade.

If you want to create an Extension, you are not allowed to make code changes to most existing objects shipped by Microsoft. There are only two supported object modification categories, those to tables and those to pages.

## Table Changes

When creating an Extension, we are allowed to add fields to existing table objects. Adding functions or changing existing code is not allowed.

## Page Changes

We are allowed to add fast tabs, fields and actions to existing pages. The actions can be connected to code unit objects using an action's RunObject property.



The MSDN entry, Extension Packages Capability Support Matrix, describes what changes can be made to Table and Page objects to support an Extension ([https://msdn.microsoft.com/en-us/library/mt574414\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/mt574414(v=nav.90).aspx)).

In the next version of NAV, Table Changes and Page Changes will be replaced by new object types called Table Extensions and Page Extensions. You can read about these planned NAV enhancements on MSDN at

<https://msdn.microsoft.com/en-us/dynamics-nav/newdev-table-ext-object>.

## Events

An **Event** is an automatic notice of a specific occurrence or change in the application. There are five types of events:

- Business Events
- Integration Events
- Global Events
- Database Trigger Events
- Page Trigger Events

If we want to make changes to the existing flow of business logic for our Extension, we must use the events that are made available by Microsoft. You can find more information on using events at [https://msdn.microsoft.com/en-us/library/mt299398\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/mt299398(v=nav.90).aspx) and by watching the video at <https://www.youtube.com/watch?v=zQHVv7PV8u8&t=2843s>.

## Creating a WDTU extension

The WDTU application that we have created in the following section falls within the defined restrictions for an Extension since we only added new objects. We have not changed any existing Microsoft code, only added fields to a table.

To create the Extension, we need to run Windows PowerShell and use PowerShell commands that Microsoft ships with the product. See the NAV 2017 Developer and IT pro Help: *How to: Create an Extension Package* at <https://msdn.microsoft.com/en-us/dynamics-nav/how-to--create-an-extension-package>.

## Step 1 - Load PowerShell

We will use the PowerShell **Integrated Scripting Environment (ISE)** that is installed by default on every Windows machine. We will start by running the following lines of script:

```
Set-ExecutionPolicy unrestricted

import-module "C:\Program Files (x86)\Microsoft Dynamics NAV100RoleTailored
ClientNavModelTools.ps1"
import-module "C:\Program Files\Microsoft Dynamics
NAV100ServiceNavAdminTool.ps1"
```



The first time you run PowerShell you may have to enable scripts. This blog describes how to enable scripts (<https://markbrummel.blog/2015/02/19/tip-44-first-time-powershell-enable-scripts/>).

## Step 2 - Create Delta files

Because Extensions use the **Delta** file format, we must run the command that creates Delta files. Delta files are outside of the scope of this book, but information can be found in MSDN at: [https://msdn.microsoft.com/en-us/library/dn762344\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/dn762344(v=nav.90).aspx)

To create Delta files we must export all the objects from our Dynamics NAV database and compare them to a similar export from a clean Microsoft database.

The following script is used to create the Delta files:

```
Compare-NAVApplicationObject      -OriginalPath "Q:\WDTUOrignal.txt" ` 
                                     -ModifiedPath "Q:\WDTUExample.txt" ` 
                                     -DeltaPath "Q:\WDTUDelta" -Force
```

In the preceding script, Q:\WDTU represents a location in your filesystem where the extension will be created.

The result of running this script is a set of files in the Q:\WDTUDelta folder with the .DELTA file extension.

## Step 3 - Manifest XML file

Our Extension must have a name, version, and publisher in order to be created. This information must be stored in an XML file called a **Manifest**.

In the Manifest XML file we can also provide more optional details as described in this MSDN reference ([https://msdn.microsoft.com/en-us/library/mt600264\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/mt600264(v=nav.90).aspx)).

Now, let's focus on the minimum required fields to define our Manifest:

```
New-NAVAppManifest -Name "WDTU" -Publisher "CDM" -Version "1.0.0.0" | New-NAVAppManifestFile -Path "Q:WDTUManifest.xml"
```

The result should look something like this:

```
<Package xmlns="http://schemas.microsoft.com/navx/2015/manifest">
  <App Id="26ec7527-88b9-4b66-a77a-5a9eebbbf627" Name="WDTU"
    Publisher="CDM" Brief="" Description="" Version="1.0.0.0"
    CompatibilityId="1.0.0.0" PrivacyStatement="" EULA="" Help="" Url=""
    Logo="" />
  <Capabilities />
  <Prerequisites>
    <Objects />
  </Prerequisites>
  <Dependencies />
  <ScreenShots />
</Package>
```

## Remembering the App ID

The App ID GUID is generated by the PowerShell command. This ID should be used when creating future versions of our Extension. To do this we must add the option `-ID` (that is blank, dash, blank, the letters ID) to the `New-NAVAppManifest` command.



This MSDN article describes how to upgrade your Extension to a new version ([https://msdn.microsoft.com/en-us/library/mt703361\(v=nav.90\).aspx](https://msdn.microsoft.com/en-us/library/mt703361(v=nav.90).aspx)).

## Step 4 - Create the NAVx package

The last step in preparing our WDTU application as an extension is to combine the Delta files and the Manifest into a single file that we can send to our customer.

This is done using the following script:

```
Get-NAVAppManifest -Path "Q:WDTUManifest.xml" | New-NAVAppPackage -Path  
"Q:WDTUWDTU.navx" -SourcePath "Q:WDTUDelta"
```

Congratulations. You have just created your first NAV Extension ready to be installed for your customer.



In the next version of NAV, the PowerShell script will be embedded in the Development Environment  
(<https://blogs.msdn.microsoft.com/nav/2017/02/16/nav-development-tools-preview-february-update/>).

## Installing the Extension

To install the Extension at a Customer, we first need to run one line of PowerShell script. This script requires NAVAdminTools.ps1 to be loaded as described in Step 1 of creating the Extension.



To publish an Extension, PowerShell should be initiated in Administrator mode.

## Publishing an Extension

Before our customer can use our application this line of PowerShell code can be executed:

```
Publish-NAVApp -ServerInstance DynamicsNAV100 -Path "Q:WDTUWDTU.navx" -  
SkipVerification
```

We use the `-SkipVerification` parameter because our extension is not signed with a certificate. The `ServerInstance` we use in our example (`DynamicsNAV100`) is the default instance and may be different on your system.

## Verification

As most system administrators we want to verify if our action was successful. We can do that by executing the following line of PowerShell:

```
Get-NAVAppInfo -ServerInstance DynamicsNAV100
```

This returns a list of available Extensions for our customer. It should now include the WDTU application as shown in the following screenshot:

Id	Name	Version	Publisher
d09fa965-9a2a-424d-b704-69f3b54ed0ce	PayPal Payments Standard	v. 1.0.0.2	Microsoft
bc45ae22-3b5b-44b5-beb4-2a42bf79cc34	Quickbooks Payroll	v. 1.0.0.0	Microsoft
c526b3e9-b8ca-4683-81ba-fcd5f6b1472a	Sales and Inventory	v. 1.0.0.2	Microsoft
1b80b577-772f-4e0f-bc13-50214fb3da6e	QuickBooks Data Migration	v. 1.0.0.2	Microsoft
e2743298-9ccb-49cd-9d8e-4b2d1ab91d36	Envestnet Yodlee	v. 1.0.0.2	Microsoft
30828ce4-53e3-407f-ba80-13ce8d79d110	Ceridian Payroll	v. 1.0.0.0	Microsoft
26ec7527-88b9-4b66-a77a-5a9eebbbf627	WDTU	v. 1.0.0.0	CDM

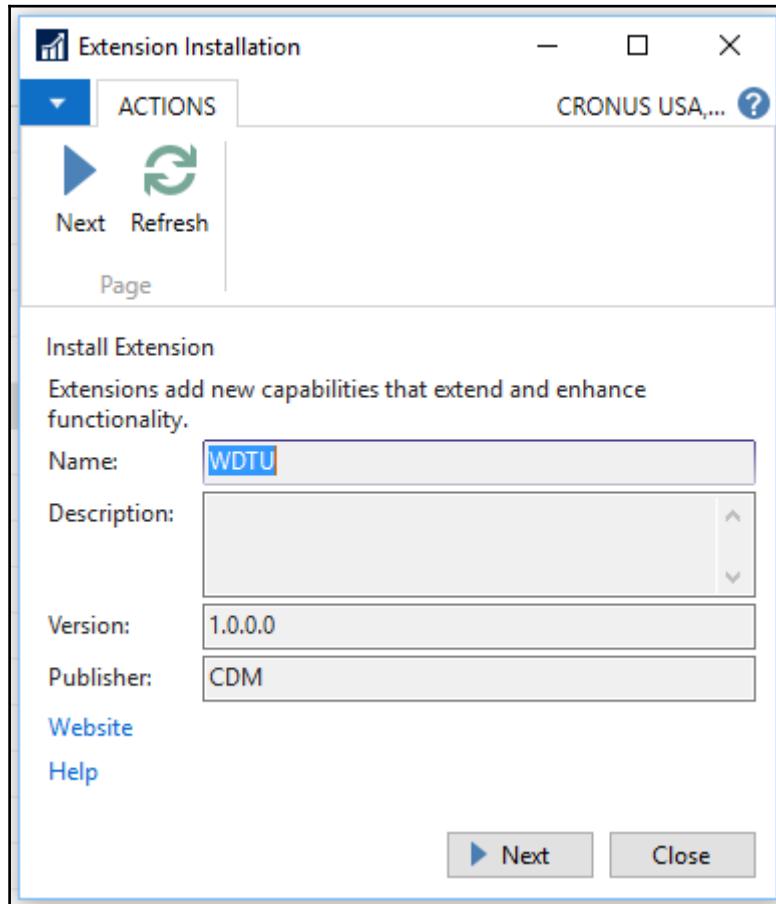
## Extension installation and setup

After the Extension is published by our system administrator, the user can install it from the Windows Client using the Extension Management page as shown in the following screenshot:

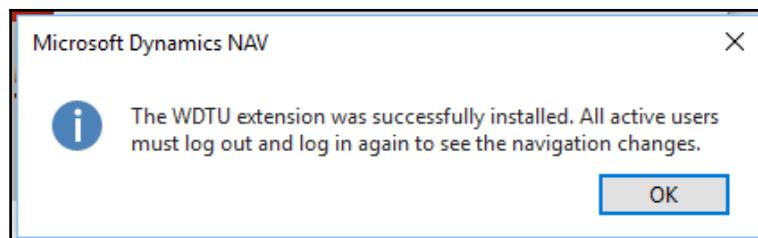
The screenshot shows the 'View - Extension Management' page. The top navigation bar includes 'Extension Management' and the company name 'CRONUS USA, Inc.'. The ribbon has tabs for 'View', 'HOME', and 'ACTIONS'. Under 'ACTIONS', the 'Install' button is highlighted with a red box. The main grid displays a list of extensions with columns for 'AdditionalInfo', 'Name', and 'Version'. One row for 'WDTU' is highlighted with a red box and shows 'Not Installed' in the AdditionalInfo column.

AdditionalInfo	Name	Version
Installed	Ceridian Payroll	v. 1.0.0.0
Installed	Envestnet Yodlee Bank Feeds	v. 1.0.0.2
Installed	PayPal Payments Standard	v. 1.0.0.2
Installed	QuickBooks Data Migration	v. 1.0.0.2
Installed	Quickbooks Payroll File Import	v. 1.0.0.0
Installed	Sales and Inventory Forecast	v. 1.0.0.2
Not Installed	WDTU	v. 1.0.0.0

When we try to install the WDTU Extension from this page we get a helpful wizard that guides us through the process (shown in the following screenshot):



And after you click on **Next** and click on **Finish**, you will see the screen shown in the following screenshot:



Congratulations! Our customer can now use the WDTU Application as an extension.



Important note: When you ship your Dynamics NAV modification as an Extension you cannot make changes to the modified code in the production database where it is used. You have to go back to the original development code and distribute your changes again as a new Extension version.

## Customizing Help

NAV 2017 comes with a Help Server component. When installing NAV 2017, we need to make sure we are also installing the NAV Help Server on a Web server system to which all clients have access. Review the following *Developer and IT Pro Help* entries along with other entries that relate to modifying and maintaining **Help** data:

- *Configuring Microsoft Dynamics NAV Help Server*
- *Microsoft Dynamics NAV Help Server*
- *Upgrading Your Existing Help Content*

All the NAV Help files are now HTML files, accessible for whatever type of changes are appropriate for the installed system. This includes adding information to existing helps, having multiple language Helps, adding new Helps associated with customized or tailored applications, whatever changes that can make the software work better for the customer.

The process for developing customized Help is similar in many ways to developing customizations to the application software. 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 operational challenges for the users as well as 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. Similar to 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.

With NAV 2017, Microsoft has published their Help source on GitHub (<https://github.com/Microsoft/nav-content>), allowing us to access it directly and contribute to it.

# NAV development projects - general guidance

Now that we 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.

When we start a new project, the goals and constraints for the project must be defined. The degree to which we meet these will determine our success. Some examples are follows:

- 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?

## Knowledge is key

Designing for NAV requires more forethought and knowledge of the operating details of the application than was needed with traditional models of ERP systems. As we have seen, NAV has unique data structure tools (SIFT and FlowFields), quite a number of NAV-specific functions that make it easier to program business applications, and a software data structure (journal, ledger, and so on) that is inherently an accounting data structure. The learning curve to become an expert in the way NAV works is not easy. NAV has a unique structure and the primary documentation from Microsoft is limited to the embedded Help, which improves with every release of the product. The NAV books published by *Packt* can be of great help, as are the NAV Development Team blogs, the blogs from various NAV experts around the work, and the NAV forums that were mentioned earlier.

## Data-focused design

Any new application design must begin with certain basic analysis and design tasks. This is just as applicable whether our design is for new functionality to be integrated into NAV or is for an enhancement/expansion of existing NAV capabilities.

First, determine what underlying data is required. What will it take to construct the information the users need to see? What level of detail and in what structural format must the data be stored so that it may be quickly and completely retrieved? Once we have defined the inputs that are required, we must identify the sources of 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 derived 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 required data views**

Define how the data should be presented by addressing the following questions:

- How does it need to be "sliced and diced"?
- What levels of detail and summary are required?
- What sequences and segmentations are required?
- What visual formats will be used?
- What media will be used?
- Will the users be local or remote?

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. Because there are a wide variety of tools available to extract and manipulate NAV data, we can start relatively simply and expand as appropriate later. The most important thing is to assure that we have all the critical data elements identified and then captured.

## **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. We will not only need to do a good job of designing the primary tables, but also all those supporting tables containing lookup and setup data. When integrating a customization, we must consider the effects of the new components on the existing processing as well as how the existing processing ties into our new work. 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**

The following are some of the principle design issues that need to be considered:

- 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 (though the SQL Server database supports sorting data without predefined NAV C/AL keys).
- Define what fields will be allowed to be visible, what are the totaling fields, how the totaling will be accomplished (for example, FlowFields or on-the-fly processing), 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.

Subsequent levels of validation may exist for particular subsets of data, which, in turn, are 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. However, if they are filled in, 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 processing, reporting, or inquiries.

## **Data design review and revision**

Perform these three steps (table design, user access, data validation) for the permanent data (Masters and Ledgers) and then for the transactions (Journals). Once all the supporting tables and references have been defined for the permanent data tables, there are not likely to be many 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, we should go back and reexamine 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, we are determining what the permanent content of the Posted data will be. If we identify any new supporting table or table reference requirements at this point, we should go back to the first step to make sure that this requirement didn't need to be in the design definition.

Whatever variations in data are permitted to be Posted must be acceptable in the final, permanent instance of the data. Any information or relationships that are necessary in the final Posted data must be ensured to be present 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, we must define what constitutes a Batch; if the former, it is 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. This allows us to work on the several preceding definitions 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, we increase the likelihood that we will be able to create common routines and standardize how similar tasks are handled across a variety of parent processes. At this point, we may identify opportunities or requirements for improvement in material defined in a previous design step. In that case, we should return to that step relative to the newly identified issue. In turn, we should also review the effect of such changes for each subsequent step's area of focus.

## Double-check everything

Lastly, review 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 (and less painful) to find and fix design issues during the design phase rather than after the system is in testing or, worse yet, in production.

## Design for efficiency

Whenever we are designing a new modification, we will 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 we can do to design an efficient solution are relatively simple:

- Properly configure system and workstation software (often overlooked)
- Make sure networks can handle the expected load (with capacity to spare)
- Have enough server memory, to avoid using virtual memory, virtual memory = disk
- Most of all, do everything reasonable to minimize disk I/O

## 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. When we begin concentrating our design efforts on efficiency, focus first on minimizing 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 access of data (especially FlowFields). If our system will have five or ten users processing a few thousand order lines per day, and is not heavily modified, we probably won't have much trouble. However, if we 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), we will need to be very careful with our design and implementation:

- Large concurrent user count
- High transaction volumes, especially in data being Posted
- Large stored data volumes, especially resulting from customizations or setup option choices
- 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.

If a deadlock occurs, there is a serious design flaw wherein each process has data locked that the other process needs and neither process can proceed. One of our responsibilities as developers or implementers, is to minimize locking problems and eliminate any deadlocks.

Locking interference between processes in an asynchronous processing environment is inevitable. There will always 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.

One might ask, what is an unreasonable delay? For most of the part, a delay becomes unreasonable when the users can tell that it's happening. If the users see stopped processes or 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 required 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 processing. That will reduce the number of lock conflicts that arise. Important recommendations for improving processing speed include the following:

- Restricting the number of active keys, especially on the SQL Server
- 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 access code is SQL Server optimized

Some additional steps that can be taken to minimize locking problems are as follows:

- 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)
- Shift long-running processes to off-hours or even separate databases

In special cases, the following techniques can be used (if done very, very carefully):

- Process data in small quantities (for example, process 10 records or one order, then `COMMIT`, which releases the lock). This approach should be 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).

Refer to the documentation with the system distribution in the NAV forums.

With NAV 2017, Microsoft has introduced a new feature that allows you to view the current locks in the database. You can access this page from the Development Environment by selecting **Tools | Debugger | Database Locks....** The following screen will be displayed:

The screenshot shows the Microsoft Dynamics NAV interface for viewing database locks. The title bar reads "View - Database Locks". The ribbon has a "HOME" tab selected. Below the ribbon are buttons for "Show as List", "Show as Chart", "OneNote", "Notes", "Links", "Refresh", "Clear Filter", and "Find". A search bar at the top right says "Type to filter (F3)" and "Table name". A message "No filters applied" is displayed. The main area is titled "Database Locks" and contains a table with the following data:

Table name	SQL Lock Resource Type	SQL Lock Request...	SQL Lock Request...	User Name	Executing AL Obj...	Executing AL Obj...	Executing AL Method
G_L_Entry	KEY	Update	Granted	ARTHUR\DAVE	Page	39	Post - OnAction
G_L_Register	KEY	Update	Granted	ARTHUR\DAVE	Page	39	Post - OnAction
Gen_Journal Line	KEY	Update	Granted	ARTHUR\DAVE	Page	39	Post - OnAction
Gen_Journal Line	KEY	Update	Granted	ARTHUR\DAVE	Page	39	Post - OnAction
Gen_Journal Line	KEY	Update	Granted	ARTHUR\DAVE	Page	39	Post - OnAction
Gen_Journal Line	KEY	Update	Granted	ARTHUR\DAVE	Page	39	Post - OnAction

A "Close" button is located at the bottom right of the window.

For more details refer to the following Help section: *Monitoring SQL Database Locks*.

## Updating and Upgrading

One must differentiate between updating a system and upgrading a system. In general, most of the NAV development work we will do is modify individual NAV systems to provide tailored functions for end-user firms. Some of those modifications will be created by developers as part of an initial system configuration and implementation before the NAV system is in production use. Other such modifications will be targeted at a system that is in day-to-day production to bring the system up to date with changes in business process or external requirements. We'll refer to these system changes as **Updating**.

**Upgrading** is when we implement a new version of the base C/AL application code distributed by Microsoft and port all the previously existing modifications into that new version. First, we'll discuss Updating, and then we'll discuss Upgrading.

## Design for updating

Any time we 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 for any other production application system. However, some disciplines are specific to NAV and the C/SIDE environment.

Increasing the importance of designing for ease of updating is Microsoft's process of providing NAV updates on a frequent basis so that systems can be kept more up to date with fixes and minor feature enhancements. Keeping up with these Microsoft-provided updates is especially important for multitenant systems running in the cloud, that is, systems serving multiple unrelated customers with the software and databases being resident on Internet-based server systems. Fortunately, in support of the pressure to apply updates more frequently, Microsoft has also provided a set of tools to help us. Many of these tools are based on Windows PowerShell scripts, also referred to as **cmdlets**. For additional information, refer to the Help topics on Deployment and Upgrading.

## Customization project recommendations

Even though there are new tools to help us update our NAV systems, we should still follow good practices in our modification designs and the processes of applying updates. Some of these recommendations may seem obvious. This would be a measure of our personal store of experience and our own common sense. Even so, it is surprising the number of projects go sour because one (or many) of the following are not considered in the process of developing modifications:

- One modification at a time
- Design thoroughly before coding
- Design the testing in parallel with the modification
- Use 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
- Document and deliver in a predefined, organized manner
- Follow up, wrap up, and move on

## **One change at a time**

It is important to make changes to objects 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 stage) before the object is released to be modified for a second purpose.

If the project is so large and complex or deadlines are so tight that this one modification at a time approach is not feasible, we should consider the use of software source management systems, such as the **Git** or **Team Foundation** Systems, that can easily help you to separate multi-modifications.

Similarly, we should only be working on one functional change at a time. As developers, we might be working on changes in two different systems in parallel, but we shouldn't be working on multiple changes in a single system simultaneously. 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 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. We should learn from the past. In mainframe development environments, having multiple developers working on the same system at the same time was common. Coordination issues were addressed as a standard part of the project management process. Applicable techniques are well-documented in professional literature. Similar solutions still apply.

## **Testing**

As we all know, there is no substitute for complete and thorough testing. Fortunately, NAV provides some very useful tools, such as those previously discussed, to help us to be more efficient than we might be in some other environment. In addition to the built-in testing tools, there are also some testing techniques that are NAV specific.

## Database testing approaches

If the new modifications are not tied to previous modifications and specific customer data, then we may be able to use the **Cronus** database as a test platform. This works well when our target is a database that is not heavily modified in the area on which we are currently working. As the Cronus database is small, we will not get lost in large data volumes. Most of the master tables in Cronus are populated, so we don't have to create and populate this information. Setups are done and generally contain reasonably generic information.

If we are operating with an unmodified version of Cronus, we have the advantage that our test is not affected by other preexisting modifications. The disadvantage, of course, is that we are not testing in a wholly realistic situation. Because the data volume in Cronus is so small, we are not likely to detect a potential performance problem.

Even when our modification is targeted at a highly-modified system, where those other modifications will affect what we are doing, it's often useful to test a version of our modification initially in Cronus. This may allow us to determine if our change has internal integrity before we move on to testing in the context of the fully modified copy of the production system.

If the target database for our modifications is an active customer database, then there is no substitute for doing complete and final testing in a copy of the production database using a copy of the customer's license. This way, we will be testing the compatibility of our 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 recent copy of the production database.

Final testing should always be done using the customer's license.



## Testing in production

While it is always a good idea to thoroughly test before adding our changes to the production system, sometimes we can safely do our 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, 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, we may be able to test just a part of the modification outside production, and then implement the table changes to complete the rest of the testing in the context of the production system.

Finally, we can use the Testing functions to control tests so that any changes to the database are rolled back at the conclusion of the testing. This approach allows testing inside a production database with less fear of corrupting live data.



Make sure not to run any test codeunit on its own as Test Isolation is only handled by the Test Runner codeunit.

## Using a testing database

From a testing point of view, the most realistic testing environment is a current copy of an actual production database. There are sometimes apparently 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 copy of the production database!

Remember, when we implement our modifications, they will receive the **test by fire** in the environment of production. We 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:** This is not a good argument relative to disk space. Disk space is so inexpensive that we can almost always afford plenty of disk space for testing. We should also make 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 a major testing process. This is one of the many places where appropriate use of the C/AL Testability tools can be very helpful by allowing various approaches to repetitive testing.

- **It's too big:** This is a meaningful argument if we are performing file processing of some of the larger files, for example, Item Ledger, Value Entry, and so on. However, NAV's filtering capabilities are so strong that we should relatively easily be able to carve out manageable sized test data groups with which to work.
- **There's no data that's useful:** This might be true. However, it would be just as true for a test database, unless it were created expressly for this set of tests. By definition, whatever data is in a copy of the production database is what we will encounter when we eventually implement the enhancements on which we are working. If we build useful test data within the context of a copy of the production database, our tests will be much more realistic and, therefore, of better quality. In addition, the act of building workable test data will help us define what will be needed to set up the production system to utilize the new enhancements.
- **Production data will get in the way:** This may be true. If it is especially true, then perhaps the database must be preprocessed in some way to begin testing or testing must begin with some other database, such as Cronus or a special testing-only mockup. As stated earlier, all the issues that exist in the production database must be dealt with when we put the enhancements into production. Therefore, we should test in that environment. Overcoming such challenges will prepare us to do 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:** These are both good points, but don't have much to do with the type of database we're using for the testing. Both the cases are addressed by properly managing the setup of our test data and keeping incremental backups of our pretest and post-test data at every step of the way. SQL Server tools can assist in this effort. In addition, the C/AL Testability Tools are explicitly designed to support regression testing.

Remember, doing the testing job well is much less expensive than implementing a buggy modification and repairing the problems during production.

## **Testing techniques**

As experienced developers, we are already familiar with good testing practices. Even so, it never hurts to be reminded about some of the more critical habits to maintain.

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 that created the design or coded the modification. It would be best if one of the testers is an experienced user, because users seem to have a knack (for obvious reasons) of understanding how the modification operates compared to how the rest of the system acts in the course of day-to-day work. This helps us obtain meaningful feedback on the user interface before going into production.

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 beneficial.

The C/AL Testability Tools provide features to support testing how system functions deal with problem data. If possible, it would be good to have users help us define test data, and then use the Testability Tools to assure that the modifications properly handle the data.

After we cover the mainstream issues (whatever it is that the modification is intended to accomplish), we need to make sure our testing covers all boundary conditions. Boundary conditions are the data items that are exactly equal to the maximum, 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 very effective for uncovering data-related errors.

## **Deliverables**

Create useful documentation and keep good records of testing processes and results. Testing scripts, both human-oriented and C/AL Testability Tool-based, should be retained for future reference. Document 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. Briefly record the testing that was done. The scope of the record keeping should be directly proportional to the business value of the modifications being made and the potential cost of not having good records. All such investments are a form of insurance and preventive medicine. We hope they won't be needed, but we have to allow for the possibility that they might be needed.

More complex modifications should be delivered and installed by experienced implementers, perhaps even by the developers themselves. Small NAV 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 instructions (preferably written) on what to expect from the change, and written instructions on how to correctly apply the change. There must also be a plan and a clearly defined process to restore the system to its state prior to the change, in case the modification doesn't work correctly. As responsible developers, whenever we supply objects for installation by others, we must make sure that we always supply .fob format files (compiled objects) and not text (.txt) 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 go well, 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 will almost certainly require the capability for the support person to view the user's display. Without that, it will be much more difficult, time consuming, and frustrating for the two-way support personnel - user communication to take place. If it doesn't already exist, 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 other complex business application 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.

Complementing the ability to be customized is the ability to upgrade a customized NAV system. While not a trivial task, at least it is possible with NAV. In many other systems, the only reasonable path to an upgrade is often to discard the old version and reimplement with the new version, recreating all customizations. Not only is NAV unusually accommodating to being upgraded, but with each new version of the system, Microsoft has enhanced the power and flexibility of the tools it provides to help us do upgrades. In the Microsoft Dynamics NAV 2017 Development Shell, among other useful cmdlets, there is the **Merge-NAVApplicationObject** cmdlet. Refer to the Developer and IT Pro Help section: *Microsoft Dynamics NAV Windows PowerShell Cmdlets* for details on starting and using a Development Shell session (hint: use the **Search** box to find information on the Development Shell).

We may say, why should a developer care about upgrades? There are at least two good reasons we should care about upgrades. First, because our design and coding of our modifications can have a considerable impact on the amount of effort required to upgrade a system. Second, because as skilled developers doing NAV customizations, we might well be asked to be involved in an upgrade project. Because the ability to upgrade is important, and because we are likely to be involved one way or another, we will review a number of factors that relate to upgrades.

## Benefits of upgrading

To ensure that we are on common ground about why upgrading is important to both the client and the NAV Partner, the following is a brief list of some of the benefits available from an upgrade:

- 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, such as new operating system versions
- 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 not complete and not every benefit will be realized in any one situation.

## Coding considerations

The most challenging and important part of an upgrade is porting code and data modifications from the older version of a system to the new version. When the new version has major design or data structure changes in an area that we have customized, it is quite possible that our modification structure will have to be redesigned and, perhaps, even be recoded from scratch.

On the other hand, many times, the changes in the new product version of NAV don't affect much existing code, at least in terms of the base logic. If our modifications are done properly, it's often not very difficult to port custom code from the older version into the new version. By applying what some refer to as low-impact coding techniques, we can make the upgrade job easier and, thereby, less costly.

## Good documentation

In the earlier chapters, we discussed some documentation practices that are good to follow when making C/AL modifications. Here is a brief list of practices that should be followed:

- 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 functional purpose of the modification in a related Documentation trigger
- Include a description of the 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
- As much as possible, make all code self-documenting, using meaningful names for all data elements and functions, and breaking code segments into logical functions so that process flow is self-evident
- 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, but will review them here, in the context of coding to make it easier to upgrade. We won't be able to follow each of these, but will have to choose the degree to which we can implement low-impact code and which of these approaches fit our situation (this list may not be all-inclusive):

- Separate and isolate new code
- Create functions for significant amounts of new code that can be accessed using single codeline function calls
- Either add independent Codeunits as repositories of modification functions or, if that is overkill, place the modification functions within the modified objects
- 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 pages
- Copy, then modify the 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, as necessary

In any modification, we will have conflicting priorities regarding doing today's job in the easiest and least expensive way versus doing the best we can do to plan for future maintenance, enhancements, updates, and upgrades. 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.

## Supporting material

With every NAV system distribution, there have been some reference guides. These are minimal in NAV 2017. There are previously published guides available but, sometimes, we 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 or on a blog.

Be a regular visitor to websites for more information and advice on C/AL, NAV, and other related topics. The [dynamicsuser.net](http://dynamicsuser.net) and [www.mibuso.com](http://www.mibuso.com) websites are especially comprehensive and well attended. Other smaller or more specialized sites also exist. Some of those available as of the writing of this book are as follows:

- *Microsoft Dynamics NAV Team Blog*: <https://blogs.msdn.microsoft.com/nav/>
- *Mark Brummel Blog*: <http://markbrummel.wordpress.com/blog/>
- *Clausl Blog*: <http://clauslblog.wordpress.com/>
- *Waldo's Blog*: [www.waldo.be/](http://www.waldo.be/)
- *Vjekoslav Babic's Blog*: <http://vjeko.com/>
- *Alain Krikilion's Blog*: <http://krikinav.wordpress.com/>
- *Soren Klemmensen's Blog*: <http://klemmensen-public.sharepoint.com/>
- *Luc van Vugt's Blog*: <https://dynamicsuser.net/nav/b/vanvugt>

There are also a number of other useful blogs available. Look for them and review them regularly. The good ideas posted by the members of the NAV community in their blogs and on the NAV forums are generously shared freely, and often.

Finally, there are a number of books focusing on various aspects of Dynamics NAV published by *Packt Publishing* ([www.packtpub.com](http://www.packtpub.com)). Even the books that are about older versions of NAV have a lot of good information about developing with the NAV tools and applying NAV's functionality in a wide variety of application environments.

## Review questions

1. Client Add-ins must be written in what language? Choose one:
  - a) C#
  - b) VB.NET
  - c) A suitable .NET language
  - d) C/AL.NET
2. The C/Side Testing tools allow the implementation of regression tests. True or False?
3. 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?

4. Both, source code changes and setting Debugger Breaks can only be done in the C/AL Editor. True or False?
5. NAV 2017 modifications should always be delivered to customers in the form of text files. True or False?
6. Which of the following defines the Client Add-in feature? Choose one:
  - a) The ability to add a new client of your own design to NAV 2017
  - b) A tool to provide for extending the RTC User Interface behavior
  - c) A special calculator feature for the RTC client
  - d) A new method for mapping Customers to Contacts
7. Designing to minimize disk I/O in NAV is not important because SQL Server takes care of everything. True or False?
8. NAV's multi-language capability allows an installation to have multiple languages active at one time. True or False?
9. Custom C/AL code is not allowed to call functions that exist in the base Microsoft created for NAV objects. True or False?
10. 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
11. The Help files for NAV cannot be customized by Partner or ISV developers. True or False?
12. Which one of the following provides access to several libraries of functions for various purposes widely used throughout the NAV system? Choose one:
  - a) Codeunit 412-Common Dialog Management
  - b) Codeunit 408-Dimension Management
  - c) Codeunit 396-NoSeriesManagement
  - d) Codeunit 1-Application Management

13. 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?
14. 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
15. The NAV 2017 Debugger allows the value of Watched Variables to be changed in the middle of a debugging session. True or False?
16. The NAV 2017 Debugger runs as a separate session. True or False?
17. The C/Side Testing tools support which of the following? Choose four:
  - a) Positive testing
  - b) Negative testing
  - c) Automated testing
  - d) C# test viewing
  - e) TestIsolation (roll-back) testing
18. NAV 2017 includes a flexible multi-currency feature that allows transactions to begin in one currency and conclude in a different currency. True or False?
19. NAV does not support linked SQL Server databases. True or False?
20. Simple debugging can be done without use of the Debugger. True or False?

## **Summary**

We have covered many topics in this book with the goal of helping you become productive in C/AL development with Dynamics NAV 2017. Hopefully, you've found your time spent with us to be a good investment. From this point on, your assignments are to continue exploring and learning, enjoy working with NAV, C/SIDE, and C/AL, and to treat others as you would have them treat you.

*"We live in a world in which we need to share responsibility. It's easy to say "It's not my child, not my community, not my world, not my problem." There are those who see the need and respond. Those people my heroes."*

- Fred Rogers

*"Be kind whenever possible. It is always possible."*

- Dalai Lama

# Index

## A

Accounting Period Calendar 574  
Action Groups  
  about 257  
  ActionContainer, properties 258  
  design criteria 262  
  Navigation Pane Button actions 261  
Action Icon Library  
  reference link 260  
Action Menus  
  about 526  
  Action Designer 528  
  Action Groups 532  
  Configuration 534  
  Personalization 534  
  Ribbon Categories 532  
  WDTU Role Center Ribbon, creating 531  
ActiveX Data Objects (ADO) 607  
Activities Area 199  
Application Area property  
  reference link 244  
arrays  
  reference link 372  
Artificial Intelligence (AI) 19  
Assemble to Order (ATO) 16  
Assisted Setup 207  
Attribute data item  
  NodeType 552  
Automation Controller  
  about 607  
  usage 607

## B

Binary Large Objects (BLOBs) 136, 162  
Blank When Zero 306  
Boolean Variables

reference link 398  
bound pages 249  
BREAK function 462  
Brick 93  
Business Intelligence (BI) 18, 19

## C

C/AL (Client Application Language) 21  
C/AL code  
  adding, to report 415  
  completed report, testing 426  
  field validation, adding to table 409  
  modifications 408  
  output, viewing in Excel 427  
  Request Page, defining 424  
  user entered report options, handling 421  
C/AL Expression  
  operators 387  
  reference link 387  
C/AL functions  
  CONFIRM function 397  
  ERROR function 395  
  FIND functions 402  
  MESSAGE function 394  
  Record functions 400  
  STRMENU function 398  
  used frequently 393  
C/AL operators  
  arithmetic functions 390  
  arithmetic operators 390  
  Boolean operators 391  
  reference link 392  
  relational functions 391  
  relational operators 391  
C/AL programming language  
  about 23, 25  
  reference link 78

C/AL routines  
  callable functions 574  
  creating 573  
  Management codeunits 582

C/AL Symbol Menu  
  about 433  
  purposes 434  
  using 435

C/AL syntax  
  about 385  
  assignment 385  
  C/AL functions, used frequently 394  
  code, indenting 408  
  conditional statements 406  
  expressions 387  
  punctuation 385

C/SIDE programming  
  about 373  
  custom functions 376  
  modifiable functions 374  
  non-modifiable functions 374

C/SIDE Report Dataset Designer 293

C/SIDE Report Designer 282, 284

C/SIDE Report Properties 300

C/SIDE Test-Driven Development 603

CALCDATE function 448

CALCFIELDS function  
  about 453  
  versus CALCSUMS function 456

CALCSUMS function 455

callable functions  
  about 574  
  codeunit 358 574  
  codeunit 359 576  
  codeunit 365 578  
  codeunit 396 579  
  function models, for reviewing 581  
  function models, using 581

card pages  
  about 39, 201

FactBoxes 40

FastTabs 39

ShowMandatory field 40

CardParts 211

CASE-ELSE statement 459

Chart Control Add-in  
  reference link 248

charts  
  about 213  
  Chart Control Add-In 214  
  Chart Part 213  
  reference link 214  
  references 213

CLEARMARKS function 472

Client Add-ins  
  about 608  
  comments 629  
  construction 609  
  creating, for WDTU 611

Client/Server Integrated Development Environment (C/SIDE)  
  about 20, 21, 22, 346  
  data definition, modifying 364  
  extensions, using 363  
  naming conventions 366  
  Object Designer 347  
  object, compiling 365  
  object, saving 365  
  programming 373  
  text objects 362  
  useful practices 363  
  variables 367

cmdlets 645

code indentation 408

codeunit 358  
  DateFilterCalc 574

codeunit 359  
  CreatePeriodFormat function 577  
  FindDate function 576  
  NextDate function 577  
  Period Form Management 576

codeunit 365  
  Format Address 578

codeunit 396  
  NoSeriesManagement 579

Codeunit Designer  
  accessing 350

codeunit  
  about 67  
  reference link 67

Column properties 281  
ColumnFilters 282  
comma-delimited files 69  
comma-separated value 69  
Common Language Specification (CLS) 294  
complex data type  
    about 152  
    automation data type 154  
    data structure 153  
    data types 161  
    DateFormula 154, 156, 157, 158, 159, 160  
    Input data type 154  
    object data type 153  
    Output data type 154  
    references 161  
conditional statements  
    about 406  
    BEGIN-END compound statement 406  
    IF-THEN-ELSE statement 407  
CONFIRM statement 591  
Confirmation Dialog page 206  
Container controls 237  
Content Modifiable tables  
    about 129  
    System table 129, 130  
Coordinated Universal Time (UTC) 152  
COPYFILTER function 470  
COPYFILTERS function 470  
Cronus database 647  
Cronus Demonstration Database 606  
Cue source table  
    URL, for Design Pattern 517  
Cues 199  
CURRENTDATETIME function 445  
custom functions  
    about 376  
    creating 377

**D**

data conversions 449  
Data Flow diagram 311  
data structure  
    examples 145  
data types  
    about 148  
complex data types 152  
fundamental data types 148  
usage 162, 163  
data-focused design  
    about 637  
    data tables, designing 638  
    data validation, designing 639  
    data views, defining 638  
    reviewing 640  
    revision 640  
    user data access interface, designing 639  
DataItem properties 309  
DataItem triggers 311  
DataItemTableFilter 282  
Date and Time functions  
    about 444  
    CALCDATE function 448  
    CURRENTDATETIME function 444  
    DATE2DMY function 446  
    DMY2DATE function 447  
    DWY2DATE function 447  
    TIME function 444  
    TODAY function 444  
    WORKDATE function 445  
DATE2DMY function 446  
debugging  
    code, changing 603  
    in NAV2017 588  
    NAV 2017 Debugger 594  
    objects, exporting into text files 589  
    with CONFIRM statement 591  
    with DIALOG function 591, 592  
    with ERROR function 593  
    with MESSAGE statement 591  
    with text output 593  
DELETE function 466  
DELETEALL function 467  
Delta files  
    about 631  
    reference link 631  
Departments page 210  
development backups 69, 70  
DIALOG function 591  
disk I/O 642  
Displaying Charts Using the Chart Control Add-in

- 523
- DMY2DATE function 447
- document page  
about 40, 41, 202  
FastTabs 203  
List Part page 40
- documentation 69, 70
- DWY2DATE function 447
- E**
- efficient solution  
designing 641  
disk I/O 642  
locking 642
- Element data item  
NodeType 552
- enterprise resource planning (ERP) 13
- ERP system 14
- ERROR function 593
- EVALUATE function 452
- event-based triggers  
documentation 26  
functions 26
- events  
about 630  
references 630
- Excel  
output, viewing 427
- EXIT function 462, 463
- Extensions  
about 629  
creating 629  
events 630  
installing 633, 634  
page changes 629  
publishing 633  
reference link 630, 632  
setting up 634  
table changes 629  
verification 633  
WDTU extension, creating 630
- F**
- FactBox Area  
about 211
- CardParts 211  
ListParts 211
- FactBoxes 201
- factory default options 643
- FastTabs 201, 203
- Field controls 242
- field events  
about 145  
reference link 145
- Field Groups  
about 89, 90, 91, 92, 93  
Brick 93, 94
- field numbering 145, 146
- field properties 138, 139, 140, 141, 142, 143
- field triggers 144
- field validation  
adding, to table 409
- field  
about 136  
data structure, examples 145  
field events 145  
field numbering 145, 146  
field properties 137, 138, 139, 140, 141, 142, 143  
field triggers 144  
naming 146, 147  
variables, naming 146, 147
- FieldClass property, options  
about 164
- FlowField 164, 166, 167
- FlowFields, using for application 171, 173, 174, 175
- FlowFilter 167, 169
- FlowFilter, using for application 171, 173, 174, 175
- Normal 164
- FIELDERROR function 441
- fields properties 137
- FILTERGROUP function 471
- filtering  
about 176, 468
- CLEARMARKS function 472
- COPYFILTER function 470
- COPYFILTERS function 470
- FILTERGROUP function 471
- GETFILTER function 470

**GETFILTERS** function 470  
**MARK** function 472  
**MARKEDONLY** function 472  
**RESET** function 473  
**SETFILTER** function 469  
**filters**  
    controls, accessing 184  
    Development Environment filter, accessing 184  
    experimenting 177, 178, 180, 181, 182  
**financial management** 16  
**FIND** functions 402  
    **FIND ([Which])** options 404  
    reference link 403  
    SQL Server alternates 404  
**flow control**  
    about 456  
    **BREAK** function 462  
    **CASE-ELSE** statement 459  
    **EXIT** function 462  
    **FOR-DOWNT0** 458  
    **FOR-TO** 458  
    **QUIT** function 462  
    **REPEAT-UNTIL** 457  
    **SKIP** function 462  
    **WHILE-DO** 457  
    **WITH-DO** statement 460  
**FlowField**  
    about 164, 166, 167, 452  
    using, for application 171, 173, 174, 175  
**FlowFilter**  
    about 167, 168, 169  
    using, for application 171, 173, 174, 176  
**FOR-DOWNT0** 458  
**FOR-TO** 458  
**Format Address**  
    reference link 579  
**FORMAT** function 451  
**formatting functions**  
    about 449  
    **EVALUATE** function 452  
    **FORMAT** function 451  
    **ROUND** function 449  
**Fully Modifiable tables**  
    about 115  
    **Entry table** 119, 120, 121  
**Journal** 117  
**Master Data** 116  
**Posted Document** 126, 127  
**Register** 125  
**Singleton** 127, 128  
**Subsidiary tables** 122, 123, 124  
**Template** 118, 119  
**Temporary** 128  
**functional terminology**  
    batch 29  
    document 29  
    Journal 29  
    ledger 29  
    posting 29  
    register 29  
**functions** 67  
**fundamental data types**  
    **Date data** 151  
    **numeric data** 148, 149, 150  
    **String data** 150  
    **Time data** 151  
**G**  
**General Ledger Entry table** 33  
**GETFILTER** function 470  
**GETFILTERS** function 470  
**Git** 646  
**GitHub**  
    URL 358  
**Global symbols** 435  
**Globally Unique Identifier (GUID)** 162  
**Greenwich Mean Time (GMT)** 444  
**Group controls** 237, 241  
**H**  
**Help**  
    customizing 636  
    reference link 636  
**human resource management** 20  
**human resources (HR)** 20  
**Hungarian naming styles**  
    reference link 366

## I

Independent Software Vendor (ISV) 27, 78, 192  
inheritance 227  
INIT function 443  
InitValue property  
  assigning 104  
Input and Output functions  
  about 463  
  DELETE function 466  
  DELETEALL function 467  
  INSERT function 465  
  MODIFY function 465  
  MODIFYALL function 467  
  NEXT function, with FIND 464  
  NEXT function, with FINDSET 464  
  INSERT function 465  
Integrated Development Environment (IDE) 20, 346  
Integrated Scripting Environment (ISE) 631  
interactive capabilities, reports  
  not visible option 335  
  sorting 333  
  visible option 335  
interfaces  
  about 606  
  Automation Controller 607  
  linked data sources 608  
internal documentation 436, 439  
InterObject communication  
  about 473  
  through data 473  
  through function parameters 473  
  via object calls 474

## J

Journal/Worksheet pages 42

## K

keys  
  about 84, 86, 87  
  using 109

## L

license  
  limitations 25  
linked data sources 608  
List page  
  about 38, 200  
  for WDTU application 109  
ListParts 211  
ListPlus page 204  
Local Currency (LCY) 168  
Local symbols 435  
locking 642

## M

Make to Order (MTO) 16  
Make to Stock (MTS) 16  
Management codeunits 582  
Manifest XML file  
  about 631  
  reference link 632  
manufacturing 16, 17  
MARK function 472  
MARKEDONLY function 472  
MenuSuites 68, 210, 433  
Merge-NAVApplicationObject cmdlet 652  
Mergetool  
  about 590  
  URL 590  
MESSAGE statement 591  
Microsoft Connect  
  reference link 286  
Microsoft Visual Team Services  
  URL 358  
Microsoft Word 284  
Microsoft Word Report Layout 282  
MODIFY function  
  about 465  
  Rec 466  
  xRec 466  
MODIFYALL function 467  
MSDN library  
  URL 31  
multi-currency system 583  
multi-language system

about 582  
features 583  
references 583  
**Multilanguage Development**  
reference link 80

## N

Namespace Identifier 544  
naming guidelines  
    reference link 146  
**NAV 2017 Debugger**  
    about 588, 594  
    activating 595  
    attaching, to session 597  
Break Events, creating 598  
C/AL code, viewing in window 600  
Ribbon Actions 600  
**NAV 2017 ERP system**  
    Artificial Intelligence (AI) 19  
    Business Intelligence (BI) 18, 19  
    financial management 16  
    human resource management 20  
    manufacturing 16, 17  
    project management 20  
    Relationship Management (RM) 19  
    reporting 18, 19  
    supply chain management (SCM) 17, 18  
**NAV 2017**  
    about 14  
    C/AL programming language 23, 25  
    C/SIDE Integrated Development Environment  
        21, 22  
    complex data type 136  
    constant 136  
    data type 136  
    debugging 588  
    definitions, used 136  
    ERP system 14  
    functional terminology 29  
    fundamental data type 136  
    object and system elements 25  
    Object Designer tool icons 22  
    object types 21  
    overview 20  
    page structure 195, 198  
report, creating 311  
user interface (UI) 29, 30  
variable 136  
**NAV Application Server (NAS)** 608  
**NAV design pattern**  
    references 24  
**NAV Design Patterns Repository**  
    reference link 573  
**NAV development projects**  
    data-focused design 637  
    guidance 637  
    knowledge 637  
    Posting processes, designing 640  
    supporting processes, designing 640  
    verifying 641  
**NAV forums**  
    references 591  
**NAV process flow**  
    about 506  
    data maintenance 510  
    data preparation 508  
    data, maintaining 509  
    data, utilizing 509  
    initial setup 508  
    Journal batch, posting 509  
    Journal batch, testing 509  
    transaction entry 508  
**NAV report data flow**  
    about 296  
    C/SIDE Report Properties 300  
    DataItem properties 309  
    DataItem triggers 311  
    report components, details 299  
    Report triggers 306  
    Request Page, properties 307  
    Request Page, triggers 308  
    Visual Studio Properties 303  
**NAV reports**  
    about 282  
    design 284  
    naming 292  
    types 287  
    types, summarization 291  
**NAV RTC** 629  
**NAV table**

modifying 113  
Navigate page  
about 207  
  Navigate function 207  
Navigate  
about 584  
modifying for 587  
Navigation Pane  
about 526, 537  
  Departments button 539  
  Home button 537  
  other buttons 540  
NAVx package  
creating 632  
negative testing 605  
NEXT function  
with FIND 464  
with FINDSET 464  
No. Series  
reference link 85  
Notifications  
reference link 395

## O

object and system elements  
control 26  
events 27  
field 25  
field numbers 27  
license 25  
object numbers 27  
properties 26  
record 26  
trigger 26  
work data 28  
Object Designer  
about 347  
MenuSuite Designer 352  
navigation 356  
object, creating 348  
object, exporting 357  
object, importing 358  
Query Designer 351  
Table object changes, importing 361  
tool icons 22

XMLport Designer 351  
object types  
about 21  
codeunit 21, 67  
MenuSuites 21, 68  
page 21  
queries 21, 68  
report 21  
reviewing 66  
table 21  
XMLports 21, 68, 69  
OData 19  
operators, C/AL  
about 387  
precedence hierarchy 392

## P

Page Actions  
about 253  
Action Groups 257  
guidelines 255  
subtypes 256  
types 256  
page changes  
reference link 630  
page components  
about 220  
inheritance 227  
Page Preview tool 226  
page properties 222  
page triggers 221  
Page Control triggers 249  
page controls  
about 234  
bound pages 249  
container controls 237  
field controls 242  
group controls 237, 241  
Page Control triggers 249  
Page Part controls 245  
unbound pages 249  
Page Design  
about 193  
guidelines 194  
Page Designer

about 215  
accessing 349  
New Page wizard 216

**Page Events**  
reference link 222

page names 214

**Page Part controls** 245

page parts

- FactBox Area 211

**Page Structure**  
about 195  
options 195, 198  
overview 193

pages, types  
about 198  
card pages 201  
charts 213

Confirmation Dialog page 206

Document page 202

List page 200

ListPlus page 204

Navigate page 207

page parts 211

Role Center page 199

special pages 208

Standard Dialog page 206

Worksheet (Journal) page 205

**pages**  
about 37, 38  
actions 194  
card pages 39, 40  
controls 194  
document pages 40, 41  
Journal/Worksheet pages 42  
list pages 38  
properties 194  
standard elements 38  
triggers 194

patterns  
reference link 415

Positive testing 605

**Posting processes**  
designing 640

Power BI 19

PowerShell

reference link 631

**Processing-Only reports** 340

project management 20

**Public Service Announcements (PSAs)** 32, 95

**Q**

queries  
about 68, 272, 278  
building 273  
reference link 68, 278

**Query Designer** 274

**Query properties**  
about 278, 279  
Column properties 281  
DataItem properties 280

**QUIT function** 462

**R**

radio station application

- Card page, creating 48, 49, 50, 52, 53
- Cronus demo database, using 31
- designing 32
- developing 31
- List page, creating 43, 44, 45, 46, 47, 48
- list report, creating 54, 55, 56, 58, 59, 60, 61, 62, 63, 64, 66
- sample data, creating 53, 54

scenario 31

table, creating 35, 36, 37

table, designing 33, 34

tables, using 33

rational normalization 510

**RDLC Report** 292

**RDLC Report Layouts**  
reference link 330

**READ function**  
reference link 272

**read-only table**  
about 130  
Virtual 131

**Record functions**  
GET function 402  
SETCURRENTKEY function 400  
SETFILTER function 401  
SETRANGE function 400

regression testing 604  
Relationship Management (RM) 19  
REPEAT-UNTIL 457  
report components  
  details 299  
  overview 292  
  Report Layout 296  
  report structure 293  
Report Dataset Designer  
  accessing 350  
Report Layout  
  reference link 331  
report structure  
  about 293  
  report data 293  
Report triggers 306  
report  
  C/AL code, adding 415  
  creating, in NAV 2017 311  
  creative plagiarism 340  
  experimentation 312  
  heading, laying out 416  
  inheritance 333  
  interactive capabilities 333  
  laying out 419  
  modifying, with Report Designer 328  
  modifying, with Word 328  
  patterns 340  
  phases 312, 316, 322  
  Processing-Only 340  
  reference link 341  
  references 318  
  Request Page 336  
  runtime, rendering 332  
  saving 417, 421  
  table data 418  
  testing 417, 426  
ReportDefinitionLanguageClient-side (RDLC) 285  
reporting 18, 19  
Request Page  
  about 209, 336  
  code, processing 424  
  defining 424  
  option, adding 337  
  properties 307  
  triggers 308  
RESET function 473  
Role Center activities pages  
  about 514  
  Cue Group Actions 519  
  Cue Groups 515  
  Cue source table 516  
  Cues 515  
Role Center Page Parts  
  about 522  
  for User Data 525  
  Page Parts Charts 523  
  Page Parts Not Visible 522  
Role Center pages  
  about 199, 510  
  Action Menus 526  
  Navigation Pane 526  
  Page Parts 522  
  structure 511  
  System Part 521  
Role Tailored Client (RTC)  
  about 29, 89, 185, 588  
  accessing 185, 186, 187  
ROUND function 449

## S

SalesHeaderShipTo 375  
secondary keys 110  
Secure Socket Layer (SSL) 555  
security filters  
  reference link 402  
Service Management (SM) 19  
SETAUTOCALCFIELDS function 454  
SETFILTER function 469  
Singleton table type  
  reference link 127  
SKIP function 462, 463  
source code management  
  references, for tools 440  
  with tools 440  
special pages  
  Departments page 210  
  Request Page 208  
SQL Joins  
  about 272

Join methods 272  
SQL Server Report Builder (SSRB) 284, 433  
SQLJoinType property 280, 281  
Stack image  
  about 515  
  URL 515  
Standard Dialog page 206  
Strong Key Name (SNK) 613  
Subform 215  
Subpage 215  
SumIndexField Technology (SIFT) 87  
SumIndexFields  
  about 87, 88, 452  
  CALCFIELDS function 453  
  CALCSUMS function 455  
  secondary keys 110  
  SETAUTOCALCFIELDS function 454  
  using 109  
supply chain management (SCM) 17, 18  
supporting processes  
  designing 640  
Synchronize Schema option  
  reference link 37  
Synchronizing Table Schemas  
  reference link 362  
System tables  
  example 129  
system updating  
  about 644  
  changes, handling 646  
  design 644  
  project recommendations, customizing 645  
  testing 646  
system upgrading  
  about 644  
  benefits 652  
  considerations, for code 653  
  documentation, creating 653  
  low-impact code, implementing 654  
  planning 651  
System-Defined Variables  
  reference link 373

**T**  
T-SQL queries 272  
table changes  
  reference link 630  
Table Designer screen  
  accessing 348  
TableRelation property  
  about 89, 111, 113  
  assigning 101, 103  
  using 109  
tables, types  
  about 115  
Content Modifiable tables 129  
Fully Modifiable tables 115  
read-only table 130  
tables  
  component 76, 77  
  creating 95, 97, 98, 99, 100, 101  
  Field Groups 89, 90, 91, 92, 93  
  field validation, adding 409  
  for activity-tracking, adding 105, 106  
  for WDTU application 106, 108  
  keys 84, 86, 87  
  modifying 95, 97, 98, 99, 100, 101  
  naming 77  
  numbering 78  
  overview 75  
  properties 78, 79, 80, 81  
  SumIndexFields 87, 88  
  trigger 83  
  triggers 81, 82  
  used, for tracking system data 130  
Task Scheduler 608  
Team Foundation system 646  
test-driven development 603  
testability framework  
  reference link 363  
TESTFIELD function 440  
testing  
  about 646  
  database, testing 647  
  documentation, creating 650  
  in production 647  
  project, concluding 651

techniques 649  
testing database, using 648  
**TestMethodType** property  
  handlers 605  
**Text data type**  
  Code data type, common properties 140  
**text objects** 362  
**TIME function** 445  
**TODAY function** 445

## U

**unbound pages** 249  
**undefined date (0D)** 445  
**undefined time (0T)** 445  
**Universal Naming Convention (UNC)** 153  
**URN (UniformResourceName)** 544  
**user entered report options**  
  handling 421  
**user interface (UI)** 15, 29, 30, 138

## V

**validation functions**  
  about 440  
  **FIELDEROR** function 441  
  **INIT** function 443  
  **TESTFIELD** function 440  
  **VALIDATE** function 443  
**variables**  
  C/AL Globals 367  
  C/AL Locals 368  
  naming 146, 147  
  working storage variables 369  
**Version List**  
  documentation 114, 115  
**virtual Date table** 310  
**Visual Studio 2015 Community Edition**  
  URL 56  
**Visual Studio Report Designer (VSRD)** 433  
**Visual Studio**  
  about 284  
  Properties 303  
  reference link 284

## W

**WDTU application, validation logic**  
  Playlist Header validations 480  
  Playlist Line validations 487  
**WDTU application**  
  activity-tracking tables, adding 105, 106  
  enhancing 94, 475  
  Factbox Page, creating 495  
  function, creating for Factbox 491  
  InitValue property, assigning 104  
  keys, using 109  
  list pages 109  
  Listenership Ledger 106  
  NAV table, modifying 113  
  page, enhancing 228, 234, 250  
  Playlist Header 105  
  Playlist Item Rate 105  
  Playlist Line 105  
  Playlist Subpage, creating 483  
  Publisher 106  
  Radio Show 105  
  Radio Show Ledger 106  
  Radio Show Type 105  
  SumIndexFields, using 109  
  table fields, modifying 476  
  TableRelation property, assigning 101  
  TableRelation property, asssigning 103  
  TableRelations property, using 109  
  tables 106, 108  
  tables, creating 95, 97, 98, 99, 100, 101  
  tables, modifying 95, 97, 98, 99, 101  
  validation logic, adding 480  
  Version List, documentation 114, 115  
**WDTU extension**  
  App ID, remembering 632  
  creating 630  
  Delta files, creating 631  
  Manifest XML file 631  
  NAVx package, creating 632  
  PowerShell, loading 631  
**web services**  
  about 555  
  disadvantages 555  
  enabling 558

exposing 556  
published object, determining 559  
publishing 558  
WHILE-DO 457  
WinForms control 609  
WITH-DO statement 460  
Wizard pages 207  
WORKDATE function 445  
Worksheet (Journal) page 205  
WSDL (WebServicesDescriptionLanguage) 560

## X

XML (eXtensible Markup Language) 68, 540  
XMLport line properties  
about 547  
SourceType as Field 551  
SourceType as Table 549

SourceType as Text 548  
XMLport line triggers  
about 552  
for DataType as Field 554  
for DataType as Table 554  
for DataType as Text 553  
XMLport object 209  
XMLports  
about 68, 69, 540  
Attribute data item 552  
components 542  
data lines 546  
Element data item 552  
properties 542  
Request Page 554  
triggers 546  
web services integration example, for WDTU 561