
Jenzabar CX

CX System Reference



JENZABAR

Technical Manual

Copyright (c) 2001 Jenzabar, Inc. All rights reserved.

You may print any part or the whole of this documentation to support installations of Jenzabar software. Where the documentation is available in an electronic format such as PDF or online help, you may store copies with your Jenzabar software. You may also modify the documentation to reflect your institution's usage and standards. Permission to print, store, or modify copies in no way affects ownership of the documentation; however, Jenzabar, Inc. assumes no responsibility for any changes you make.

Filename: tmcxref

Distribution Date: 02/15/2002

Contact us at www.jenzabar.com

Jenzabar CX and QuickMate are trademarks of Jenzabar, Inc.

INFORMIX, PERFORM, and ACE are registered trademarks of the IBM Corporation

Impromptu, PowerPlay, Scenario, and Cognos are registered trademarks of the Cognos Corporation

UNIX is a registered trademark in the USA and other countries, licensed exclusively through X/Open Company Limited

Windows is a registered trademark of the Microsoft Corporation

All other brand and product names are trademarks of their respective companies

JENZABAR, INC.
CX SYSTEM REFERENCE TECHNICAL MANUAL

TABLE OF CONTENTS

SECTION 1 – USING THIS MANUAL	1
Overview.....	1
Purpose of This Manual.....	1
Intended Audience.....	1
Product Differences.....	1
Structure of This Manual.....	1
SECTION 2 – JENZABAR CX OVERVIEW	3
Overview.....	3
Introduction.....	3
Background Knowledge.....	3
System Integration Features.....	4
Diagram.....	4
Product Integration.....	5
Enabling and Disabling Jenzabar CX Products.....	5
User Access of Products.....	5
Reporting and Output.....	5
SECTION 3 – JENZABAR CX SCHEMAS	7
Overview.....	7
Introduction.....	7
Access.....	7
File Naming Conventions.....	7
Creating Schemas Using Dbmake.....	8
Introduction.....	8
DBMAKE Environment variable.....	8
Dbmake Options.....	8
Alter Table Processing.....	9
Schema File Structure.....	10
Introduction.....	10
Schema Template.....	10
Database Name.....	11
Table Section.....	11
Column Section.....	11
Constraint Section.....	11
Index Section.....	12
Grant Section.....	12
Specifying Table Attributes in Schemas.....	13
Introduction.....	13
Table Attributes Section.....	13
Table Attributes.....	13
Specifying Columns in Schemas.....	16
Introduction.....	16
Column Section.....	16
Column Type.....	16
Default Values.....	17
Constraint Analysis in Schemas.....	18
Introduction.....	18
Column Section.....	18
Analysis Output.....	18

Constraint Format.....	19
Field Level Constraints.....	19
Table Level Constraints.....	19
Implementing Constraints.....	20
Unique Index/Constraint Conflict.....	20
Check Constraints Rebuilds.....	20
Specifying Triggers/Audit Trails in Schemas.....	21
Introduction.....	21
Tracking Changes.....	21
Tracking Changes Made by Programs.....	21
Grant/Trigger Section.....	21
Trigger/Audit Trail Section Syntax.....	22
Trigger Naming Convention.....	22
Trigger Processing.....	22
Audit Columns.....	22
CAPTURE FOR Clause.....	23
FOR CHANGE OF Clause.....	23
Triggered Actions.....	23
Audit Trail Permissions.....	23
Regular Triggers.....	23
The Process.....	24
Simple Trigger Example.....	25
Complex Trigger Example.....	25
Specifying Stored Procedures in Schemas.....	26
Introduction.....	26
Access.....	26
Stored Procedure.....	26
Procedure File Header.....	27
Privilege Definitions.....	27
Grant Option Privileges.....	27
Storage of Permissions.....	27
Input Parameters.....	28
Output Parameters.....	28
Compile Process.....	28
Example Schemas.....	29
Introduction.....	29
Basic Schema Example.....	29
Complex Schema Example.....	30
SECTION 4 – COMMON TABLES AND RECORDS.....	31
Overview.....	31
Using this Section.....	31
Accessing Tables and Records.....	31
What Is a SQL Table?.....	31
What Is a Jenzabar CX Table?.....	31
What Is a Jenzabar CX Record?.....	31
Disabled Fields.....	32
Locating Tables and Field Descriptions.....	33
Introduction.....	33
Fields By File Report.....	33
Files By Track Report.....	33
Fields By Track Report.....	33
Common Records.....	34
Introduction.....	34
Accomplishment Table.....	36
Purpose.....	36

How to Access	36
Screen Example	36
Field Descriptions	36
Report Example	37
ADR Table	38
Purpose	38
How to Access	38
Screen Example	38
Field Descriptions	38
Report Example	39
Alternate Address Table	40
Purpose	40
How to Access	40
Screen Example	40
Field Descriptions	40
Report Example	41
App Server Message Table	42
Purpose	42
How to Access	42
Screen Example	42
Field Descriptions	42
Report Example	43
Building Table	44
Purpose	44
How to Access	44
Creation Sequence	44
Screen Example	44
Field Descriptions	44
Report Example	45
Citizen Table	46
Purpose	46
How to Access	46
Screen Example	46
Field Descriptions	46
Report Example	46
Communication Table	48
Purpose	48
How to Access	48
Screen Example	48
Field Descriptions	48
Report Example	48
Configuration Table	50
Purpose	50
Changes to Table	50
How to Access	50
Screen Example	51
Field Descriptions	51
Report Example	51
Contact Table	53
Purpose	53
How to Access	53
Screen Example	53
Field Descriptions	54
Report Example	54
Country Table	56
Purpose	56

How to Access	56
Screen Example	56
Field Descriptions	56
Report Example	57
County Table	58
Purpose	58
How to Access	58
Screen Example	58
Field Descriptions	58
Report Example	59
Day Table	60
Purpose	60
How to Access	60
Screen Example	60
Field Descriptions	60
Report Example	61
Degree Table	62
Purpose	62
How to Access	62
Screen Example	62
Field Descriptions	62
Report Example	63
Denomination Table	64
Purpose	64
How to Access	64
Screen Example	64
Field Descriptions	64
Report Example	65
Division/Department Table	66
Purpose	66
How to Access	66
Creation Sequence	66
Screen Example	66
Field Descriptions	67
Report Example	67
Report Example	69
Entry Selection/Sort Criteria Table	70
Purpose	70
How to Access	70
Creation Sequence	70
Screen Example	70
Field Descriptions	70
Report Example	71
Ethnic Table	73
Purpose	73
How to Access	73
Screen Example	73
Field Descriptions	73
Report Example	74
Exam Table	75
Purpose	75
How to Access	75
Screen Example	75
Field Descriptions	75
Report Example	76
Facility Table	77

Purpose	77
How to Access	77
Screen Example	77
Field Descriptions	77
Report Example	78
Form Order Table	80
Purpose	80
How to Access	80
Screen Example	80
Field Descriptions	80
Report Example	81
Handicap Table	83
Purpose	83
How to Access	83
Screen Example	83
Field Descriptions	83
Report Example	84
Hold Tables	85
Purpose	85
How to Access	85
Creation Sequence	85
Hold and Hold Action Relationships	85
Absolute Holds	85
Office Permissions	85
Screen Example	86
Field Descriptions	86
Report Example	87
Report Example (Hold Action Table)	88
Report Example (Office Permissions Table Report)	89
ID Office Permissions Table	90
Purpose	90
How to Access	90
Screen Example	90
Field Descriptions	90
Report Example	90
Interest Table	92
Purpose	92
How to Access	92
Screen Example	92
Field Descriptions	92
Report Example	93
Involvement Table	94
Purpose	94
How to Access	94
Screen Example	94
Field Descriptions	94
Report Example	95
Marital Table	96
Purpose	96
How to Access	96
Screen Example	96
Field Descriptions	96
Report Example	97
Occupation Table	98
Purpose	98
How to Access	98

Screen Example	98
Field Descriptions	98
Report Example	99
Office Table	100
Purpose	100
How to Access	100
Screen Example	100
Field Descriptions	100
Report Example	100
Permission Table	102
Purpose	102
How to Access	102
Screen Example	102
Field Descriptions	102
Report Example	103
Privacy Act Tables	104
Purpose	104
How to Access	104
Creation Sequence	104
Screen Example	104
Field Descriptions	104
Report Example	105
Relationship Table	106
Purpose	106
How to Access	106
Screen Example	106
Field Descriptions	106
Report Example	107
State Table	108
Purpose	108
How to Access	108
Screen Example	108
Field Descriptions	108
Report Example	109
Subscription Table	110
Purpose	110
How to Access	110
Screen Example	110
Field Descriptions	110
Report Example	111
Suffix Table	112
Purpose	112
How to Access	112
Screen Example	112
Field Descriptions	112
Report Example	113
Tickler Table	114
Purpose	114
How to Access	114
Screen Example	114
Field Descriptions	114
Report Example	115
Title Table	116
Purpose	116
How to Access	116
Screen Example	116

Field Descriptions	116
Report Example	117
User ID Table	118
Purpose	118
How to Access	118
Screen Example	118
Field Descriptions	118
Report Example	119
Veteran Chapter Table	120
Purpose	120
How to Access	120
Screen Example	120
Field Descriptions	120
Report Example	120
Visa Table	122
Purpose	122
How to Access	122
Screen Example	122
Field Descriptions	122
Report Example	123
Zip Code Table	124
Purpose	124
How to Access	124
Screen Example	124
Field Descriptions	124
Report Example	125
SECTION 5 – JENZABAR CX MACROS	127
Overview	127
Introduction	127
What Is a Macro?	127
Configuration Table	127
The Relationship Among Macros, Includes, and C Programs.	128
Benefits of Jenzabar CX Macros	129
Introduction	129
Benefits of Jenzabar CX Macros	129
Contents of a Macro File	130
Introduction	130
Example Macro File	130
Parts of a Macro File	130
M4_Include Statements	131
The Four Types of Macro Files	132
Types of Macro Files	132
Macro Files That the Institution Can Customize	132
Macro Files That the Institution Should Not Customize	133
The Macro Directory Structure	134
How to Access the Macro Files	134
Macro Directory Structure	134
Custom Macro Files	135
Descriptions of Custom Macro Files	135
User Macro Files	137
Descriptions of User Macro Files	137
Common Macros	139
Introduction	139
Access	139
Enable Feature	139

Common Enable Macros	139
Common Periodic Macros	141
SECTION 6 – JENZABAR CX INCLUDES	149
Overview	149
Introduction	149
Policy Decision	149
Macro Dependency	149
How an Include Works	150
Relationship Between a Macro, Include, and C Program	150
Contents of an Include File	151
Introduction	151
Parts of an include file	151
How to Interpret the Include	152
Description of the Parts of an Include File	152
Examples of Includes	153
Introduction	153
Example of an Active Include Outside a Comment	153
Example of an Inactive Include Inside a Comment	153
Interpreting the Include Inside the Comment	153
Nine Types of Include Files	154
How to Access the Include Files	154
Include Directory Structure	154
Types of Include Files	154
Include Files That an Institution Can Customize	155
Include Files That an Institution Should Not Customize	155
Custom Include Files	156
Descriptions of custom include files	156
Common Includes	157
Setting Up Includes	159
What is the Process?	159
How to Set Up an Include	159
SECTION 7 - FORM ENTRY PROGRAM	161
Overview	161
Introduction	161
Program Features Detailed	161
Process Flow	162
Diagram	162
Data Flow Description	162
Program Relationships	163
Tables and Records Used	163
Parameters	164
Introduction	164
Parameter Syntax	164
Parameters	164
Operational Modes	165
Program Screens and Windows	166
Introduction	166
Access	166
Screen Files and Table/Record Usage	166
SECTION 8 – COMMON PROGRAMS	169
Overview	169
Introduction	169
Common Programs in this Section	169

ID Entry Program.....	170
Introduction.....	170
Accessing the ID Maintenance Feature.....	170
The ID Add for Individual Screen.....	170
Setup for this Feature.....	171
Results of Selecting the Add-ID Command.....	171
Duplicate ID Detection Program.....	172
Introduction.....	172
Dupid Terms.....	172
Program Arguments.....	174
Dupid Modes.....	175
Dupid Main Menu.....	175
Database Tables Used by Dupid.....	175
Modifying Table Definitions.....	176
Loading Data.....	176
Running Duplicate ID Detection in Background Mode.....	177
Introduction.....	177
Scheduling a Process.....	177
Starting from an Interactive Login.....	177
Dupid Configuration.....	178
Limitations.....	178
System Demands.....	178
Running Duplicate ID Detection in Interactive Mode.....	179
Introduction.....	179
Interactive Mode Screen Example.....	179
Data Displayed On The Screen.....	179
Fields Accessed with the Parameters Command.....	180
Initial Screen Commands.....	180
Screen Commands after Selecting Query ID or Input.....	181
Interactive Mode Detail Pop-up Window.....	181
Command Options for Detail Window.....	182
Running Duplicate ID Detection in Review Mode.....	183
Introduction.....	183
Review Mode Screen Example.....	183
Data Displayed On The Screen.....	183
Screen Commands.....	184
ID Audit Program.....	186
Introduction.....	186
The Process.....	186
Permissions.....	186
Idaudit Program Arguments.....	187
Running ID Audit.....	189
Processing Notes.....	189
The allow_delete Flag.....	191
Crash Recovery.....	191
Merge ID Program.....	192
Introduction.....	192
Merge Logic.....	192
Overview of the Process.....	192
What is an ID Column?.....	193
Merge ID Features.....	194
Merge ID Terms.....	194
Merge ID Tables and Records.....	197
Introduction.....	197
Configuration Macro.....	199
Running Merge ID in Interactive Mode.....	200

Introduction	200
Entering ID Pairs for Merge Processing	200
Merge ID Interactive Screen	201
Introduction	201
Data Displayed on the Screen	201
Commands on the Merge ID Interactive Screen	202
Merge ID List Screen	203
Introduction	203
Fields on the Merge ID List Screen	203
Merge Table List Window	204
Introduction	204
Column Descriptions	204
Expanded Merge Item Window	205
Introduction	205
Fields on the Expanded Merge Item Window	205
Running Merge ID in Batch Mode	206
Database Administration Program	207
Introduction	207
Program Arguments	207
Dbadmin Screen	208
Menu Options	208
Options Pop-Up Window	210
Options Pop-Up Window Fields	210
Audit Processing	210
Audit Scripts	211
Example Audit Script	212
Additional Table	212
Schedule Entry Program	213
Introduction	213
Windows Available in Schedule Entry	213
Records Used in Schedule Entry	213
Setup Issues for Schedule Entry	213
Sortpage Program	214
Introduction	214
Macros You Must Set	214
Sample ACE Report	216
Program Flow	217
Sortpage Processing	217
Sortpage Process Commands	218
Bulk Mailing Mode	218
Setting Up Bulk Mail Sorting	218
Program Error Messages	219
Crash Recovery	219
SECTION 9 – JENZABAR CX ENTRY LIBRARY	221
Overview	221
Introduction	221
Adding Tables for Use in Entry Library Programs	222
Introduction	222
Adding a Table	222
Adding a Detail Table	222
Displaying Specific Detail Table Rows on a Form	222
Adding a Lookup Table	222
Limiting the Number of Detail Tables in a Form	222
Entry Library Def.c Macros	223
Introduction	223

Def.c Macro Definitions	223
Example of Macros	223
Entry Library Def.c Variables.....	224
Introduction	224
Variables That You Can Specify	224
Example of Variables.....	225
Entry Library Def.c Local Functions	226
Introduction	226
Check Functions.....	226
Special Functions	226
Local Functions Example	227
Entry Library Def.c Program Parameters	228
Introduction	228
Parameter Types	228
Parameter Labels	229
Program Parameters Example	229
Detail Tables.....	230
Introduction	230
Def.c Scroll Tables Array.....	230
Example Scroll File Array	230
Tables for Entry Library Screens.....	231
Introduction	231
Filename Array Fields.....	231
Special Flags You Can Specify	231
Other Special Function Flags	232
Table Level Functions	232
Special Flag Example	233
Table Update Order.....	234
Introduction	234
Update Order Array Fields.....	234
Matching Entries in the Filename Array	234
Update Order Array Example	235
TABLENAME Array in an Entry Library Program	235
Table and Field Links	236
Introduction	236
Common Field Array Structure Definition.....	236
Common Field Array.....	236
Information for Loading Rows	236
Buffers for Binding Columns and Updating Records.....	237
Common Fields Array Example.....	237
Update Field Array.....	238
Update Field Array Example.....	238
Add Field Array.....	238
Add Field Array Example.....	238
Special Check Functions.....	239
Introduction	239
Check Field Array Fields	239
Check Functions You Can Specify	239
Check Function Array Example	240
Process to Process (PTP) Functionality.....	241
Introduction	241
Process to Process Field Structure	241
Specifying PTP Functionality.....	241
Process To Process Example	242
Address Maintenance.....	243
Introduction	243

Relationship Field Structure	243
Address Maintenance Example.....	243
GET_PRIMARY_REC Functions	244
Introduction	244
GET_PRIMARY_REC Processing	244
Suggestions for Writing a GET_PRIMARY_REC Function.....	244
File Type Structure Example	245
File Type Structure Members	245
IS_DISPLAY_ONLY Functions	247
Introduction	247
Determining a Column's Value.....	247
Check Functions.....	248
Introduction	248
Check Function Return Statuses.....	248
Check Function Parameters.....	248
Check Function Pointers	248
Special Functions	250
Introduction	250
Return Statuses	250
Events.....	250
Special Function Parameters	251
Special Function Example	251
Transaction Procedures	252
Introduction	252
Transaction Procedure Example.....	252
SECTION 10 – SCREENS AND FORMS.....	253
Overview.....	253
Introduction	253
Typical Entry Screens.....	253
Typical Detail Windows	253
Using the PERFORM Screen Commands	254
PERFORM screen commands.....	254
Creating Screen and Form Definition Files	256
Introduction	256
Screen Section	256
Types of Fields	256
Screen Section Features.....	256
Attributes Section	258
Attributes Section Format.....	259
Guidelines for the Attributes Section	259
Attributes You Can Specify.....	259
Instruction Section Format.....	267
Instructions You Can Specify	267
SECTION 11 – REPORTS AND OUTPUT CONTROL.....	269
Overview.....	269
Introduction	269
ACE Reports Sorting Program	269
ACE Report Writer Commands	270
Introduction	270
Running an ACE Report.....	270
ACE Commands.....	270
Defining Variables and Functions.....	273
Information Macros in the Define Section	273
Output Commands	274

Print Commands.....	274
Aggregate Commands.....	275
Pause Command.....	275
Skip Commands	275
Example ACE Reports.....	276
Introduction.....	276
Report to Print All Names in the Database.....	276
Select and Sort Report	276
Example SELECT and ORDER BY Reports	277
Formatting ACE Reports	278
Introduction.....	278
FORMAT Command Clauses.....	278
Page Headers.....	278
Page Trailers	279
ON LAST RECORD Statements	279
Troubleshooting ACE Reports.....	280
Introduction.....	280
Apparent problem with data.....	280
Core dump when translating the report.....	280
Core dump when the Ace report is run.....	281
Acearray Functions in ACE Reports	282
Introduction.....	282
Access to Acearray Functions.....	282
Summary List of Acearray Functions	282
Use of the Acearray Functions	283
Acearray Function Variables	283
The _vardef Function.....	284
_vardef Examples.....	284
The _varstore Function.....	285
_varstore Examples.....	285
The _varistore Function.....	285
_varistore Examples	285
The _varget Function.....	285
_varget Examples.....	286
The _variget Function.....	286
_variget Examples	286
The _varaccum Function.....	286
_varaccum Examples	287
The _variaccum Function	288
_variaccum Examples	288
The _varpct Function.....	288
_varpct Examples	288
The _varpctold Function	288
_varpctold Examples	289
Troubleshooting Array Functions	290
Sample Report.....	291
Sample Output.....	292
SQL Functions.....	293
Introduction.....	293
The _exec_sql Function	293
The _ctrl_trans Function.....	293
The _ctc_add Function.....	293
The _ctcdetl_add Function	294
Sample Report.....	294
Troubleshooting Use of SQL Functions	300
Introduction.....	300

Security Setup with SQL functions	300
Runreport Script: A Report Sorting Enhancement	301
Introduction	301
File Locations	301
WHERE and SORT Clauses	301
Example WHERE and SORT Clauses	302
Runreport Script	303
Runreport Processing	306
Script Menu Option File	306
Using Print Spooler Software	307
Introduction	307
The Spooling Process	307
Creating a Spool Queue	307
Spooler File Locations	308
SECTION 12 – THE MENU SYSTEM	309
Overview	309
Introduction	309
The Process for Placing a Screen or Report on the Jenzabar CX Menu	309
Menu Option (Menuopt) Files	310
Introduction	310
Example: Dean’s List Menuopt File	310
Menu Option Prompt	312
Menu Option Attributes	312
How to Create a Menuopt File	316
How to Modify a Menuopt File	317
Menu Description (Menudesc) Files	318
Introduction	318
Example: Jenzabar CX master menudesc file	318
Example: Jenzabar CX Master Menu	318
Menu Description Attributes	319
How to Create a Menudesc File	320
Menu Parameter (Menuparam) File	321
Introduction	321
Example	321
Menu Parameter Options	321
News and Mail Menu Features	323
Introduction	323
Different Mail Program	323
News Program	323
Nomenu Feature: Controlling Menu Access	324
Introduction	324
Nomenu Files	324
Menu Process	324
SECTION 13 - CX SYSTEM STANDARDS	325
Overview	325
Introduction	325
Use of the Standards	325
CX Data Dictionary Standards	326
Introduction	326
Definition	326
CX-Specific Definition	326
INFORMIX Data Files	326
ACE Reports	327
PERFORM Data Entry Screens	328

Application Software	328
Dictionary Data Schemas	328
Schema Definitions	328
Entering Data	329
Retrieving Data	329
Data Structure Standards	330
Introduction	330
Data Tables	330
Data Records	330
Comments	330
Names	330
Social Security Numbers	331
Phone Numbers	331
Data Integrity/Security	331
Data Dictionary Files	331
Standard Data Abbreviations	331
Program Name Abbreviations	333
Introduction	333
Files In BINPATH	333
Files In UTLPATH	335
Schema Standards	338
Introduction	338
Naming of Files and Fields	338
Location	338
Table or Record	338
Testing for Correct Naming Conventions	338
The Schema File	338
Header Section	339
Database Section	339
Table Section	339
Column Section	341
Text and Description	341
Common Types of Fields	342
Constraint Section	344
Index Section	344
Grant Section	344
Keys	344
Use of Indexes:	345
Naming of Suffixes	345
Standard Types and Lengths	345
Standard Schema Abbreviations	346
User Interface Standards: Menu Source	354
Introduction	354
General Conventions	354
Punctuation	355
Macros	355
User Interface Standards: Menu Options	357
Introduction	357
General Conventions	357
Punctuation	358
Macros	358
User Interface Standards: Program Screens	360
Introduction	360
General Conventions	360
Punctuation	363
Macros	363

User Interface Standards: PERFORM Screens	366
Introduction	366
General Conventions	366
User Interface Standards: Comment Macros	368
Introduction	368
General Conventions	368
PERFORM Screen Standards	369
Introduction	369
Access	369
Source Code: Documentation Header	369
Source Code: Format of screen	369
Source Code: Attributes	370
Source Code: Instructions (joins)	370
Compilation	370
Menu Definition	371
Testing	371
Support	371
Entry Library Screen Standards	373
Introduction	373
Introduction of Entry Library Features	373
Differences Between Libentry Screens and PERFORM screens	373
Locations for Forms and Detail Windows	374
File Naming Conventions	375
Screen Field Naming Conventions:	375
Screen File: The Attribute Section:	375
Tips for Creating Entry Screens	376
ACE Report Standards	377
Introduction	377
Definitions	377
Access	377
Source Code: Documentation Header	377
Source Code: Defined Variables, Parameters, and Functions	377
Source Code: Output definition	378
Source Code: Read Statements (Including JOIN, WHERE clauses)	378
Source Code: Sort Clauses	378
Source Code: Report Format	378
Compilation (Translation)	380
Executing ACE Reports	380
Menu Definition	381
Testing	381
Support	381
Menu Option Standards	382
Introduction	382
Menu Option Tags	382
Menu Option Attributes	384
Menu Option Standards	385
Related Scripts and Menu Option Testing	387
Testing a Menuopt	388
Menu Option Examples	388
Programming Style, Standards, and Conventions	394
Introduction	394
Design Guidelines	394
Program Design	394
Use of Standard CX Functions	394
Use of Transaction Processing	395
Audit on Summary Fields	395

ESQL Guidelines	395
Program Arguments	395
General Guidelines for Program Arguments	395
Program Arguments for Entry Programs	396
Naming Conventions	396
Variable Naming Conventions	397
Function Names	397
Common CX Files for Program Development	397
Building a Make File	399
Source File Organization	399
General Coding Structure Rules	400
Indentation	400
Braces	400
White Space	400
Comments	400
Compound Statements	401
Long Lines	402
Expressions and Constants	403
Syntax Changing	404
Embedded Assignments	404
Ternary Operator	405
GoTo Statements	405
Variable Definitions and Declarations	405
Function Definition	406
Function Header	407
Function Return Types and Parameters	407
Function Variable Declarations	407
Function Length	407
Function Endings	408
Functions and Macros	408
Portability	408
Separate Portable and Non-Portable Code	408
Avoid Dependence On Word Sizes	409
Specific Bit Representation	409
Special Character Expectations	409
Alignment Considerations	409
Boolean Testing	410
Numeric Values	410
Function Argument Evaluation Order	410
Project Dependent Standards	410
User Interface Standards	410
Screens	410
Output/Mail	412
Errors/Messages to User	412
Menus	413
Example C Code	414
Software Maintenance Standards	416
Introduction	416
Product Advisory	416
Product Issues	416
Program Documentation Standards	417
Introduction	417
Abstract	417
Introduction Section	417
Procedures Section	417
Parameters Section	417

Compilation Values.....	418
Program Flow Section	418
Program Errors Section	418
Crash Recovery Section.....	418
Database Input and Output Sections	419
Output Samples Section.....	419
INDEX	421

SECTION 1 – USING THIS MANUAL

Overview

Purpose of This Manual

This manual provides technical reference information that you can use to implement, support, and maintain the CX product. For specific information on implementing and maintaining the product, see the *CX Implementation and Maintenance Technical Manual*.

Intended Audience

This guide is for use by those individuals responsible for the implementation, customization, and maintenance of CX.

Product Differences

This manual contains information for using all features developed for the CX product. Your institution may or may not have all the features documented in this manual.

Structure of This Manual

This manual contains general reference information. The manual's organization follows:

Overview Information

- Section 1 - Information about using this guide
- Section 2 - Overview information about the module

System Reference Information

- Section 3 - Schemas
- Section 4 - Common Tables and Records
- Section 5 - Macros
- Section 6 - Includes
- Section 7 - Common Programs
- Section 8 - Entry Library
- Section 9 - Screens and Forms
- Section 10 - Reports and Output
- Section 11 - Menu System

Reference Information

- Index

SECTION 2 – JENZABAR CX OVERVIEW

Overview

Introduction

This section provides an overview to the features that integrate the product areas of CX. The products of CX are integrated by a shared relational database and system-wide features, such as the Common tables and records, and common programs.

Background Knowledge

The following lists and describes the necessary background information that you should know before using the features of CX described in this manual:

UNIX

Know the following about the UNIX operating system:

- Csh environment and commands
- Editor commands (e.g., vi)

INFORMIX-SQL

Know about the following INFORMIX tools:

- SQL database
- PERFORM screens
- ACE reports

QuickMate features

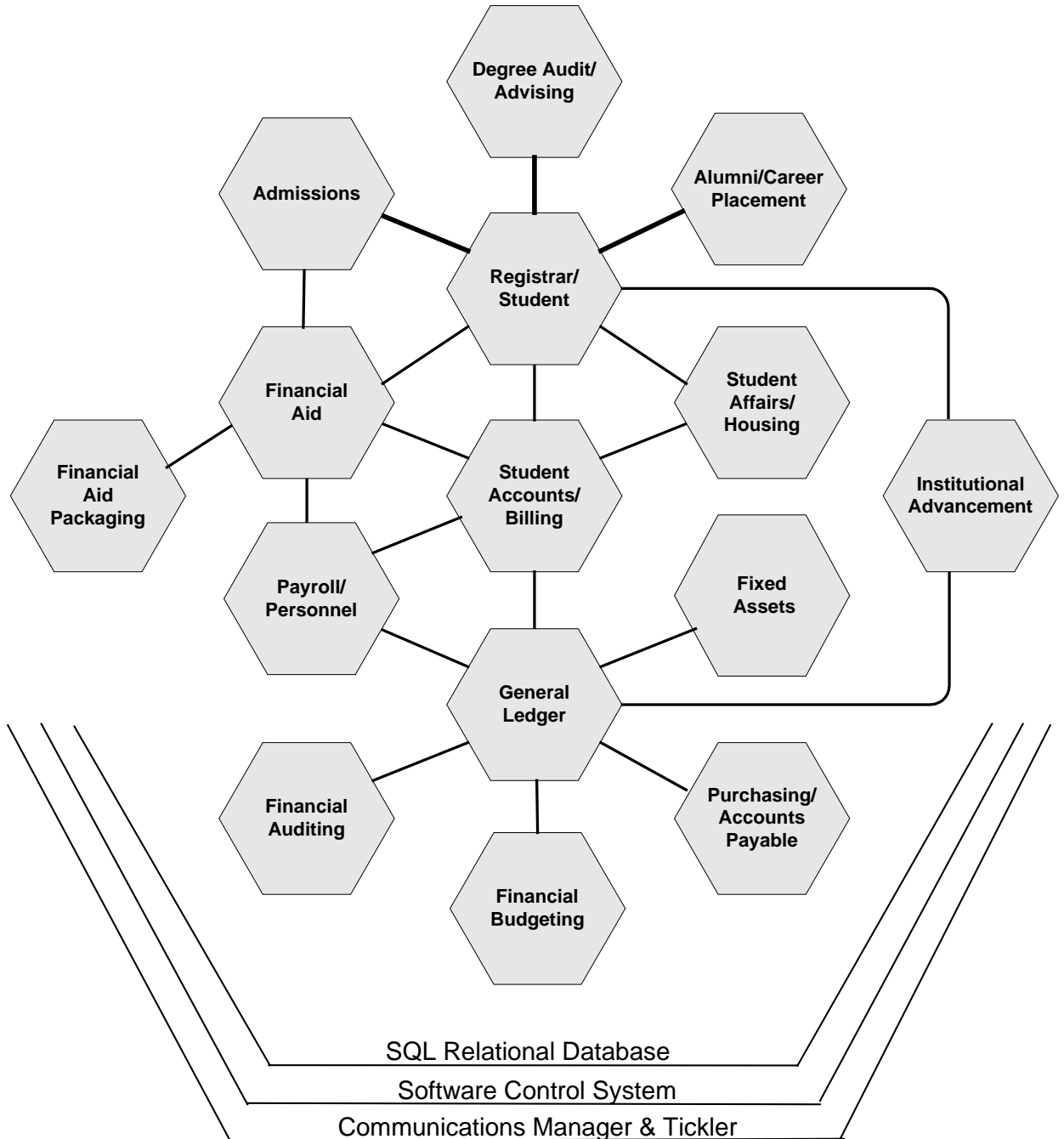
Know the following about the CX Graphical Server:

- Client/Server processing
- Network settings
- Keyboard settings
- Mouse settings
- GUI mode commands

System Integration Features

Diagram

The following diagram lays out the integration of CX.



Product Integration

The above graphic depicts the various products of CX. The connecting lines between products show the products' integration. This integration comes from programmatic associations and associations between records and tables in the shared database.

Programmatic Integration and Commonality

CX products can contain programmatic links between them. However, another aspect of the system's integration is the commonality of CX entry programs in the various products. Entry programs are designed using the Entry Library feature, which provides similar functionality between programs and access to a common library of tables and records. See *Section 8 - Entry Library* in this manual for more information.

Database Integration

CX product associations occur through the INFORMIX SQL Relational Database. Each product contains specific records pertaining to the area, such as Financial Aid records for the Financial Aid product. Other CX products can have access to a product-specific record. For example, the Registration product has access to a student's Financial Aid records. CX records can have links between them. You specify associations between CX records in schema files. See *Section 3 - Jenzabar CX Schemas* in this manual for more information.

Database integration between CX products also occurs through the use of the Common tables and records. All CX products use the same common table for common information. For example, every product has access to the State table for valid abbreviations of states. See *SECTION 4 – Common Tables and Records* in this manual for more information.

Enabling and Disabling Jenzabar CX Products

The above graphic displays the various integrated products of CX. Your institution, however, may or may not use all of the products displayed. CX includes tools for enabling and disabling products and product features, which are as follows:

- Macros
- Includes
- Common table: the Configuration table

See the appropriate sections in this manual for more information about these tools.

User Access of Products

Users access CX products through the CX menu system and screens and forms. The CX menu system provides a common access to all products of CX; however, with permissions and user login access types, you can set up the system so that a user accesses only that product area to which he has permissions.

- For more information, see *SECTION 12 – THE MENU SYSTEM* in this manual.
- For more information on user logins and permissions, see *Volume I - Implementation and Maintenance* of this manual.

All CX product screens and forms are designed and displayed through the CX screen package (SCR). For more information, see *SECTION 10 – SCREENS AND FORMS* in this manual.

Reporting and Output

CX products all contain reports and forms for correspondence that users can print. Output from reports and processes is controlled through CX spooler software. See *Section 10 - Reports and Output Control* in this manual for more information.

SECTION 3 – JENZABAR CX SCHEMAS

Overview

Introduction

This section describes CX schemas, their contents and maintenance. In this section, the following are described:

- The *dbmake* utility, which maintains schemas
- The structure of a schema file
- Table attributes in schemas
- Table columns in schemas
- Constraints
- Triggers
- Audit Trails
- Stored Procedures

Access

Schemas are located in the \$CARSPATH/schema directory path. Subdirectories under the schema path include each product area of CX, and can include the following:

- admissions (Admissions)
- common (Common)
- development (Institutional Advancement)
- financial (Financial Management)
- eis (Enterprise Information System)
- student (Student Academic Management/Financial Aid)

File Naming Conventions

CX makes name distinctions in the naming of schemas. For schema files containing definitions of CX tables, the UNIX file name begins with the letter *t* followed by characters describing the table's English name (e.g., *tst* for the State table). For schema files containing definitions of CX records, the UNIX file name describes the record's English name (e.g., *as id* for ID record).

Creating Schemas Using Dbmake

Introduction

CX makes use of the INFORMIX utility, *dbmake*, to maintain the structure of database tables. *Dbmake* reads the schema file and uses the information to:

- Create or modify the structure of the database table
- Grant users' access to the database table associated with the schema.
- Build audit trails, constraints, triggers, and stored procedures

DBMAKE Environment variable

When you call *dbmake* using the *make* utility, you can pass options, or flags, to *dbmake* using the environment variable DBMAKE. For example, if you want to see on your screen the SQL statements being executed as *make* runs on the State table (tst), you specify the "-v" flag.

Example: % cd schema/common
 % make build F=tst DBMAKE=-v

If you always want to see output, you can set the environment variable. This example uses C-Shell.

Example: % cd schema/common
 % setenv DBMAKE -v
 % make build F=tst

Dbmake Options

You can use several options to enable or disable process features of *dbmake*. The following specifies how you specify the options for *dbmake*:

Example: Usage:
 dbmake: [-abcinpqruvwxy] [file ...]

The following lists the *dbmake* options and what they signify to *dbmake*.

- a Only analyze table constraints
- b Do not build the table
- c Do not create or delete constraints
- i Do not create or delete indexes
- n Do not update dbfile, dbfield or dbattr tables
- p Do not modify the permissions
- q Print SQL statements to file named <file>.sql

- R** Always re-create the table on rebuild
- u** Do not update anything
- v** Print to standard out
- w** Wait until requested resource is available
- W** Suppress printing any error or warning message
- N** Suppress nested comments
- M** Do not force rebuild if only data location changes
- x** Print line number debug information
- y** Build without interactive prompt

Alter Table Processing

Unless you specify the **-R** option (force build), *dbmake* uses the *alter table* command to change a table instead of creating a new table and renaming it. The following table characteristics are addressed by the *alter table* command:

- Add column
- Drop column
- Change column size
- Change column type
- Change not null
- Change default value of column
- Change constraint on column

Note: You use the *rename column* command to rename columns.

The alter table command changes the table ID (*tabid*) of the table being processed. The following handles the changes to a table's *tabid*:

- INFORMIX handles the changing of *tabid* on its internal system tables.
- The *dbmake* program handles the changing of the *tabid* for internal *CX* tables.

Note: You cannot rollback the changes of the alter table and rename column commands in a transaction. Therefore, if one of the changes is not successful, the table can be left in an incomplete state.

The alter table command does not require exclusive access to the table being modified. The INFORMIX database recognizes the change and informs the application of the schema change in the form of an ISAM error.

Schema File Structure

Introduction

This section describes the contents of a schema file. Following an example of a template schema file, the sections of the file are described.

Schema Template

The following is the template schema file you use to define a database table.

Note: The following conventions apply to the following figure:

- *Upper case* words represent keywords and must be typed in lower case.
- *Lower case* words represent an identifier.
- *Square brackets* ([]) represent optional items.
- *Braces* ({}) indicate that one of the parameters enclosed by the braces and separated by the *pipe symbol* (|) must be included.
- *Double quotes* or *parenthesis* appearing in the syntax specifications are to be included.
- *Less than* (<) and *greater than* (>) symbols are used to indicate descriptive text and are not included.

```
{
    <Description information>
}
Revision Information (Automatically maintained by 'make' - DON'T CHANGE)
-----
$Header: ed,v 8.1 99/99/99 00:00:00 <uid> Developmental $
-----
}

[DATABASE database-name]

TABLE table-name
DESC "description string"
LOCATION "dbspace name"
LOCKMODE { ROW | PAGE }
PREFIX "prefix used for makedef"
ROWLIMITS { <integer> | ?? } : { <integer> | ?? }
STATUS "status"
TEXT "text description string"
TRACK "Track code"

COLUMN column-name [TYPE] <column-type>
[DEFAULT <default-value>] [NOT NULL]
[<constraint-def> CONSTRAINT constraint-name]
COMMENTS "comment string"
DESC "description string"
HEADING "heading string"
TEXT "text string"

...<More column definitions, if any>

CONSTRAINTS
<constraint-def> CONSTRAINT constraint-name

...<More constraint definitions, if any>

INDEX
[ UNIQUE ] index-name ON (column-name [DESC] [,
column-name [DESC] [ ... ])

...<More index definitions, if any>

GRANT
<access-type> TO ( user-name [ , user-name [ ... ] ] )

...<More access definitions, if any>

TRIGGER
```

```
[AUDIT ([<audit-column-list>]) [IN "<audit-server-name>"]
  [CAPTURE FOR (<action-type-list>)]
  [FOR CHANGE OF (<trigger-column-list>)]
  GRANT SELECT TO (<user-list> )
[ON INSERT {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
[ON UPDATE {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
[ON DELETE {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
```

Database Name

The first line in a schema file, after revision information, defines the database where the schema will be used. The entry of the database name must be preceded by the key word 'database' (e.g., **database cars**).

Table Section

In the TABLE section, you specify the INFORMIX database table that the schema defines. For example, *st_table* (State table) is specified in the *tst* schema file. Even though the INFORMIX software is not case sensitive, you should specify the table name in all lower-case letters to ensure that CX works correctly.

Column Section

In the COLUMN section, you define each field in the schema, including the following basic information about each field:

- The field name
- The type of field. Valid types include the following:
 - character (contains letters, numbers and symbols)
 - long (contains integers)
 - integer (contains whole numbers)
 - logical (single character column, expected to have a value of Y/N)
 - date (format mm/dd/yy)
 - serial (fields are assigned sequential serial numbers)
 - double (contains signed floating point numbers)
 - money (formatted as \$000.00)
 - float (contains single precision floating point numbers)
- The length of the field, specified in a parameter, for example: `addr1(24)`
- The description, heading, and text entries for the column, which are stored in the Database Field record (`dbfield_rec`)

Constraint Section

In the CONSTRAINT section, you specify constraints, which are rules that must be satisfied whenever a program attempts to insert, update, or delete data in a table.

Index Section

The INDEX section specifies those indexes required for:

- A specific application program
- An ACE report defining the primary index
- A composite join(s) in a PERFORM screen
- Creating data integrity (indexes without duplicates)

Note: The system assumes that indexes are ascending unless you declare the DESC (descending) attribute.

Grant Section

The GRANT section specifies access permissions for users and groups. Access types include:

- alter
- control
- delete
- index
- insert
- read
- select

You specify triggers and audit trails within this section.

Note: To help make merging easier, you should keep access-types in alphabetical order.

Specifying Table Attributes in Schemas

Introduction

You provide the name of the database table in the TABLE section. You also define the attributes of the table, such as the table's:

- Purpose
- Location in the database
- Associated track
- Size limits

Table Attributes Section

The following is the section of the schema template in which you specify table attributes.

Note: Keywords are presented in upper case for clarity; you should enter keywords in lower case.

```
TABLE table-name
      DESC           "description string"
      LOCATION      "dbspace name"
      LOCKMODE      { ROW | PAGE }
      PREFIX         "prefix used for makedef"
      ROWLIMITS     { <integer> | ?? } : { <integer> | ?? }
      STATUS        "status"
      TEXT          "text description string"
      TRACK         "Track code"
```

Table Attributes

The following lists the attributes you can specify for a table.

Note: For CX-supplied schemas, you generally do not need to change any of the attribute values in the TABLE section of the schema.

DESC Attribute

The DESC attribute indicates the purpose of the table. You can indicate the need for the table, or the reason for the information in the table.

LOCATION Attribute

The LOCATION attribute specifies the table's location in the database engine. The engine has several areas, called *dbspaces*, where tables can be located. You place a table in a dbspace for performance reasons, or because the dbspace is large enough to hold the table. Generally, you specify the location using a macro that begins with "DBS_". These macros are located in the macro file: \$CARSPATH/macros/custom/configure.

LOCKMODE Attribute

The LOCKMODE attribute specifies the mode that the engine will use to lock a record in the table.

Note: Generally, you should use ROW level locking for a table. However, in cases where a table is rarely modified, but many rows are changed at a time, you can use PAGE level locking as a way to control the number of locks used in the engine during the changes to the data in the table.

PREFIX Attribute

The PREFIX attribute specifies the table prefix used by the MAKEDEF utility to create the structures used by C code.

ROWLIMITS Attribute

The ROWLIMITS attribute specifies the initial size of the table, and the expected growth of the table. DBMAKE expects two integer values separated by a colon (:), or ?? to indicate using the default value for that integer. ROWLIMITS ??:?? indicates the default value for both initial size and expected growth.

As rows are added to a table, the database allocates disk space to the table in units called *extents*. Each extent is a block of physically contiguous pages in the dbspace. There are two types of extents: *initial* and *next*. The database allocates *initial* extent when the table is first created in the database. The database allocates a *next* extent whenever the current space for the table has been used up. The integers that you specify are used for calculating the size of extents in the database.

- The first integer indicates the *initial* number of rows that will be in the table when it is created and loaded. This number provides *dbmake* with the information to create a sufficiently large initial extent (when the table is created) to hold this number of rows in a single extent. This can reduce the possibility of the table using several extents just for the initial loading.
- The second integer indicates the maximum number of rows expected to be in the table. You can use this number to indicate the speed of growth for the table. For example, the State table (*st_table*) does not increase in size, so you can use the default value. The General Ledger Transaction record (*gltr_rec*) in the financial area will continue to increase in size, so you should indicate a large number for this second integer.

The default size of an extent is eight (8) pages, which translates to 16 KB on most platforms. Since the database engine does not automatically know table sizes, table space cannot be preallocated. Therefore, the database adds extents only as they are needed. The ROWLIMITS attribute provides a way to indicate the initial size of a table and to indicate the expected rate of growth of a table. Because the engine does have a limit for the number of extents a table may have, the ROWLIMITS attribute provides a way to automatically reduce the number of extents needed for the table.

Note: If you use a dbspace that does not have enough contiguous space for an extent of the specified size, the database engine allocates the largest available contiguous block for the extent. Thus, the engine can create extents that are smaller than your specified number in the schema. The *dbmake* utility attempts to use your specified initial number of rows (the first number in the rowlimits attribute value), or the actual number of rows in the table, whichever is larger, to calculate the size of the initial extent for the revised table when:

- You are rebuilding a table because you changed a field size or type
- You are rebuilding a table because you added or deleted a field
- You invoked the *dbmake* -R option

This feature provides a method for reducing the number of extents in use by the table for already existing rows, as long as a large enough contiguous space is available. It also maximizes the amount of contiguous space used for the table, which is more efficient for the applications accessing that table.

STATUS Attribute

The STATUS attribute specifies the status of the table as far as the programs are concerned. You generally set this attribute to *active*. For tables, which standard CX programs no longer access, set the attribute to *inactive*.

TEXT Attribute

The TEXT attribute provides a text string to specify a user-friendly name for the table.

TRACK Attribute

The TRACK attribute specifies the area in which the data is most applicable, for example, financial.

Specifying Columns in Schemas

Introduction

You define each column in a database table in the COLUMN section of a schema file. When you define columns, you specify:

- Column name and type (TYPE)
- Column default value (DEFAULT)
- Constraints (CONSTRAINT)
- Comments (COMMENTS)
- A description of the column (DESC)
- Field label (HEADING)
- Field level help information (TEXT)

This section provides the values that you can specify for the column type and default.

Column Section

The following is the section of the schema template in which you specify columns.

Note: Keywords are presented in upper case for clarity; you should enter keywords in lower case.

```
COLUMN column-name [TYPE] <column-type>
          [DEFAULT <default-value>] [NOT NULL]
          [<constraint-def> CONSTRAINT constraint-name]
COMMENTS      "comment string"
DESC          "description string"
HEADING       "heading string"
TEXT          "text string"

...<More column definitions, if any>
```

Column Type

The following lists the values you specify for the column type (e.g., <column-type>).

CHAR (<length>)

A character column of a specified length (<length>)

INTEGER

A 32 bit integer

SMALLINT

A 16 bit integer

SERIAL [(<start>)]

A computer-assigned sequential number starting at the specified number (<start>)

FLOAT

A binary floating point that is machine dependent

SMALLFLOAT

A binary floating point that is machine dependent

DATE

A date column

DECIMAL [(m [, n])]

A floating point number with specified significant digits (<m>) and specified places to the right of the decimal point (<n>).

MONEY [(m [, n])]

A floating point number with specified significant digits (<m>) and specified places to the right of the decimal point (<n>).

Note: <m> defaults to 17. <n> defaults to 2.

Default Values

The following lists the values you specify for the column default value (e.g., <default-value>).

<literal value>

A string or numeric constant

NULL

A NULL value.

Note: This is the default when you specify no default clause.

CURRENT

A date/time value based on the current system clock

USER

An eight character name of the user running the process.

CAUTION: Many of the CX applications run as the user *carsu* in which case the default USER is *carsu* and not the person's login name.

TODAY

The date value based on the current system date.

SITENAME

An 18 character database server name based on the currently selected database.

Constraint Analysis in Schemas

Introduction

A constraint is a rule that must be satisfied whenever a program (including isql and isql scripts) attempts to insert, update, or delete data in a table. This facilitates data integrity by defining, at the database level, what constitutes a valid value. The system performs constraint checking for every schema build.

The constraint process operates on a single table and checks for the following constraints applied to that table:

- Null values when *not null* is specified
- Duplicate values when unique constraint is specified
- Values not matching a foreign key
- Values not matching a value check

To run the constraint analysis, run *dbmake* using the *-a* option; this option provides constraint analysis only. To activate this feature, use a make target of *analyze*. The analysis contains the number of cases and number of rows that do not conform to a given constraint. The system prints the analysis to the *dbmake* output file (either <schema_file>.sql or <schema_file>.err).

Column Section

The following is the section of the schema template in which you specify field level constraints.

Note: Keywords are presented in upper case for clarity; you should enter keywords in lower case.

```
COLUMN column-name [TYPE] <column-type> [DEFAULT <default-value>] [NOT NULL][<constraint-def>]
CONSTRAINT constraint-name]
COMMENTS      "comment string"
DESC          "description string"
HEADING       "heading string"
TEXT          "text string"
```

Analysis Output

While building a schema, the system checks any existing data to ensure that the data meets the rules imposed by any constraints. If any data fails to meet the constraints' criteria, the build of the table will fail. You must run the *analyze* make target to show an analysis.

The system sorts the analysis by the constraint type. For multiple column constraints, the system separates columns that make up the constraint from the next violation by a blank line.

Single-column Constraint

```
Column name      Occurrences
Column name      Occurrences
...
```

Multi-column Constraint

Column name	Occurrences
Column name	
Column name	Occurrences
Column name	
...	

Constraint Format

You can impose constraints at the field level and at the table level.

Field Level Constraints

The following lists the constraint definitions (<constraint-def>) that you can specify:

UNIQUE

Enforces all values of this column to be unique.

PRIMARY KEY

Declares that this column is the primary key that could be used by another table's references constraint.

REFERENCES <table> [(column)]

Requires all values of this column to exist in the referenced table. If no column is specified, *dbmake* will use same name as the current column.

CHECK (<boolean-expression>)

Requires the column to satisfy the boolean expression. *Dbmake* does no syntax verification other than checking parenthesis levels.

Table Level Constraints

You define table level constraints in the schema file in the CONSTRAINTS section of the file. The CONSTRAINTS section immediately follows the column definitions and immediately precedes the INDEX section.

Remember that keywords are shown here in upper case for clarity but should be typed in lower case.

CONSTRAINTS

<constraint-def> CONSTRAINT constraint-name
...<More constraint definitions, if any>

<constraint-def> is one or more of the following:

UNIQUE (column-list)

Requires the data in the column list to be unique.

PRIMARY KEY (column-list)

The list of columns is used as a reference in another table

FOREIGN KEY (column-list) REFERENCES table [(col-list)]

References the specified table

CHECK (<boolean-expression>)

Requires the table to satisfy the boolean expression. *Dbmake* does no syntax verification other than checking parenthesis levels.

Implementing Constraints

For the initial installation of a foreign key, you do not need to check for primary keys if primary keys were applied in a SMO previous to the application of foreign keys. If you do check for primary keys, you can add a final case which checks for values in other tables which do not match the constraints.

Note: If you wish to add a primary or foreign key on a column which already has an established index, you must first drop that index. When the system builds the schema with the primary or foreign key defined, the system will recreate the index.

Unique Index/Constraint Conflict

The use of unique indexes and unique constraints are mutually exclusive. If you specify a unique constraint on a number of columns, you cannot specify a unique index on the same set of columns. This situation creates a potential conflict in the processing of constraints and indexes. Because constraints are processed before indexes, if you attempt to change a unique index into a unique constraint, the following error appears:

Example: "ISAM error: key already exists."

The above error occurs when the constraint processing attempts to add the unique constraint before the index has been dropped. In this error case, you need to do one of the following:

- Complete the make in two steps (first drop the index, and then add the constraint)
- Manually drop the unique index in ISQL.

Check Constraints Rebuilds

Dbmake drops and rebuilds check constraints during every schema build regardless of whether or not the check constraint has changed. The time required to rebuild a check constraint is insignificant compared with the effort otherwise to completely parse the check constraint body and recognize changes to the constraint.

Specifying Triggers/Audit Trails in Schemas

Introduction

Triggers provide a method of executing an action based on a triggering event, such as an insert, update, or delete that occurs on a table. Audit trails are complex triggers that are automatically generated by *dbmake*. While CX tables support full trigger functionality, to ensure the ability to provide audit trail capabilities, you must implement column level triggers as table level triggers using a WHEN condition.

Tracking Changes

The audit trail feature provides a means of selectively recording the insert, update, and delete changes that have occurred to a specific table. The system records changes in a special audit table created for each schema that has auditing specified. The audit table contains a standard set of columns that record:

- The date and time that the change occurred
- The login name of the person who made the change
- A flag to indicate the type of change

The audit table also contains specified database columns from the audited table that capture the values of the specific columns.

Note: For information on setting up the Audit Trail database, see *Setting Up an Audit Trail Database* in the *CX Implementation and Maintenance Technical Manual*.

Tracking Changes Made by Programs

Many of the CX programs run as *carsu*. In these cases, the *username* field in the audit trail indicates that the change was made by *carsu* and does not indicate the username that spawned the process.

Grant/Trigger Section

The following is the section of the schema template in which you specify field level constraints.

Note: Keywords are presented in upper case for clarity; you should enter keywords in lower case.

```
GRANT
    <access-type> TO ( user-name [ , user-name [ ... ] ] )
    ...<More access definitions, if any>

TRIGGER
    [AUDIT ([<audit-column-list>]) [IN "<audit-server-name>"]
    [CAPTURE FOR (<action-type-list>)]
    [FOR CHANGE OF (<trigger-column-list>)]
    GRANT SELECT TO (<user-list> )
    [ON INSERT {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
    [ON UPDATE {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
    [ON DELETE {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
```

Trigger/Audit Trail Section Syntax

You place the trigger/audit trail specification in the schema file in the GRANT and *synonym* sections, set apart with the TRIGGER keyword. The four basic categories of triggers are: UPDATE, INSERT, DELETE, and the special AUDIT trigger. You can validly use any combination of one or more of these trigger categories. *Dbmake* groups multiple triggers of the same category into one trigger.

Trigger Naming Convention

You create the trigger name by substituting the last character of the table name with the letters *u*, *i*, *d*, or *a*.

An example of the naming convention follows:

Update Trigger

id_rec id_reu

Insert Trigger

id_rec id_rei

Delete Trigger

id_rec id_red

Audit Trigger

id_rec id_rea

Trigger Processing

You can add trigger action statements into the schema in any order because *dbmake* sorts the action statements.

Dbmake drops and rebuilds triggers during every schema build regardless of whether or not the check constraint has changed. The time required to rebuild a trigger is insignificant compared with the effort to completely parse the trigger text and recognize changes to the constraint.

Audit Columns

The following lists the audit column definitions (<audit-column-list>) that you can specify:

<column-list>

A comma separated list of columns within the table

*

All columns within the table

* minus <column-list>

All columns within the table except those columns listed

Note: The audit-column-list determines which columns will be included in the audit table. An * indicates all columns are to be included.

Audit tables cannot contain text or BLOB fields; therefore, if the table that you are defining the audit trail for has a text or BLOB field, you cannot include all (*) columns. You can include all columns except for the text or BLOB fields, using the * *minus blob_field, text_field* clause.

CAPTURE FOR Clause

The following lists the CAPTURE FOR clauses (<action-type-list>) that you can specify:

Note: The CAPTURE FOR clause is optional. If you do not specify the clause, the system captures rows for each of the four conditions (insert, before update, after update, and delete). If you specify the CAPTURE FOR clause, you define exactly which of the conditions on which to perform audit processing.

insert

Create an audit record after inserting a record into the table

before update

Create an audit record with the before update image of the record

after update

Create an audit record with the after update image of the record

delete

Create an audit record before deleting a record from the table

FOR CHANGE OF Clause

The following lists the FOR CHANGE OF clauses (<trigger-column-list>) that you can specify:

Note: The FOR CHANGE OF clause is optional. If you do not specify the clause, the system sets the trigger-column-list to be the same as the audit-column-list. The trigger-column-list specifies which fields will trigger an update audit if their values are modified.

<column-list>

A comma separated list of columns within the table

*

All columns within the table

minus <column-list>

All columns within the table except those columns listed

Triggered Actions

The triggered-action <triggered-action> refers to a *Triggered Action Clause* as defined in the Informix manual, *Using Triggers*. Within the triggered action clause, the old and new values use the correlation name of "o" and "n", respectively.

Audit Trail Permissions

The GRANT SELECT TO clause is required and is used to provide select permissions on the audit trail table for the users specified. If you use the GROUP keyword, the user list will also support the specification of all users within a group.

Regular Triggers

The ON INSERT, ON UPDATE and ON DELETE clauses allow you to specify custom trigger action clauses that can be properly merged with the optional audit trail triggers. You can specify each type of clause multiple times to create complex action clauses. The *action* refers to one of the following trigger actions:

- BEFORE
- FOR EACH ROW
- AFTER

The Process

The following lists the process to implement and use triggers and audit trails in a table.

1. Dbmake combines together any specified AUDIT, ON INSERT, ON UPDATE, and ON DELETE clauses to build the Informix SQL trigger syntax.
2. Dbmake creates the audit table in a secondary database

Note: The secondary database must have transaction logging turned on.

3. Dbmake:
 - Names the table with the same name as the table being audited in the primary database.
 - Creates the audit table at the time the schema is built.

4. The base set of columns contain:
 - Date/time stamp of event
COLUMN audit_timestamp DATETIME
 - User name of the individual triggering the event
COLUMN audit_username CHAR(8)
 - Type of change indicator
COLUMN audit_event CHAR(2) CHECK VALUE
IN ("I", "U1", "U2", "D ")

5. Dbmake adds additional columns listed in the audit-column-list to the audit schema.

Note: All of these columns will maintain their original name and type with the exception of SERIAL types which will be changed to INTEGER type.

6. The system copies the old audit trail table into a table, based on the original name, and creates a new audit trail table, if one of the following occurs:
 - Columns are added or removed from the column list
 - The column type or size of a column in the column list changes

Note: This step ensures that old audit trails survive table changes. Also, since the content of the triggers is dependent upon the column list, column changes require the creation of new audit triggers.

7. The system makes insert, update and delete triggers to perform the action of adding the audit trail information.

Note: The new table does not require special insert permissions because triggers operate as user *informix*.

You can grant additional select permissions to allow non-DBA users to read an audit trail.

8. The system implements audit trails using table level triggers.

Note: The system supports column level triggers indirectly through the use of the WHEN clause on the table level triggers. This ensures good audit trail functionality without excluding the ability to perform column based trigger actions.

Simple Trigger Example

The following trigger specification section defines an audit trail only.

```
TRIGGER
AUDIT (*) GRANT SELECT TO (coord)
```

The system records a snapshot of all of this table's columns in the audit trail database.

- Since no CAPTURE FOR clause was specified, the system takes snapshots of the current row for each of the four conditions: after insert, before update, after update, and before delete.
- Since no FOR CHANGE OF clause was specified, a change in any column specified in the <audit-column-list> (*, thus any column in the table) triggers an update audit.
- The mandatory GRANT SELECT TO clause grants select permission for the audit table to user coord.

Complex Trigger Example

The following trigger specification section defines an AUDIT trail, an INSERT trigger, and an UPDATE trigger.

```
TRIGGER
AUDIT (id, fullname, ss_no, title, suffix, addr_line1, addr_line2, st, zip, phone, mail,
correct_addr, decsd)
  CAPTURE FOR (INSERT, AFTER UPDATE)
  FOR CHANGE OF (fullname, addr_line1, addr_line2, st, zip)
  GRANT SELECT TO (GROUP auditors, coord)
  ON INSERT BEFORE (EXECUTE PROCEDURE log_id_adder())
ON UPDATE FOR EACH ROW (WHEN (o.decsd != "Y" AND n.decsd = "Y")
  (EXECUTE PROCEDURE register_decsd(n.id)))
```

The audit trail records images of only the columns specified (id, fullname, ss_no, etc.).

- The CAPTURE FOR clause specifies that a record should be added to the audit trail table only upon an insert or after an update.
- The FOR CHANGE OF clause specifies that updates will be recorded in the audit table only if the update modifies the value of one of the fields listed (fullname, addr_line1, addr_line2, st, or zip).
- The GRANT SELECT TO clause grants select permission for the audit table to GROUP auditors and user coord.

An INSERT trigger is defined.

- Each time a record is added to this table, the system executes the stored procedure *log_id_adder* before executing the insert.

An UPDATE trigger is also specified.

- For each row of the table that is updated, the system executes the procedure *register_decsd*.

The WHEN clause allows the stored procedure to execute only when the value of the decsd field was changed from some other value to Y.

- The system passes the newly updated value of the ID field as a parameter to the stored procedure for each row that the procedure is executed on.

Specifying Stored Procedures in Schemas

Introduction

The stored procedure allows database programs to be precompiled and packaged within the INFORMIX database. The body of the stored procedure is composed of a mixture of SQL and Stored Procedure Language (SPL) statements that allow for receipt and return of parameters. For the body of the stored procedure, the system passes the syntax parsing and compiling of the stored procedure onto the INFORMIX database.

Each stored procedure is defined in a file stored in a directory under the \$CARSPATH/procedures path.

Note: Stored procedure code is stored by track rather than module since the code is closely linked to the data it services. The track subdirectories are same as those used for schema definition.

Access

Each stored procedure is defined in a file stored in a directory under the \$CARSPATH/procedures path. Stored procedure code is stored by track rather than module since the code is closely linked to the data it services. The track subdirectories is the same as those used for schema definition.

Note: You can quickly access development information because a multiple line description of the procedure is stored with the stored procedure.

Stored Procedure

The following is the format of a stored procedure file.

```
PROCEDURE          <procedure name>
PRIVILEGE          <privilege>
DESCRIPTION        "<description lines>"
INPUTS             <input parameters>
OUTPUTS            <output parameters>
NOTES              ["<note lines>"]

BEGIN PROCEDURE

.
. <standard procedure commands>
.

END PROCEDURE

GRANT
EXECUTE TO (GROUP <groupname-list>, <username-list>)
```

Procedure File Header

The header of a stored procedure file includes:

- Procedure name
- Purpose
- Inputs
- Outputs
- Notes
- Privilege

The header is organized to encourage a specific documentation style that ensures that program parameters and corresponding documentation stay in unison.

Privilege Definitions

The following lists the privilege definitions (<privilege>) that you can specify:

Note: If the procedure is called by a trigger, it is effectively executed by user *informix*.

DBA

The stored procedure will execute with DBA privileges.

OWNER

The stored procedure will execute with the owner's privileges allowing permissions to all objects referenced in the procedure.

Grant Option Privileges

When you assign DBA privilege to a stored procedure, remember the following about using the GRANT option in a schema:

- With the GRANT option, other users can run the procedure with DBA permissions
- Without the GRANT option, only a user with DBA permissions can run the procedure

When you assign OWNER privilege to a stored procedure, remember the following about using the GRANT option in a schema:

- With the GRANT option, other users have the owner's permissions to all objects referenced in the procedure
- Without the GRANT option, the user's object access is limited by normal database permissions

Storage of Permissions

The system stores procedure permissions in the Sysprocauth table, similar to the Systabauth table for table permissions. The *dbmake* program maintains the Sysprocauth table, based upon the permissions specified by the stored procedure source file. The *dbadmin* program reads the Sysprocauth table for the purpose of granting user and group level permissions for stored procedures.

Input Parameters

The following is the format of input parameters (<input parameters>).

```
<paramname1><paramtype1> "<param description1a>"
                        "<param description1b>"
                        ...
<paramname2><paramtype2> "<param description2a>"
                        "<param description2b>"
                        ...
...
```

Output Parameters

The following is the format of output parameters (<output parameters>).

```
<paramtype1> "<param description 1a>"
              "<param description 1b>"
              ...
<paramtype2> "<param description 2a>"
              "<param description 2b>"
              ...
...
```

Compile Process

The stored procedure compile process is different from that for other code in CX. Since the procedure is immediately created and stored into the database, no *local* execution of the procedure occurs. If you want to perform testing and debugging of procedures, you must perform such tasks *live* or on a testing system.

You compile stored procedure files using the *make build F=* target. When you attempt to install a stored procedure, the message, "No installing for stored procedures." appears, similar to schemas.

Example Schemas

Introduction

The following examples illustrate the features of the schema file, including:

- Table definition
- Column definition
- Constraints
- Index definitions
- Triggers
- Audit trail definition

Basic Schema Example

This sample schema contains an example of a table definition, column definition, not null, and column level primary key constraint.

```
{
  Example schema to demonstrate syntax.
}

table eqtyp_t
  desc          "Contains the descriptions of valid types"
               "of equipment items."
  location      "DBS_COMMON"
  lockmode      row
  prefix        "eqtyp"
  rowlimits     ???
  status        "Active"
  text          "Equipment Type Table"
  track         "COMMON"

column descr    char(15) not null
               primary key constraint eqtyp_constr
  comments      "This is the eqtyp descr comments string."
  desc          "Valid types of equipment items."
  heading       "Equipment Type"
  text          "Equipment Type"

grant
  alter    to ()
  control to (group carsprog)
  delete  to (group carsprog)
  index   to ()
  insert  to (group carsprog)
  select  to (public)
  update  to (group carsprog)
```

Complex Schema Example

The following sample schema contains examples of table and column definitions, table and column level constraints, index definitions, triggers, and an audit trail definition. The constraints include defaults, not nulls, check constraints, and a foreign key constraint which references the primary key defined above.

```
{
    Example schema to demonstrate syntax.
}

table equip_table
desc "Keeps track of equipment descriptions and history."
location "DBS_COMMON"
lockmode row
prefix "eqp"
rowlimits "??:??"
status "Active"
text "Equipment Table"
track "NONE"

column inv_no serial not null
comments ""
desc "Serial number uniquely assigned by the system when an"
"equipment record is added."
heading "Equipment"
text "Equipment Number"

column typ char(15) not null
references eqptyp_table constraint typ_constr
comments ""
desc "Type of equipment"
heading "Type"
text "Type"

column id integer default 0 not null
comments ""
desc "Identifies the person who currently has this piece of"
"equipment checked out. 0=checked in"
heading "ID"
text "ID Number Checked out to"

column due_date date
comments ""
desc "Date this piece of equipment is due to be returned."
heading "Due Date"
text "Equipment Due Date"

column fine money (4,2) default 1.00
comments ""
desc "Amount to be fined for each week this piece of equipment"
"is overdue."
heading "Weekly Fine"
text "Weekly Fine"

constraints
check ( id < 99999 ) constraint tbl_lvl_constr

index
memb_id on (id)
unique inv_no on (inv_no)

grant
alter to ( )
control to (group carsprog)
delete to (group carsprog)
index to ( )
insert to (public)
select to (public)
update to (group carsprog)

trigger
audit (*)
grant select to (auditor)
on update after
(EXECUTE PROCEDURE test1("12/07/1942",3))
on delete before
(EXECUTE PROCEDURE cleanup(id))
```

SECTION 4 – COMMON TABLES AND RECORDS

Overview

This section describes the common tables and records of CX. You use CX Common tables to specify your institution's codes for common values, such as name and address values, or degree codes. CX products use these Common table values in entry screens and detail windows. To ensure data entry consistency, a user can access the Common table entries in table lookup fields. Certain Common tables, such as the Configuration table, allow you to define the setup of CX features.

CX products use the Common records to provide integration between product areas. For example, your institution creates one ID record (*id_rec*) for each student. All CX products access the same ID record for the student.

Using this Section

The Common tables in this section are presented in alphabetical sequence. These are the Common tables contained in the CX base product. It is possible that you may not use all of the tables shown here, due to customizing the system to fit your specific needs. To find information about a specific table, look through the table of contents to find the page for the table you need.

Since the fields on a screen for a table can be customized by your institution, the definitions for the screen fields are presented in alphabetical sequence for ease of reference.

Accessing Tables and Records

You can access the common tables from the Systems Management: Table Maintenance Menu or from a specific module table maintenance menu.

You access common records in the various CX products' screens and detail windows.

What Is a SQL Table?

In a relational SQL database, a table is an organized set of any kind of data, regardless of its purpose for validation or information maintenance. The basic unit of organization of a table is a column, a category of data. A table can have multiple columns, and columns typically contain multiple rows of data.

What Is a Jenzabar CX Table?

CX makes name distinctions in the usage of database tables. A *table* in CX contains information that remains static and is denoted with the *_table* extension. For example, the State table, named *st_table*, contains the list of the United States of America. On CX menu, you can access most tables in *Table Maintenance* menus.

What Is a Jenzabar CX Record?

CX makes name distinctions in the usage of database tables. A *record* in CX is a table that contains information that changes on a regular basis and is denoted with the *_rec* extension. For example, the Alternate Address record, named *aa_rec*, contains any other addresses at which students can be contacted, such as a summer address. You access records in CX program screens, detail windows, and PERFORM screens.

Disabled Fields

Each Common table contains two fields, Active Date and Inactive Date, that are disabled on the table screens. You do have the option to enable these fields to fit your specific needs. The following list describes the fields.

Active Date

The date that this table entry is to become valid. Default value is "today's" date.

Example: 08/15/1996

Inactive Date

The date that this entry is to become invalid

Example: 08/15/1996

Locating Tables and Field Descriptions

Introduction

The System Management area of CX contains three reports that provide information about schemas and field descriptions in schemas. The System Management: Data Dictionary menu contains the reports.

Fields By File Report

The Database Fields report (*dbfield*) lists the fields in the database by table. You can specify the beginning and ending of an alphabetical range of table names to be included in the report.

Note: You can use wildcards to specify a range of table names. For example, to specify all tables names from a to m, specify a* and m* in the parameter screen for the report.

Files By Track Report

The Database Files report (*dbefile*) lists the tables in the database by track. You can specify the beginning and ending of a alphabetical range of track names to be included in the report. The track values you can specify include:

- A (Admissions)
- C (Common)
- D (Development and Donor Accounting)
- F (Fiscal and Accounting)
- M (Management)
- S (Student)

Fields By Track Report

The Fields By Track report (*dbetrack*) lists the tables and fields in the database for the tracks that you specify. The track values you can specify include:

- A (Admissions)
- C (Common)
- D (Development and Donor Accounting)
- F (Fiscal and Accounting)
- M (Management)
- S (Student)

Common Records

Introduction

Common records are records that are used by multiple products of CX. For example, most CX products use the ID Record (id_rec). The following lists the common records.

Accomplishment record (accomp_rec)

Provides details of an individual's accomplishment.

Addressee record (addree_rec)

Defines the preferred salutation and/or name line for an individual for a particular case. Also, saves changes to names and social security numbers.

Addressing record (adr_rec)

Describes the type of addressing to be performed for the specified run code, alternate address, and/or individual.

Alternate Address record (aa_rec)

Provides alternate address information for an individual.

Application License record (aplicense_rec)

Stores application licenses that clients have registered at the institution.

Business record (bus_rec)

Provides information about businesses that have a relationship with the institution.

Church record (church_rec)

Provides information about churches that have a relationship with the institution.

Contact Detail record (ctcdetl_rec)

Defines the handling requirements for letter contacts.

Contact record (ctc_rec)

Records the sending or receipt of correspondence with another individual.

Contact BLOB record (ctc_blob_rec)

Records additional free-form text associated with a ctc_rec.

Contact Image record (ctc_image_rec)

Records additional free-form text associated with a ctc_rec.

Database Field record (dbfield_rec)

Describes fields within the database. This is a CX table maintained by the *dbmake* program.

Database File record (dbfile_rec)

Describes files within the database. This is a CX table maintained by the *dbmake* program.

Education record (ed_rec)

Records attendance at another educational institution; includes class rank, gpa, and attendance dates.

Employment record (emp_rec)

Identifies the company and position held by an individual primarily for the purpose of donations and matching gifts.

Event record (evnt_rec)

Contains information concerning the scheduling and type of event planned.

Examination record (exam_rec)

Contains test results and examination dates.

Faculty record (fac_rec)

Contains faculty information and identifies those who are faculty members.

Forms Order record (formord_rec)

Maintains information on all form orders placed on the system.

Group Scheduling record (grp_schd_rec)

Contains information for use in scheduling group gatherings.

Hold record (hold_rec)

Identifies the type of hold to be applied to an individual.

ID Contact record (idctc_rec)

Used for batch updates to the ctc_rec.

ID record (id_rec)

Contains the names and general information of individuals and entities.

Note: Adding fields to the id_rec is not recommended. However, if you must add a column (field), add it to the end of the schema. You will then have to perform a number of reinstalls depending on the number of programs that use the id_rec.

Image Document record (im_doc_rec)

Used as the primary indexing table for the document processing system.

Interest record (int_rec)

Contains an individual's stated interests for use in selective or personalized mailings.

Involvement record (involve_rec)

Contains an individual's involvements for use in selective or personalized mailings.

Military record (milit_rec)

Contains the military status for an individual.

Organization record (org_rec)

Describes organizations which are not foundations or businesses.

Phone Call record (phcall_rec)

Records details of a telephone contact.

Profile record (profile_rec)

Contains personal information for individuals in the ID record.

Relationship record (relation_rec)

Identifies two individuals and the relationship between them.

Step Objective record (stepobj_rec)

Defines the objective for a particular step in a track within a defined tickler system.

Step Requirement record (stepreq_rec)

Used to define the requirements to activate a step for a specified track and tickler. (All requirements must be met.)

Temporary ID Data record (idtmp_rec)

Temporarily stores imported data while verifying duplication status.

Tickler record (tick_rec)

Contains information needed to place an individual on a specific tickler system and specifies the completion date.

Accomplishment Table

Purpose

The Accomplishment table (accomp_table) contains all the valid accomplishments that are to be used within the system. An accomplishment (not to be confused with an interest) is something an individual has earned. Dean's List is an example.

Note: Accomplishments, when entered through the accomp_rec, can be listed in the transcript body in the session/year they were achieved. Some institutions may want to treat teacher certification as an accomplishment for listing on a transcript.

How to Access

The screen file for the Accomplishment table is located in the following directory path:
\$CARSPATH/modules/common/screens/tacomp

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Accomplishment Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  Current  Master
Detail ...
Searches the active database table.          ** 1: accomp_table table**

          ACCOMPLISHMENT TABLE

Code.....[      ]
Description...[          ]
Display on Web.[ ]
```

Field Descriptions

The following describes the fields contained on the Accomplishment table screen.

Code

The code for an accomplishment recognized by your institution.

Example: DEAN (for the Dean's List)

Description

The description for the accomplishment code.

Example: Dean's List (for the code DEAN)

Display on Web

A Y/N field indicating whether or not you want this field to display on the Web Admissions Application.

Report Example

The following Accomplishment Table report may vary from your report format and content due to your institution's particular specifications.

Code		Text		Active / Inactive	
		W		Date	Date
	Blank text for testing	Y			
ATHL	Athletic	Y			
CLIN	Clinical Honors	Y			
DEAN	Dean's List	Y			
FORE	Forensics Award	Y			
HONR	National Honor Society	Y			
HWHO	Who's Who in American High Sch	Y			
LOA	Academic Leave of Absence	Y			
MCL	Magna Cum Laude	Y			
MERT	Merit List	Y			
PROB	** Academic Probation	Y			
WHO	Who's Who in Amer Col. & Univ.	Y			

ADR Table

Purpose

The ADR table (adr_table) contains the addressing run codes to be used by the system. These codes are very important to the system since they are used to advise the system of a code to be used if there is none specified in the program.

Note: The ADR run codes are very important to the ADR (address) process. There are some default features in the programs to advise the system of the code to be used if there is not one specified. In general, the default for regist is REGIST. In addition to REGIST, GRDRPT and TRANS should also be in the table.

How to Access

The screen file for the ADR table is located in the following directory path:
\$CARSPATH/modules/common/screens/tadr

You can access this table from the Systems Management Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following ADR Runcode Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM: Query Next Previous View Add Update Remove Table Screen
          ** 1: adr_table table **

          ADR RUNCODE TABLE
Code.....[      ]
Description [      ]
```

Field Descriptions

The following describes the fields contained on the ADR Runcode Table screen.

Code

The ADR (addressing) run code to be used.

Example: CKSLCT (for Check Select Process)

Description

The description of the ADR run code.

Example: Check Select Process (for the ADR run code CKSLCT)

Report Example

The following ADR Table report may vary from your report format and content due to your institution's particular specifications.

Runcode	Text
CKSLCT	Check Select Process
GIFTRCPT	Gift Receipt Process
JOINT	Joint - Formal
JOINTD	Joint - Formal Dups
JOINTDI	Joint - Informal - Dups
JOINTI	Joint - Informal
PARENT	Parent - Formal
PARENTD	Parent - formal - Dups
PARENTDI	Parent - Informal - Dups
PARENTI	Parent - Informal
PURCH	Purchasing Process
REGIST	Registration Process
SDSBILL1	sds bill test prior =1
SINGLE	Single - Formal
SINGLED	Single - Formal - Dups
SINGLEDI	Single - Informal - Dups
SINGLEI	Single - Informal
STMT	Statement Process
TRANS	Transcript Process
VOUCHER	Voucher /Journal Process

Location: /base1/carsdevi/modules/common/reports/tadr
Revision: G.101 02/14/92 17:30:15

Alternate Address Table

Purpose

The Alternate Address table (aa_table) contains all the codes used for other addresses for an individual, such as a business address.

How to Access

The screen file for the Alternate Address table is located in the following directory path:
\$CARSPATH/modules/common/screens/taa

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Alternate Address table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM: | Query Next Previous View Add Update Remove Table Screen ...
Searches the active database table.                ** 1: aa_table table**

                ALTERNATE ADDRESS TABLE

Code.....[   ]
Description...[   ]
Priority.....[   ]
Maintenance...[   ]
Email.....[   ]
Active Date...[   ]
Inactive Date..[   ]
```

Field Descriptions

The following describes the fields contained on the Alternate Address table screen.

Code

The code for an alternate address.

Example: BUS (for business address)

Description

The description for the code

Example: Business Address (for the code BUS)

Email

A Y/N flag indicating whether this alternate address type is an e-mail address. Use Y for yes or N for no. Default value is N.

Maintenance

A Y/N flag indicating whether this alternate address type should maintain PREV addresses. Use Y for yes or N for no. Default value is N.

Note: When you set this field to Y, the system saves previous address information with this alternate address type setting.

Priority

The priority for the alternate address. Zero (0) is the highest priority. Priority controls the order in which addresses are reviewed for appropriateness and selected for use. For example, if at a given time there are two appropriate addresses, the address with the lowest priority number will be selected.

Report Example

The following Alternate Address Table report may vary from your report format and content due to your institution's particular specifications.

```

Wed Apr 26 2000          CARS College          Page 1
14:25                  ALTERNATE ADDRESS TABLE REPORT        taa

```

Code	Text	Priority	Maint	Active /Inactive Email Date Date
ABBR	Abbreviated Address	1	N	
BILL	Billing Address	50	N	
BUS	Business Address	5	N	
CURR	Current Address	90	Y	
EMER	Emergency Address	0	N	
EML	Email address	1		Y
EMLK	Second e-mail address	2		Y
EMLT	Third e-mail address	3		
FOR	Foreign Address	2	N	
LOC	Local Address	20	N	
MATR	Matriculation Address	0	N	
PAR	Parent's Address	0	N	
PERM	Permanent Address	40	Y	
PREV	Previous Address	99	Y	
SUMR	Summer Address	15	N	
WINT	Winter Address	10	N	01/01/95

* - Record is inactive according to active/inactive dates
Location: /usr/carsdevi/modules/common/reports/taa
Revision: Developmental 04/22/95 07:05:02

App Server Message Table

Purpose

The App Server Message table (apsmsg_table) contains messages to be returned by an app server for given conditions.

How to Access

The screen file for the App Server Message table is located in the following directory path: \$CARSPATH/modules/common/screens/apsmsg.

You can access this table from the Systems Management: Table Maintenance Menu.

Screen Example

The following App Server Message table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM: Query Next Previous View Add Update Remove Table Screen ...
Searches the active database table.          ** 1: apsmsg_table table**

MESSAGE TABLE

App Server.....[register_aps      ]
Message Code....[ACST          ]
Description.....[Not elig to register  ]
Comment .....[You are not eligible to register. Yo]
Standard Comment.[You are not eligible to register. Yo]
```

Field Descriptions

The following describes the fields contained on the App Server Message table screen.

App Server

Identifies the app server to which this message applies. This is the name of the app server (e.g., register_aps).

Comment

Free format text blob used to store the actual message returned by the app server.

Description

Describes the error or condition that will result in this message.

Message Code

Code that identifies this particular message to the app server. This code, combined with the aps name provides the access to the proper message by the app server.

Standard Comment

Free format text blob used to store the standard CX message. Used for reference only.

Report Example

The following App Server Message Table report may vary from your report format and content due to your institution's particular specifications.

Msg Code	App Server	Description
ACST	register_aps	Not elig to register
Comment:	You are not eligible to register. Your academic status disallows it. Your academic status is:	
Std Comm:	You are not eligible to register. Your academic status disallows it. Your academic status is:	
BILLHOLD	stubill_aps	Hold on student bill SDS
Comment:	You have a hold that prevents release of a student billing statement.	
Std Comm:	You have a hold that prevents release of a student billing statement.	

Building Table

Purpose

The Building table (bldg_table) contains the building codes used by your institution. It defines all the buildings for a particular campus.

Note: If only one campus is to be used, it should be designated as MAIN.

How to Access

The screen file for the Building table is located in the following directory path:
\$CARSPATH/modules/common/screens/tbldg

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Creation Sequence

Considerations for the sequence of creating this table in relation to other tables are:

- Complete this table before the Facility table.

Screen Example

The following Building Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
Searches the active database table.                ** 1: bldg_table table**

                BUILDING TABLE

Campus.....[   ]
Building.....[   ]
Description....[           ]
IVR Phrase No..[   ]
```

Field Descriptions

The following describes the fields contained on the Building Table screen.

Building

The building code. It identifies a particular building on a particular campus.

Example: REED (for the Reed Hall of Science)

Campus

The campus code. It identifies the campus on which the building is located.

Example: MAIN (for the main campus)

Description

The building description. The name or description of the building.

Example: Reed Hall of Science (for the building code REED)

IVR Phrase No

The Interactive Voice Response (IVR phrase) number associated with this campus and building.

Report Example

The following Building Table report may vary from your report format and content due to your institution's particular specifications.

```

Fri Jan 26 1996
11:29
CARS College
BUILDING TABLE REPORT
Page 1
tblbg

```

Campus	Code	Text	Active Date	Inactive Date
	ARMY	Armory Bookstore		
	LILY	Art G. Lily Laboratory		
		Blank location		
	CARS	Cincinnati Ohio		
	PRCE	Franklin Pierce Hall		
	HEAV	Heavilon English Center		
	HORT	Horticulture Building		
	DORM	Main Dormitory		
	PSYH	Psychological Services		
	SMTH	Smith Mortuary		
	WEDE	Wedemeyer Hall		
	WMBY	Wembley Stadium		
	WKRP	Wes K. Reed Planetarium		
	WH	West Chester, Ohio		

* - Record is inactive according to active/inactive dates
Location: /disk06/carsdevi/modules/common/reports/tbldg
Revision: G.100.110.2 08/17/92 18:24:48

Citizen Table

Purpose

The Citizen table (citizen_table) contains codes for citizenship.

How to Access

The screen file for the Citizen table is located in the following directory path:
\$CARSPATH/modules/common/screens/tcitizen

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Citizen Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: citz_table table**

                CITIZEN TABLE
                Code.....[      ]
                Description...[      ]
```

Field Descriptions

The following describes the fields contained on the Citizen Table screen.

Code

The code for a specific citizenship.

Description

The description for the citizenship code.

Report Example

The following Citizen Table report may vary from your report format and content due to your institution's particular specifications.

Wed Apr 28 1993
10:35

CARS College
CITIZEN TABLE REPORT

Page 1
tcitz

Code	Text
USA	United States of America

* - Record is inactive according to active/inactive dates
Location: /disk07/carsbetai/modules/common/reports/tcitz
Revision: G.53 12/07/92 22:12:14

Communication Table

Purpose

The Communication table (comm_table) contains codes for each form of communication that your institution issues. For example, the code PHON represents a telephone communication.

How to Access

The screen file for the Communication table is located in the following directory path:
\$CARSPATH/modules/common/screens/tcomm

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Communication Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: comm_table table**
                COMMUNICATION TABLE
Code.....[      ]
Description....[      ]
```

Field Descriptions

The following describes the fields contained on the Communication Table screen.

Code

The communication code. The code identifying the type of communication.

Example: LETT (for letter)

Description

The description of the communication code.

Example: Letter (for the code LETT)

Report Example

The following Communication Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992
10:45

CARS College
COMMUNICATION TABLE REPORT

Page 1
tcomm

Code	Text
----	-----
BROC	Brochures
CAMP	Campus Visit
CARD	Card
DOCU	Document
LABL	Label
LETT	Letter
LTLB	Letter and Label
PACK	Packet
PERS	Personal Contact
PHON	Phone Call
REPO	Report

Location: /base1/carsdevi/modules/common/reports/tcomm
Revision: G.101 02/14/92 17:20:24

Configuration Table

Purpose

The Configuration table (config_table) allows you to enable a feature of CX. The Configuration table replaces the ENABLE_MACRO for every application of CX. When you want to make changes that enable or disable a feature of CX, you make entries to the Configuration table. You do *not* make changes directly to the ENABLE_MACRO.

Note: Macros for all new CX products, in addition to ENABLE_MACROS, also exist in the Configuration table.

Changes to Table

All enable macros in the Configuration table are also loaded in the \$CARSPATH/install/m4custom/tconfigmac.m4 file. All make processes include the tconfigmac.m4 file when expanding macros. The Configuration table contains a trigger that creates a new copy of the tconfigmac.m4 file whenever a change is made to the table.

How to Access

The screen file for the Configuration table is located in the following directory path:
\$CARSPATH/modules/common/screens/tconfig

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Configuration Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM: Query Next Previous View Add Update Remove Table Screen ...
Searches the active database table.          ** 1: config_table table**

product          [          ]
name             [          ]
value           [          ]
                [          ]
                [          ]
                [          ]
std_value       [          ]
                [          ]
                [          ]
                [          ]
comm            [          ]
```

Field Descriptions

The following describes the fields contained on the Configuration Table screen.

comm

The documentation about the feature and its use.

name

The name of the feature. For example, the AUTH_CROSSPROG is Registration Entry's feature that enables the existence of an Authorization record for a student.

product

The product name of the feature (e.g., REGIST for Registration Entry).

std_value

The standard CX setup value (e.g., y for Yes). This is the setting for this feature when CX delivers the standard CX product.

value

The value that enables the feature (e.g., y for Yes).

Report Example

The following Configuration Table report may vary from your report format and content due to your institution's particular specifications.

Fri Jan 26 1996
11:51

CARS College
CONFIGURATION REPORT
Sorted by product, name

Page 1
tconfig

REGIST product

AUTH_CROSSPROG
Value Y
System setup Y

AUTH_CRS_CAPACITY
Value Y
System setup Y

AUTH_CRS_CONFLICTS
Value Y
System setup Y

AUTH_CRS_REQUIREMENTS
Value Y
System setup Y

Location: /usr/carsdevi/modules/common/reports/tconfig
Revision: H.0 10/13/95 17:19:41

Contact Table

Purpose

The Contact table (ctc_table) contains the valid types of contacts that drive much of the software. The contents of this table are very important to many processes of the Registrar module, especially grade reports, student data sheets, and transcripts.

Note: Since the entire system is heavily contact driven, entries should be made to the table for the basic Registrar functions. These especially include SDS (student data sheets), TRANS (transcripts), and GRDRPT (grade reports).

The two primary contacts used for letter production are DEANLIST and PROBAT (probation).

Bulk Mail, Span Waived, and Reissued are normally set with an **N**.

Enrollment Status is normally left blank.

How to Access

The screen file for the Contact table is located in the following directory path:

\$CARSPATH/modules/common/screens/tctc

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Contact Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: ctc_table table**
                CONTACT TABLE
Code.....[          ]
Description..[          ]
Tickler.....[          ]
Comm Code...[          ]
Routing.....[          ]
Span Waived..[          ]
Reissued....[          ]
Ace Report...[          ]
Run Code....[          ]
Bulk Mail....[          ]

Document Tracking Type..[          ]
Enrollment Status..[          ]
```

Field Descriptions

The following describes the fields contained on the Contact Table screen.

Ace Report

The name of the ACE report using this contact.

Example: ltradmit

Code

The contact code. It identifies the communication to be sent or received.

Example: ACCLET (for acceptance letter)

Comm Code

The communication code. This code is used to identify the type of communication this contact is.

Example: LTLB (for letter and label)

Description

Describes the contact code.

Example: Acceptance Letter (for ACCLET)

Document Tracking Type

The optional document control type for this contact.

Enrollment Status

The enrollment status achieved with this contact.

Example: ACCEPTED (for a contact code of ACCLET - acceptance letter)

Reissued

Can this contact occur more than once? Use Y for yes or N for no. Default value is N.

Routing

Is this an incoming or outgoing contact? Use I for incoming or O for outgoing.

Run Code

The ADR run code for this type of contact.

Example: SINGLE

Span Waived

Is the normal span for days between contacts waived for this contact? Use Y for yes or N for no. Default value is N.

Tickler

The code for the tickler system which may be used with the described contact.

Example: ADM (for administration)

Report Example

The following Contact Table report may vary from your report format and content due to your institution's particular specifications.

Fri Jan 26 1996
12:03

CARS College
CONTACT TABLE REPORT
Tickler: ADM

Page 1
tctc

Active /Inactive Code Date	Text	Comm	Report	Run Code	Rt	Spn Wv Rs Dc	Enr_Stat	Date
-								
	ACCEPTED Acceptance letter	LETT		SINGLEI	0 N N			
	ACCPDT Accept date deadlines	LETT ltradmit		SINGLEI	0 Y N		ACCEPTED	
	ACC_LET Acceptance letter	LTLB ltradmit		SINGLEI	0 N N			
	ACK_LET App. Acknowledgement Ltr	LTLB ltradmit		SINGLEI	0 N N			
	ACTIVITY Activities letter	LTLB ltradmit		SINGLEI	0 Y N			
	ADMNOTE Student Not Admitted	LETT		SINGLEI	I N N			
	ADVISOR Advisor letter	LTLB ltr1bl		SINGLEI	0 N N			
	APPCALL Application phonecall	PHON		SINGLEI	0 N N			
	APPLIED Applied for admission			I			APPLIED	
	APPRECV Application received			I			APPLIED	
	ASKDPST Request deposit	LETT ltradmit		SINGLEI	0 N N			
	ASKFEE Ask for fee	LETT		SINGLEI	0 Y N			
	ASKVISIT Invite to campus	LETT ltradmit		SINGLEI	0 N Y			
	ATHLETIC Athletic activity	LETT ltradmit		SINGLEI	0 N N			
	BIRTHDAY Birthday Greetings Ltr.	LTLB ltradmit		SINGLEI	0 N N N			
	CALLED Call from student	PHON		SINGLEI	I N N		INQUIRED	
	CANCEL Sent acknowledgement	LETT ltradmit		SINGLEI	0 Y Y		CANCEL	
	CAREER Career options	LTLB ltradmit		SINGLEI	0 Y N			

* - Record is inactive according to active/inactive dates
Location: /disk06/carsdevi/modules/common/reports/tctc
Revision: G.100.110.1 08/17/92 18:26:30

Country Table

Purpose

The Country table (ctry_table) contains the valid country codes (US and territories, and international) for use within the system.

Note: Additions and deletions may be made as necessary. However, the table is adequate as it exists to begin using it.

How to Access

The screen file for the Country table is located in the following directory path:
\$CARSPATH/modules/common/screens/tctry

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Country Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: ctry_table table**
                COUNTRY TABLE
Code.....[   ]
Description...[   ]
Postal Code....[   ]
```

Field Descriptions

The following describes the fields contained on the Country Table screen.

Code

The country code. A code specifying a country.

Example: AUS (for Australia)

Description

The description of the country code.

Example: AUSTRALIA (for the code AUS)

Postal Code

The US Postal Service code for the country.

Example: AS (for Australia)

Report Example

The following Country Table report may vary from your report format and content due to your institution's particular specifications.

Code	Text	Postal Code
AFG	AFGHANISTAN	AF
AL	ALBANIA	AL
DZ	ALGERIA	AG
AS	AMERICAN SAMOA	AQ
AND	ANDORRA	AN
AO	ANGOLA	AO
AY	ANTARCTICA	AY
AC	ANTIGUA & BARBUDA	AC
RA	ARGENTINA	AR
AUS	AUSTRALIA	AS
A	AUSTRIA	AU
PO	AZORES	PO
BRN	BAHRAIN	BA
BNG	BANGLADESH	BG
BDS	BARBADOS	BB
B	BELGIUM	BE
BH	BELIZE	BH
DY	BENIN	DM
BD	BERMUDA	BD
BT	BHUTAN	BT
BOL	BOLIVIA	BL
RB	BOTSWANA	BC
BR	BRAZIL	BR
IO	BRITISH INDIAN OCEAN TER	IO
VI	BRITISH VIRGIN ISLANDS	VI

Location: /sysdisk/carsgSQL/modules/common/reports/tctry
Revision: G.100 08/21/91 19:32:35

County Table

Purpose

The County table (cty_table) contains valid county codes, as specified by your institution.

Note: Unless your institution is in the state of Kentucky, this table will have to be built entirely. If you want to have counties from more than one state, a schema will have to be developed to deal with the same county name in different states. Some institutions have defined their county codes with the letter of the state in the first position of the county code.

How to Access

The screen file for the County Table is located in the following directory path:
\$CARSPATH/modules/common/screens/tcty

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following County Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                         ** 1: cty_table table**
                COUNTY TABLE
Code.....[      ]
Description....[      ]
```

Field Descriptions

The following describes the fields contained on the County Table screen.

Code

The county code. A code specifying a county.

Example: ADAI (for Adair county)

Description

The description of the county code.

Example: Adair (for the county code ADAI)

Report Example

The following County Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992 10:48	CARS College COUNTY TABLE REPORT	Page 1 tcty
Code	Text	
-----	-----	
ADAI	Adair	
ALLE	Allen	
ANDE	Anderson	
BALL	Ballard	
BARR	Barren	
BATH	Bath	
BELL	Bell	
BON	Bond	
BOON	Boone	
BOUR	Bourbon	
BOYD	Boyd	
BOYL	Boyle	
BRAC	Bracken	
BREA	Breathitt	
BREC	Breckinrid	
BULL	Bullitt	
BUTL	Butler	
CALD	Caldwell	
CALL	Calloway	
CAMP	Campbell	
CARL	Carlisle	
CARR	Carroll	
CART	Carter	

Location: /disk15/carsdevi/modules/common/reports/tcty
Revision: G.101.110.1 03/04/92 10:22:33

Day Table

Purpose

The Day table (days_table) contains codes for each valid abbreviation for a day of the week. For example, Monday has the following valid abbreviations: M, Mo, Mon, MO, and MON.

How to Access

The screen file for the Day Table is located in the following directory path:
\$CARSPATH/modules/common/screens/tdays

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Day Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: tdays_table table**
                DAY TABLE
                Number.....[      ]
                Description..[      ]
                Abbreviation.....[  ]
                Abbreviation 2.....[  ]
                Abbreviation 3.....[  ]
                Upper Case Abbrev 2..[  ]
                Upper Case Abbrev 3..[  ]
```

Field Descriptions

The following describes the fields contained on the Day Table screen.

Abbreviation

A one character abbreviation for a day of the week.

Example: M (for Monday)

Abbreviation 2

A two character abbreviation for a day of the week.

Example: Mo (for Monday)

Abbreviation 3

A three character abbreviation for a day of the week.

Example: Mon (for Monday)

Description

The description for the specified day of the week.

Example: Monday

Number

The order of the specified day of the week.

Example: 1 (for Monday as the first day of the week)

Upper Case Abbrev 2

An upper case two character abbreviation for a day of the week.

Example: MO (for Monday)

Upper Case Abbrev 3

An upper case three character abbreviation for a day of the week.

Example: MON (for Monday)

Report Example

The following Day Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992 13:04	CARS College DAY TABLE REPORT					Page 1 tdays
Day	Text	Abbr1	Abbr2	Abbr3	Upabbr2	Upabbr3
0	Sunday	S	Su	Sun	SU	SUN
1	Monday	M	Mo	Mon	MO	MON
2	Tuesday	T	Tu	Tue	TU	TUE
3	Wednesday	W	We	Wed	WE	WED
4	Thursday	H	Th	Thu	TH	THU
5	Friday	F	Fr	Fri	FR	FRI
6	Saturday	S	Sa	Sat	SA	SAT

Location: /base1/carsdevi/modules/common/reports/tdays
Revision: G.101 02/14/92 17:20:45

Degree Table

Purpose

The Degree table (deg_table) should include all approved degrees for an institution (Associate, Baccalaureate, Master, etc.). Dependent on the registrar's desires, all IPEDS (Integrated Postsecondary Education Data System) completion programs may be entered in this table. CX IPEDS Completion report uses the values in this table.

Note: Normally this table will require updating only. In conversion, this table should include degrees that are no longer used. This, however, can create a problem because it could confuse the users as to what is valid or invalid. A suggestion is to add a distinctive character or word in the description (*, obsolete) or make the first position of the code a consistent character to specify obsolete degrees.

How to Access

The screen file for the Degree table is located in the following directory path:
\$CARSPATH/modules/common/screens/tdeg

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Degree Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  Current  Master
Detail ...
Searches the active database table.          ** 1: deg_table table**
                DEGREE TABLE
                Code.....[      ]
                Description.....[          ]
                Years.....[      ]
                Terminal Degree..[ ]
                IPEDS Level.....[ ]
```

Field Descriptions

The following contains descriptions of the fields contained on the Degree Table screen.

Code

The degree code. A code identifying the type of degree granted.

Example: BA (for Bachelor of Arts)

Description

The text describing the degree code.

Example: Bachelor of Arts (for the code BA)

IPEDS Level

Specifies the relative level of the degree that an undergraduate student can receive. The IPEDS GRS program enrollment update options uses this value to determine the highest degree earned by a student who has completed more than one degree. Valid codes include:

- 1 - Bachelor's degree (at least a four year degree)
- 2 - Degrees of at least two years but less than four years
- 3 - Completers of programs which are less than two years

Leave this field blank if none of the above conditions apply.

Terminal Degree

Is the degree a Baccalaureate, a first professional degree, as defined by FISAP specifications? Use Y for yes or N for no. Default value is N.

Years

The number of years expected to complete this degree

Report Example

The following Degree Table report may vary from your report format and content due to your institution's particular specifications.

Mon Jan 5 1998 15:08	CARS College DEGREE TABLE REPORT		Page 1 tdeg
		IPEDS	
	Code Text	Level	Years
	-----	-----	-----
	AAS Associate in Applied Sci		2
	ASB Assoc. Special Business		2
	BA Bachelor of Arts		4
	BME Bachelor of Music Educat		4
	BS Bachelor of Science		4
	MA Masters of Arts		2
	MBA Master of Business Admin		2
	MDIV Master of Divinity		2
	PHD Doctor of Philosophy		4

Denomination Table

Purpose

The Denomination table (denom_table) contains the religious denominations that apply to your institution.

Note: This table contains a wide range of denominations and normally is adequate for beginning the implementation process.

Note that this information is valuable to private, church affiliated colleges, but less important to public institutions.

How to Access

The screen file for the Denomination table is located in the following directory path:
\$CARSPATH/modules/common/screens/tdenom

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Denomination Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: denom_table table**
                DENOMINATION TABLE
Code.....[      ]
Description....[      ]
```

Field Descriptions

The following describes the fields contained on the Denomination Table screen.

Code

The denomination code. A code specifying a religious denomination.

Example: CATH (for Roman Catholic)

Description

The description of the denomination code.

Example: Roman Catholic (for the denomination code CATH)

Report Example

The following Denomination Table report may vary from your report format and content due to your institution's particular specifications.

Code	Text
ADVE	Advent Christian
AFME	African Meth Episcopal
ANGL	Anglican
APOS	Apostolic
ASSM	Assembly of God
BAHI	Bahai Faith
BAPT	Baptist
BIBL	Bible Church
BIBP	Bible Baptist
BPAM	Baptist, American
BPFW	Baptist, Free Will
BPST	Baptist, Southern
BUDH	Buddhist
CATH	Roman Catholic
CC	Church of Christ
CCCH	Ch of Christ, Christian
CCHO	Ch of Christ, Holiness
CG	Church of God
CGIC	Church of God in Christ
CH	Christian
CHAR	Charismatic
CHCU	Christ, Christian Union
CHMA	Christian & Mission All
CHME	Christian Meth Episcopal
CHRE	Christian Reformed

Location: /base1/carsdevi/modules/common/reports/tdenom
Revision: G.101 02/14/92 17:21:01

Division/Department Table

Purpose

The Division/Department table contains the divisions and departments used within the system. The Division table (div_table) has the major academic entities within the academic sector. An academic division has departments under it. The Department table (dept_table) contains those departments. For example, the division "Fine Arts" has the departments of "Art" and "Music" under it. The Division table and Department table function as master/detail relationships.

Note: The division code is part of the key for the Department table department code.

When adding/updating the department in the course record, the corresponding division code is automatically brought forward on the screen. It is suggested to leave one blank record for the Division table. This will be used for any departments that are not associated with a division or for all departments if the division concept is not used.

Before creating the Division/Department table, coordinate with the business and academic offices since academic departments are frequently considered financial cost centers. All affected offices should be in agreement as to "official" departments. Enrollment reporting and grade distribution statistics often depend on the department breakdowns. Certain applications which apply to divisions, such as the schedule of classes, can be produced in either department or division/department order.

How to Access

The screen file for the Division/Department table is located in the following directory path:
\$CARSPATH/modules/common/screens/tdivdept

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Creation Sequence

Create these tables in the following sequence: establish the Division Table first, then the Department Table to add all the departments for a given division

Screen Example

The following Division/Department Table screen may vary from your table format and content due to your institution's particular specifications.

```

PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
          ** 1: div_table table**

          DIVISION TABLE

          Division Code....[   ]
          Description.....[           ]

=====
          DEPARTMENT TABLE

          Department Code..
          Description.....

```

Field Descriptions

The following describes the fields contained on the Division/Department Table screen.

Department Code

A code that identifies an academic department.

Example: ART (for the Art Department)

Description (department)

The text that describes the department code.

Example: Biology (for the department code BIO)

Description (division)

The text that describes the division code.

Example: Language and Literature (for the division code LLIT)

Division Code

A code that identifies the division an academic department belongs to.

Example: LLIT (for the Language and Literature Division)

Report Example

The following Department Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992
13:08

CARS College
DEPARTMENT TABLE REPORT

Page 1
tdept

Code	Text	Division
AC	Accounting - Cent. Penn.	ACCT
ART	Art	ARTS
BIO	Biology	NSCI
BUS	Business/Economics	ACCT
CHE	Chemistry	NSCI
COA	Communication Arts	ARTS
DP	Comp Info Sys /Cent Penn	SSCI
DS	Dev. Studies - Cent Penn	SSCI
EDU	Education	EDUC
ENG	English	LLIT
EX	Communication/Cent. Penn	LLIT
HIS	History	SSCI
HOE	Home Economics	EDUC
HPR	Health, Phys. Ed. & Rec.	EDUC
INT	Internships	
LAN	Foreign Languages	LLIT
MC	Mass Media - Cent. Penn.	SSCI
MG	Management - Cent. Penn	ACCT
MM	Retail Mgmt - Cent. Penn	ACCT
MUS	Music	ARTS
NUR	Nursing	NSCI
PC	Purchasing Department	
PHI	Philosophy	SSCI
POS	Political Science	SSCI
PSY	Psychology	SSCI
REL	Religion	SSCI

Location: /base1/carsdevi/modules/common/reports/tdept
Revision: G.101 02/14/92 17:21:11

Report Example

The following Division Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992 13:08	CARS College DIVISION TABLE REPORT	Page 1 tdiv
Code	Text	
-----	-----	
ACCT	Business / Accounting	
ARTS	Fine Arts	
EDUC	Education	
HUM	Humanities	
LLIT	Language and Literature	
NSCI	Natural Science	
SSCI	Social Science	

Location: /base1/carsdevi/modules/common/reports/tdiv
Revision: G.101 02/14/92 17:21:18

Entry Selection/Sort Criteria Table

Purpose

The Library Entry programs have a feature that allows users to define the select and sort capabilities in an Entry Program detail window. A detail window with the sort feature contains the Sort command. The Entry Selection table (entsel_table) defines the name and the database record for the sort selection. The indicated database record corresponds directly with any detail window that accesses that database record. The Sort Criteria table (entselcrit_table) establishes how the system selects and/or sorts data in a detail window linked to the database record in the Table Name field of the entsel_table.

Note: The *admentry* program does not recognize this function of the entsel_table unless you do one of the following:

- Comment out the following line in the commonfld section of the def.c file:
{ "ctc_rec", "tick", NULL, "tick", PROG_BUFFER }
- Provide users with the ability to pass a different tickler code every time they access *admentry*

How to Access

The screen file for the Entry Selection/Sort Criteria table is located in the following directory path: \$CARSPATH/modules/common/screens/tentsel

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Creation Sequence

Create these tables in the following sequence: establish an Entry Selection table entry then add the corresponding Sort Criteria table entry.

Note: If you create multiple Sort Criteria table entries for a Entry Selection table entry, use AND or OR logical operators in each Sort Criteria table entry.

Screen Example

The following Entry Selection/Sort Criteria Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
Searches the active database table.                ** 1: entsel_table table**
                ENTRY SELECTION/SORT CRITERIA

Entry Selection Code.....[          ]
Description.....[          ]
Table Name.....[          ]
Program Name.....[          ]
Permission Code.....[          ]
Default.....[          ]

=====

Boolean Condition.....
Column Name.....
Relational Operator.....
Column Value.....
Sort Order.....
Descending.....
```

Field Descriptions

The following describes the fields contained on the Entry Selection/Sort Criteria Table screen.

Boolean Condition

The logical operator for multiple selection and sort tests. Use AND or OR.

Column Name

The column name in the detail window (scroll screen) for the selection and sort criteria

Column Value

The column value for selection criteria

Default

Is this entry considered the default (Y/N)? Use Y for yes or N for no.

Note: Only one Selection code can be a default for each table.

Descending

Should the column values be sorted in descending order? Use Y for yes or N for no. The default value is N.

Description

The description of the selection code

Entry Selection Code

The selection code (e.g., A&R_CTCS for Admissions and Registration Contacts)

Program Name

The program applicable to this selection code (e.g., stumentry for Student Entry)

Permission Code

The permission code indicating who can assess the selection code (e.g., REGIST for Registrar users)

Relational Operator

The relational operator for the selection and sort criteria. Leave blank if the criteria is sort only.

Sort Order

The sort order for the values selected from the column

Note: Enter a unique value per selection code

Table Name

The table name applicable to the selection code (e.g., ctc_table for Contact table)

Report Example

The following Entry Selection/Sort Criteria Table report may vary from your report format and content due to your institution's particular specifications.

Ethnic Table

Purpose

The Ethnic table (ethnic_table) contains the standard race codes that are used for reporting purposes.

Note: This table contains the valid ethnic codes and should need no change. While it is possible to add codes to the table, you must be careful because reports and macros may have to be modified to accommodate the additions. Reports can have a significant loss of student data unless they provide for counts of ethnic codes that are not the CX standard. Possibly an "Other" category may have to be added.

How to Access

The screen file for the Ethnic table is located in the following directory path:
\$CARSPATH/modules/common/screens/tethnic

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Ethnic Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: ethnic_table table**

                ETHNIC TABLE

Code.....[  ]
Description....[  ]
```

Field Descriptions

The following describes the fields contained on the Ethnic Table screen.

Code

The ethnic code. A code taken from HEGIS standards identifying an ethnic background.

Example: AS (for Asian/Pacific Islander)

Description

The ethnic description. It describes the associated ethnic code.

Example: Asian/Pacific Islander (for AS)

Report Example

The following Ethnic Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992 13:10	CARS College ETHNIC TABLE REPORT	Page 1 tethnic
Code	Text	
AM	Amer Indian/Alaskan Natv	
AS	Asian/Pacific Islander	
BL	Black	
HI	Hispanic	
NO	Non-Resident Alien	
UN	Unknown/Undecided	
WH	White, non-Hispanic	
Location: /basel/carsdevi/modules/common/reports/tethnic		
Revision: G.101 02/14/92 17:21:31		

Exam Table

Purpose

The Exam table (exam_table) includes all tests, measurement instruments, and examinations that an institution uses. It not only provides the valid types, but formats the various labels which are to appear when an exam record is added or updated.

This table is very important to the Admissions Office and should be coordinated accordingly.

Note: This table should have entries for every type of exam/test that your institution plans use. Typically this includes the tests that are used for admission purposes (ACT, SAT, GRE, GMAT, LSAT, MCAT, etc.).

The registrar may want to use this table and the corresponding exam record to account for certain tests for which waivers are granted or credit is received (speech proficiency, CLEP, AP exams, etc.).

How to Access

The screen file for the Exam Table is located in the following directory path:
\$CARSPATH/modules/common/screens/texam

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Exam Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  Current  Master
Detail ...
Searches the active database table.          ** 1: exam_table table**
                EXAM TABLE
                Exam.....[ACT ]
                Description...[American College Test           ]
                Label 1.....[English   ]
                Label 2.....[Math     ]
                Label 3.....[Reading  ]
                Label 4.....[Sci Reason]
                Label 5.....[Composite ]
```

Field Descriptions

The following contains descriptions of the fields contained on the Exam Table screen.

Description

A text description of the exam code. For example, "American College Test" for ACT.

Exam

The exam code. This code identifies the exam taken.

Example: ACT

Label 1

The label for the first examination category.

Example: English

Label 2

The label for the second examination category.

Example: Math

Label 3

The label for the third examination category.

Example: Reading

Label 4

The label for the fourth examination category.

Example: Sci Reason

Label 5

The label for the fifth examination category.

Example: Composite

Report Example

The following Exam Table report may vary from your report format and content due to your institution's particular specifications.

Mon Jan 5 1998	CARS College					
Page 1	EXAM TABLE REPORT					
15:33						
texam						
Code	Label 1	Label 2	Label 3	Label 4	Label 5	CA Code
ACT	English	Math	Reading	Sci Reason	Composite	
ACT1	Usage/Mech	Rhet Skill	Elem Algbr	Alg/Coor G	Pl Geo/Trg	
ACT2	Soc St/Sci	Arts/Litr				
ACTE	English	Math			Composite	
ACTP	English	Math	Soc Sci	Nat Sci	Composite	
AP	English	Math	Frgn Lang	History	Nat Sci	
CLEP						
GMAT	VERBAL	QUANT	TOTAL			
MATH	PRE					
MCAT						
PEP	PSYCH	MAT/CHILD	MED/SURG	COMM HLTH		
RVMA	RVC MATH	RVC MATH FLG				
RVRD	RVC READ	RVC READ FLG				
SAT	Verbal	Math	Reading	Vocabulary	TWSE	
SATI	Verbal	Math	Verbal %	Math %	Total	
SATS	Reading	Analogies	Sentences	Algebra	Geometry	
SATW	Sent Error	Imp Sent	Imp Para	Wr Sample		
TOEF	TOEFL	EXAM				
Location: /disk15/carsdevi/modules/common/reports/texam						
Revision: G.101.110.1 03/04/92 10:24:34						

Facility Table

Purpose

The Facility table (facil_table) contains codes for each facility on your campus.

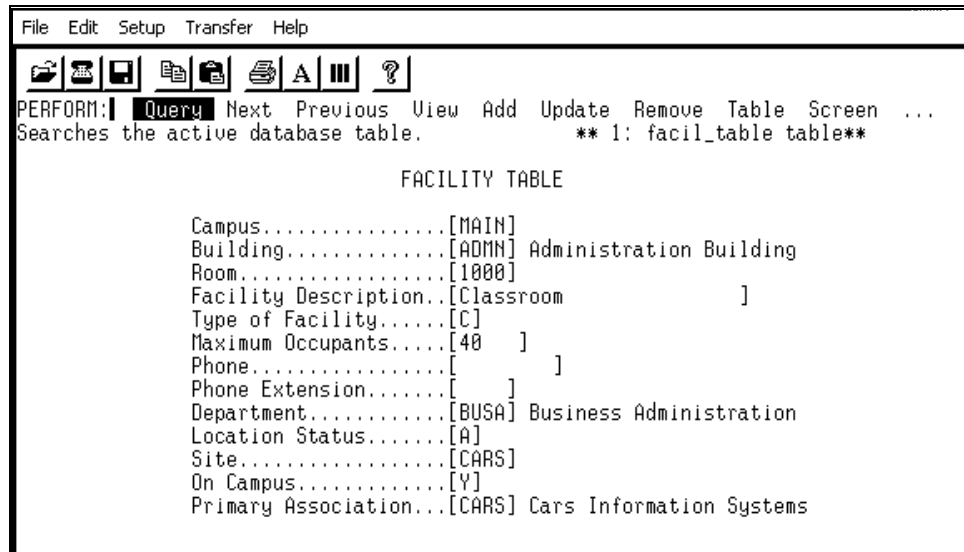
How to Access

The screen file for the Facility table is located in the following directory path:
\$CARSPATH/modules/common/screens/tfacil

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Facility Table screen may vary from your table format and content due to your institution's particular specifications.



Field Descriptions

The following describes the fields contained on the Facility Table screen.

Building

The building code. It identifies the building on the campus where the facility is located.

Example: REED (for the Reed Hall of Science)

Campus

The campus code. It identifies the campus where the facility is located.

Example: MAIN (for the main campus)

Department

The department owner. It identifies the department which has control of the use of this facility.

Example: CHE (for the Chemistry Department)

Facility Description

The description of the facility.

Example: Lecture/Lab room

Location Status

The status of this facility. Use A for active or I for inactive. Default value is A.

Maximum Occupants

The maximum occupancy of the facility.

On Campus

Does this facility reside on campus for billing purposes? Use Y for yes or N for no. Default value is Y.

Phone

The phone number. The telephone number of the facility.

Phone Extension

The extension number for the phone number of the facility.

Primary Association

The Primary Association code of the facility being evaluated.

Room

The room code or number. It identifies the room in the building on the campus where the facility is located.

Example: 102 (for room 102)

Site

The institution linked to location of the facility.

Example: CARS (for CARS College)

Type of Facility

The facility code. It identifies the facility usage type.

Example: C (for classroom); D (for dormitory room); O (for office)

Report Example

The following Facility Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992
13:12

CARS College
FACILITY TABLE REPORT
Site: CARS

Page 3
tfacil

Camp	Bldg	Room	Description	Occup	Telephone Extension	Type Status	Site Bill
MAIN	REED	100	Physics lab	24		C	CARS
	MPC					A	Y
MAIN	REED	101	Physics Lab	24		C	CARS
	MPC					A	Y
MAIN	REED	102	Lecture/Lab room	30		C	CARS
	CHE					A	Y
MAIN	REED	103	Lecture Hall	25		C	CARS
	MPC					A	Y
MAIN	REED	104	Lecture/Lab room	24		C	CARS
	MPC					A	Y
MAIN	REED	105	Lecture Hall	50		C	CARS
	MPC					A	Y

Location: /disk06/carsdevi/modules/common/reports/tfacil
Revision: G.101.110.2 04/10/92 12:34:07

Form Order Table

Purpose

The Form Order table (formord_table) defines the forms that can be ordered for printing using the Form Order program.

How to Access

The screen file for the Form Order table is located in the following directory path:
\$CARSPATH/modules/common/screens/tformord

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Form Order Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query Next Previous View Add Update Remove Table Screen ...
Searches the active database table.                ** 1: formord_table table**

                                FORM ORDER ENTRY

Order Code.....[          ] Description....[          ]
Printer.....[          ] Trans. Type...[ ]
Form Type.....[          ] Trans. Charge..[ ]
Operator Only..[ ] Tickler Code...[ ]
Alternate ID...[ ] Resource Code..[ ]
ACE Report.....[ ] Active Date...[ ]
Hold Action....[ ] Inactive Date..[ ]
Site.....[          ]

Program Parameters:
[          ]
```

Field Descriptions

The following describes the fields contained on the Form Order Table screen.

ACE Report

Is this form being generated by an ACE report? Use Y for yes or N for no.

Active Date

The date (mm/dd/yyyy) on which this table entry becomes valid.

Alternate ID

Is this form capable of being sent to an alternate ID? Use Y for yes or N for no.

Description

The text describing the form order code.

Example: Grades for Spring 1984 (for the form order code GRDSP84)

Form Type

The form type code.

Example: Trans (for transcript)

Hold Action

The hold action associated with this form.

Inactive Date

The date (mm/dd/yyyy) on which this table entry becomes invalid.

Order Code

The form order code (e.g., ARCTEST for ARC report test output).

Operator Only

Is this form to be restricted to operators only? Use Y for yes or N for no.

Printer

The printer name.

Example: p300

Program Parameters

The program and its parameters.

Example: trans -S

Resource Code

The resource code to use in batch mode

Note: Leave blank for immediate mode

Site

The site code for this form.

Example: CARS (for CARS College)

Note: The operator's login site (or the value of the global CARSSITE variable if you have not specifically set sites for logins) must be the same as the site value in order for the operator to execute the process specified in the table entry. This prevents an operator at site "A" from running a process that is set up for and specific to site "B".

Tickler Code

The tickler code to use in batch mode

Note: Leave blank for immediate mode.

Trans. Charge

Should transcript charging be enabled for this form? Use Y for yes or N for no.

Trans. Type

The transcript order form to track. Valid values are:

- O (Official)
- U (Unofficial)
- N (No tracking)

Report Example

The following Form Order Table report may vary from your report format and content due to your institution's particular specifications.

Fri Jan 26 1996
13:12

CARS College
FORM ORDER TABLE REPORT

Page 1
tformord

FORM ORDER ENTRY

Formord Code:	ARCTEST	Description:	ARC report test output.
Printer:	lascli	Formtype:	wide
Begin Date:	01/01/94	Ending Date:	01/01/98
Operator Only:	Y	Alternate ID:	N
Site:	CARS	Hold Action:	
Transcript:		Charging:	
Tickler Code:		Resource Code:	

Program Parameters:

'\$SCPPATH/common/runreports.scp -f "dsktest" \$ARCPATH/common/tformord lascli'

FORM ORDER ENTRY

Formord Code:	ARCTEST2	Description:	ARC report test output.
Printer:	lascli	Formtype:	wide
Begin Date:	01/01/94	Ending Date:	01/01/98
Operator Only:	Y	Alternate ID:	N
Site:	CARS	Hold Action:	
Transcript:		Charging:	
Tickler Code:		Resource Code:	

Program Parameters:

'sacego -q -d devi /disk09/cisids/dkim/dummy'

Location: /disk06/carsdevi/modules/common/reports/tformord
Revision: G.0.110.2 01/11/94 03:46:58

Handicap Table

Purpose

The Handicap table (hand_table) contains the types of handicaps that are sufficiently important for an institution to track.

Note: The table should contain all the handicaps that your institution plans to use. The extent of use of this table is normally dependent on the magnitude of the handicap student capability of your institution. Many institutions have a special office set up to meet the needs of handicapped students. If you have such an office, expect to add many different handicap codes.

One blank entry in the table is required for the category of no handicap.

How to Access

The screen file for the Handicap table is located in the following directory path:
\$CARSPATH/modules/common/screens/thand

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Handicap Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: hand_table table**

                                HANDICAP TABLE

Code.....[ ]
Description...[ ]
```

Field Descriptions

The following describes the fields contained on the Handicap Table screen.

Code

The handicap code. It identifies a specific handicap.

Example: BL (for blind)

Description

A description of the handicap code.

Example: Blind (for the handicap code BL)

Report Example

The following Handicap Table report may vary from your report format and content due to your institution's particular specifications.

```
Thu Jul 9 1992          CARS College          Page 1
14:55                  HANDICAP TABLE REPORT        thand
```

Code	Text
-----	-----
	No Handicap
BL	Blind

```
Location: /disk15/carsdevi/modules/common/reports/thand
Revision: G.101.110.1 03/04/92 10:25:47
```


Hold Tables

Purpose

The Hold table (hold_table) is used to restrict some functions from the student or other individual. These functions might be student accounts, registrar, etc.

The Hold Action table (hold_act_table) works in conjunction with the Hold table. As a result of the Hold Table entry, the Hold Action table specifies if the hold is absolute or a warning hold.

The Office Permission table (ofcperm_table) is used to grant permission to an office and individual for updating holds. Typically, one or more table entries will be added for each office allowed to have holds. The business office, registrar, admissions, and student services normally will have holds of various types.

How to Access

The screen file for the Hold tables is located in the following directory path:
\$CARSPATH/modules/common/screens/thold

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Creation Sequence

Considerations for the sequence of creating the tables in relation to other tables are:

1. Create the Hold table before the Hold Action and Office Permission tables.
2. Create the Hold Action table for the hold in the Hold table.
3. Create the Office Permission table for the specified hold.

You must create the Hold table before the Hold Action table. Associated with each hold code in the Hold table will be one or more hold action code entries that specify what is to be held as a result of the hold code. Multiple hold action code table entries are permissible for each Hold table entry.

Hold and Hold Action Relationships

An example of a hold and hold action relationship would be the following. A Hold table has a code for student accounts and the Hold Action table has an entry for transcript with an Absolute entry of yes. If a student with such a hold is delinquent in paying his student fees, an absolute hold is placed so as not to release the transcript until the hold is cleared. It is also common to place a registration absolute Hold Action table entry to student accounts to prevent registration. It is less common to place such a hold on grade reports.

Absolute Holds

Use absolute holds with some discretion since they are to prevent some action and can create additional work if used carelessly.

Office Permissions

A user can add, update, and delete holds only if he has database permissions set in the Office Permissions table (ofcperm_table). (See *Setting Up Office Permissions Checking* in Section 2 of the *CX Implementation and Maintenance Technical Manual*.) Programs check the ofcperm_table for a user's office permissions before allowing the adding or updating of holds. To delete a hold, the user must have ownership of the hold.

The system determines ownership of a hold based on whether or not an appropriate entry exists in:

- The ofcperm_table for the user or group

- The ofc_add_by column of the hold_rec

The system also takes active/inactive dates into consideration when determining the ownership of a hold.

Screen Example

The following Hold tables screen may vary from your table format and content due to your institution's particular specifications.

```

PERFORM:  Query Next Previous View Add Update Remove Table Screen ...
Searches the active database table.                ** 1: hold_table table**

          HOLD TABLE

          Code.....[A/P ]
          Description.....[Accounts Payable Hold  ]
          Phrase Number.....[1700 ]
          Comment (w/ HTML markups).....

[
=====
          HOLD ACTION TABLE

          Action..... PURCH
          Absolute..... Y
=====
          OFFICE PERMISSIONS TABLE

          User Office Code.....
          Group or User Number..
          Type.....

```

Field Descriptions

The following describes the fields contained on the Hold Tables screen.

Absolute

This indicates whether the affect of this hold is absolute or is just a warning. Use Y for yes or N for no. Default value is N.

Action

This code identifies the process affected by the associated hold.

Example: GRDRPT (for grade reporting)

Code

The hold code identifies the type of hold.

Example: ADMS (for Admissions)

Comment (w/ HTML Markups)

A blob field containing the explanation of the hold and the steps needed by the student to clear it. Used in web applications to provide more information to the user.

Description

The text describing the hold code.

Example: Admissions (for the hold code ADMS)

Group or User Number

The UNIX group (gid) or user (uid) identification number that has permission to update the associated hold.

Type

The permission type. Valid values are:

- G (Group ID#)
- U (User ID#)

User Office Code

The code indicating which office has permission to update the associated hold.

Example: ACAD (for Academic Affairs)

User uid Number

The login ID. This is the specific ID number of an individual, within the specified office, who is allowed to update the hold.

Example: 151 (for the user within the specified office who has the login ID of 151)

Report Example

The following Hold Table report may vary from your report format and content due to your institution's particular specifications.

```
Wed Dec 18 1996          CARS College          Page 1
12:23                   HOLD TABLE REPORT      thold

Code  Text
-----
A/P   Accounts Payable Hold
A/PA  Accts Pay Advisory Hold
ACAD  Academic Dean Hold
ADMS  Admissions Hold
ADVI  Advisor Hold
BUSO  Business Office
DSOL  Donor Solicitation Hold
EDTR  Education Transc Incompl
FEES  Registration Fees Unpaid
FINA  Financial Aid Hold
FINP  Financial Purge - No
INTR  Incomplete transcripts
MAIL  Mailing Hold
NOND  Non-Declared Major Hold
PUB   Publicity Hold
REGI  Registrar Hold
REGR  Registrar Holds
STUA  Student Accounts Hold
STUS  Student Services Hold

Location: /basel/carsdevi/modules/common/reports/thold
Revision: G.101 02/14/92 17:22:08
```

Report Example (Hold Action Table)

The following Hold Action Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992 14:56	CARS College HOLD ACTION TABLE REPORT		Page 1 tholdact
Code	Hold	Absolute	
REGIST	ACAD	NO	
REGIST	ADMI	NO	
REGR	ADMS	YES	
TRANS	ADVI	NO	
REGIST	ADVI	NO	
TRANS	BUSO	YES	
SOLICIT	DSOL	YES	
REGIST	FINA	NO	
JN	MAIL	YES	
DPJN	MAIL	YES	
NOJN	MAIL	YES	
REGIST	MAIL	NO	
CKSLCT	MAIL	YES	
SINGLEI	MAIL	YES	
PURCH	MAIL	YES	
NPREMIUM	PREM	YES	
NPREMIUM	PRM	NO	
GRDRPT	REG	NO	
TRANS	REGI	NO	
REGIST	REGI	NO	
JOIN	SOL	YES	
SINGLE	SOL	YES	
TRANS	STUA	NO	
REGIST	STUA	NO	
TRANS	STUS	NO	
REGIST	STUS	NO	

Location: /sysdisk/carsgSQL/modules/common/reports/tholdact
Revision: G.100 08/21/91 19:33:07

Report Example (Office Permissions Table Report)

The following Office Permissions Table report may vary from your report format and content due to your institution's particular specifications.

Wed Dec 18 1996 12:31		CARS College OFFICE PERMISSIONS REPORT		Page 1 tofcpem	
Code	Office Description	User / Group Type	ID		
ACAD	Academic Affairs	U	150		
ACAD	Academic Affairs	U	246		
ACAD	Academic Affairs	U	151		
ADM	Admissions Office	G	100		
ADMS	Admissions	U	147		
ADMS	Admissions	U	143		
ADMS	Admissions	U	185		
BASK	Basketball	G	100		
BOFF	Business Office	U	121		
BUSA		U	122		
DEVL	Development	U	122		
DEVL	Development	U	271		
FAID	Financial Aid	U	271		
MATR		U	228		
REGI		U	185		
REGR	Registrar's	U	153		
REGR	Registrar's	U	147		
REGR	Registrar's	U	116		
SOC		U	122		

* - Record is inactive according to active/inactive dates
Location: /usr/carsdevi/modules/common/reports/tofcpem
Revision: H.1 08/23/96 13:35:53

ID Office Permissions Table

Purpose

The ID Office Permissions table (idperm_table) defines the permissions to ID records (id_rec) for users and user groups for each office at your institution.

How to Access

The screen file for the ID Office Permissions table is located in the following directory path:
\$CARSPATH/modules/common/screens/idperm

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following ID Office Permissions Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query Next Previous View Add Update Remove Table Screen ...
Searches the active database table.                ** 1: idperm_table table**

                                ID PERMISSIONS TABLE
                                User Office Code.....[    ]
                                Group or User Number..[0    ]
                                Type.....[    ]
```

Field Descriptions

The following describes the fields contained on the ID Office Permissions Table screen.

Group or User Number

The UNIX group or user identification number (100:999).

Type

The permission type for the user. Valid values are:

- G (Group ID#)
- U (User ID#)

User Office Code

The office code (e.g., ADM for Admissions office).

Report Example

The following ID Office Permissions Table report may vary from your report format and content due to your institution's particular specifications.

Fri Jan 26 1996
13:35

CARS College
ID PERMISSIONS REPORT

Page 1
tidperm

Code	Office Description	User / Group Type	ID	Active / Inactive Date	Date
		U	159		
		U	157		
		U	146		
ADM		U	185	01/01/94	
ADM		G	120		
ADM		G	100		
ADMS	Admissions	U	146		
BOFF	Business Office	U	233		
DEVL	Development	U	146		
DEVL	Development	U	166		
DEVL	Development	U	122		
KEVI		G	998	05/18/93	12/31/99
REG		U	166		
REG		U	157		
REGR	Registrar's	G	999	05/18/93	12/31/99
REGR	Registrar's	U	146		
REGR	Registrar's	U	166		

* - Record is inactive according to active/inactive dates
Location: /disk06/carsdevi/modules/common/reports/tidperm
Revision: G.0 06/09/93 13:55:49

Interest Table

Purpose

The Interest table (int_table) contains those activities (academic, social, athletic) that may be of interest to a prospective student, but do not fit into the categories of accomplishment or involvement.

Note: Add interest codes as necessary to accommodate the various interests of students. Do not confuse these with accomplishments or involvements, which are in separate tables.

These entries are frequently of much interest to the admissions office because they may be used for recruitment purposes. For example, someone has an interest in the Honors Program which may be the key to send literature on that program. These interests usually include a wide range of academic, social (extra-curricular), and athletic areas.

How to Access

The screen file for the Interest table is located in the following directory path:
\$CARSPATH/modules/common/screens/tint

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Interest Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  Current  Master
Detail ...
Searches the active database table.          ** 1: int_table table**

                INTEREST TABLE

Code.....[   ]
Description...[           ]
Type.....[   ]
Display on Web.[ ]
```

Field Descriptions

The following describes the fields contained on the Interest Table screen.

Code

The interest code. A code given to an area of interest by which that interest may be referred.

Example: ACCO (for accounting)

Description

The text that describes the interest code.

Example: Accounting (for the code ACCO)

Type

The interest type code. A code identifying the type of interest in general terms.

Example: ACAD, ATHL, MUSI, etc.

Display on Web

A Y/N field indicating whether or not you want this field to display on the Web Admissions Application.

Report Example

The following Interest Table report may vary from your report format and content due to your institution's particular specifications.

Mon Jan 5 1998 14:24	CARS College INTEREST TABLE REPORT		Page 1 tint
Code	Text	Type	W
----	-----	----	-
ACCO	Accounting	ACAD	Y
ADVE	Advertising	ACAD	Y
AERO	Aeronautics	ACAD	Y
AESP	Aerospace Studies	ACAD	Y
AGRO	Agronomy	ACAD	Y
AMER	American Studies	ACAD	Y
ANTH	Anthropology	ACAD	Y
ARCH	Archery	ACAD	Y
ART	Art - Painting	ACAD	Y
ARTE	Art Education	ACAD	Y
AUTO	Automotive Technology	ACAD	Y
BASE	Baseball	ATHL	Y
BASK	Basketball	ATHL	Y
BIBL	Biblical Literature	ACAD	Y
BIOL	Biology	ACAD	Y
BLAC	Black World Studies	ACAD	Y
BOTA	Botany	ACAD	Y
BROA	Broadcasting	ACAD	Y
BUSA	Business Administration	ACAD	Y
BUSE	Business Education	ACAD	Y
BUSL	Business Law	ACAD	Y
CART	Cartography	ACAD	Y
CHED	Christian Education	ACAD	Y
CHEM	Chemistry	ACAD	Y
Standard input			
* - Record is inactive according to active/inactive dates			
Location: /usr/carsbeta/modules/common/reports/tint			
Revision: Released 12/12/97 18:57:15			

Involvement Table

Purpose

The Involvement table (involve_table) contains those activities that a student might participate in. Participation in varsity basketball, debate club, etc. are representative examples. Do not confuse involvements with interests or accomplishments.

Note: This table contains the various types of activities a student is or has been involved in. Some institutions track their athletes through this table and the associated accomplishment record because they can specify periods of involvement. Also, it is a good way of tracking occurrences of participation in such activities as offices held and leadership positions.

How to Access

The screen file for the Involvement table is located in the following directory path:
\$CARSPATH/modules/common/screens/tinvolve

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Involvement Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  Current  Master
Detail ...
Searches the active database table.          ** 1: invl_table table**

                INVOLVEMENT TABLE

Code.....[      ]
Description.....[                               ]
Display on Web...[ ]
IPEDS Sports Lev.[ ]
```

Field Descriptions

The following contains descriptions of the fields contained on the Involvement Table screen.

Code

The involvement code. A code given to an activity by which a person's participation in that activity may be referred.

Example: BASE (for the baseball team)

Description

The text that describes the activity referred to in the involvement code.

Example: Baseball Team (for the involvement code BASE)

Display on Web

A Y/N field dictating whether or not you want this field to display on the Web Admissions Application.

IPEDS Sports Lev.

Specifies the level of a sport which is eligible for athletic aid as specified by the Department of Education Valid codes are as follows:

- 1 - Football
- 2 - Basketball
- 3 - Baseball
- 4 - Cross - country/track
- 5 - All other sports

Leave this field blank if 1- 5 does not apply.

Report Example

The following Involvement Table report may vary from your report format and content due to your institution's particular specifications.

```

Mon Jan 5 1998          CARS College          Page 1
13:52                 INVOLVEMENT TABLE REPORT        tinvolve

```

Code	Text	Sports W Level	Active /Inactive Date Date
ATHL	Athletic	Y	
BASE	Baseball Team	Y	
DEB	Debate Society	Y	
DELT	Tri-Delta Sorority	Y	01/01/90
FHOC	Field Hockey	Y	
FOOT	Football Team	Y	
FRIE	Friend of College	Y	
LIFP	Lifetime Presidents Club Giving	Y	
PRES	Presidents Club	Y	
RG	Rockwood Giddings Member	Y	
RGDF	R Giddings Dist Giving Club	Y	
RGF	Rockwood Giddings Fellow Giving	Y	
RGM	Rockwood Giddings Giving Club	Y	
SWIM	Swim Team - Men's	Y	
SWIW	Swim Team - Women's	Y	
TENM	Tennis Team - Men's	Y	
TENW	Tennis Team - Women's	Y	
YAC	Young Alumni Club	Y	

* - Record is inactive according to active/inactive dates
Location: /usr/carsbetai/modules/common/reports/tinvolve
Revision: Released 12/19/97 13:32:41

Marital Table

Purpose

The Marital table (mrtl_table) contains the valid marital statuses that are used at your institution.

How to Access

The screen file for the Marital table is located in the following directory path:
\$CARSPATH/modules/common/screens/tmrtl

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Marital Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                         ** 1:mrtl_table table**

                               Marital TABLE
Code.....[      ]
Description...[      ]
```

Field Descriptions

The following describes the fields contained on the Marital Table screen.

Active Date

The date on which the table entry becomes valid. CX displays this field on reports and screens if the macro ENABLE_FEAT_BEG_END_DATE is set to Y.

Code

The marital status code. A code given to a marital status by which that status may be referred.

Example: M (for Married)

Description

The text used to describe the marital status referred to by the marital status code.

Example: Married (for the code M)

EDI Marital Status Code

The marital status code used on the data obtained via EDI (Electronic Data Interchange). CX translates this code on records to the Code you assign. This field appears on your table screen only if the macro ENABLE_TRANS_EDI is set to Y.

Inactive Date

The date on which the table entry becomes invalid. CX displays this field on reports and screens if the macro ENABLE_FEAT_BEG_END_DATE is set to Y.

Report Example

The following Marital Table report may vary from your report format and content due to your institution's particular specifications.

```
Thu Jul 9 1992          CARS College          Page 1
15:41                  MARITAL TABLE REPORT      tmrtl

      Code   Text                               Active /Inactive
      ----   -
      D      Divorced
      M      Married
      P      single Parent
      S      Single
      T      Separated
      W      Widowed

* - Record is inactive according to active/inactive dates
Location: /usr/carsdevi/modules/common/reports/tmrtl
Revision: Developmental 08/29/97 15:08:18
```

Occupation Table

Purpose

The Occupation table (occ_table) consists of generalized jobs (occupations) that may be used for your institution's purposes.

Note: This table contains many occupation types. Dependent upon your use of the table, it may require some modification to the level of detail desired. A source of occupational titles is the *Directory of Occupational Titles*, published by the federal government. The advantage of using such a document is that it will provide consistency to the level desired for positions.

This occupational information is valuable for recruiting purposes and identifying individuals by type of work when it is input on the employment record.

How to Access

The screen file for the Occupation table is located in the following directory path:
\$CARSPATH/modules/common/screens/tocc

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Occupation Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: occ_table table**

                OCCUPATION TABLE

Code.....[      ]
Description....[      ]
```

Field Descriptions

The following describes the fields contained on the Occupation Table screen.

Code

The occupation code. A code given to an occupation by which that occupation may be referred.

Example: ADV (for advertising)

Description

The text used to describe the occupation referred to by the occupation code.

Example: Advertising (for the occupation code ADV)

Report Example

The following Occupation Table report may vary from your report format and content due to your institution's particular specifications.

```
Thu Jul 9 1992          CARS College          Page 1
15:02                OCCUPATION TABLE REPORT          tocc
```

Code	Text
ADV	Advertising
AGR	Agriculture;Ranching
AIR	Aircraft Industry
ARC	Architecture;Landscaping
ART	Art
ATH	Athletics
AUS	Armed Forces
AUT	Automobile Business
AUTH	Author
AVI	Aviation
BEV	Beverage Manu/Distr
BK	Banking
BLD	Construction;Carpentry
BTCR	Butcher
BUS	Business;Comm Enterprise
CARS	CARS Solution Systems Rep
CHM	Chemists;Chemistry
CARS	CARS Account Manager
CLP	Clinical Psychology
COL	Coal&Fuel;Explor/Develop
COM	Computers;Manuf/Sales
DDS	Dentistry
DIET	Dietitian
DNC	Dance: Teach,Perform etc
EAM	Education Administration

```
Location: /base1/carsdevi/modules/common/reports/tocc
Revision: G.101 02/14/92 17:22:49
```

Office Table

Purpose

The Office table (ofc_table) should contain all the offices on a campus. This information is used in a variety of ways. One of the more important ways is to specify which office added a record. Examples of such records are: id_rec, hold_rec, aa_rec, adree_rec, etc.

Note: You should update this table to include all offices that will be using the system directly or indirectly. Typically, all offices on a campus should be added.

How to Access

The screen file for the Office table is located in the following directory path:
\$CARSPATH/modules/common/screens/tofc

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Office Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                         ** 1: ofc_table table**

                                OFFICE TABLE

Code.....[      ]
Description....[      ]
```

Field Descriptions

The following describes the fields contained on the Office Table screen.

Code

The office code. A code given to an office for reference purposes.

Example: ACAD (for the Academic Affairs office)

Description

The office code description. The text describing the office code.

Example: Academic Affairs (for the office code ACAD)

Report Example

The following Office Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992
15:03

CARS College
OFFICE TABLE REPORT

Page 1
tofc

Code	Text
ACAD	Academic Affairs
ADMS	Admissions
ADMT	Administration & Treas.
BASK	Basketball
BOFF	Business Office
BOOK	Bookstore
CAMP	Campus Activities
CARR	Career Planning
CHAN	Chancellor
COMP	Computer Services
DEVL	Development
FAID	Financial Aid
FOOD	Food Services
FOOT	Football
GRAD	Graduate Studies
HOUS	Housekeeping
INFR	Infirmery
LIBR	Library
MAIN	Maintenance
MINS	Campus Ministry
NOTE	Notes Payable Office
PERS	Personnel/Payroll
POST	Post Office
PRES	President
PRNT	Printing

Location: /basel/carsdevi/modules/common/reports/tofc
Revision: G.101 02/14/92 17:22:57

Permission Table

Purpose

The Permission table (perm_table) contains codes for each level of access on the CX database.

How to Access

The screen file for the Permission table is located in the following directory path:
\$CARSPATH/modules/common/screens/tperm

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Permission Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: perm_table table**

                PERMISSION TABLE

Program or Process...[          ]
Category .....[          ]
Permission Code .....[          ]
Group or User Number..[          ]
Type .....[          ]
Exclude Permission...[          ]
```

Field Descriptions

The following describes the fields contained on the Permission Table screen.

Category

The permission category. It identifies a restriction category for this permission.

Example: GLPERM (for general ledger permission)

Exclude Permission

Is permission being denied to the user/group for this permission? Use Y to deny permission or N to grant permission. Default value is N.

Group or User Number

The system user or group ID number associated with this permission.

Example: 121

Permission Code

The permission code that is granted or denied access.

Example: ALL or CASHIER

Program or Process

The name of the application program using this record. If left blank, this record applies to all programs using this table.

Example: acquery

Type

Specifies whether the system ID number is for a user or a group. Use U for user ID number or G for group ID number.

Report Example

The following Permission Table report may vary from your report format and content due to your institution's particular specifications.

Program Code	Category	Code	Number	User/Group	Exclude
	CATPERM	KELLY	121	U	N
	CATPERM	*****	121	U	N
	CATPERM	KELLY	147	U	Y
	CATPERM	*****	147	U	N
	CATPERM	KELLY	149	U	N
	CATPERM	COMMON	149	U	N
	CATPERM	KELLY	151	U	N
	GLPERM	ALL	140	G	N
	GLPERM	ALL	102	U	N
	GLPERM	ALL	103	U	N
	GLPERM	ALL	105	U	N
	GLPERM	ALL	107	U	N
	GLPERM	ALL	110	U	N
acquery	GLPERM	ALL	121	U	N
	GLPERM	CASHIER	121	U	N

Location: /sysdisk/carsgSQL/modules/common/reports/tperm
Revision: G.100 08/21/91 19:33:24

Privacy Act Tables

Purpose

The Privacy table (priv_table) contains a code and text description of the privacy style that you mark as "private" on screens in entry programs. Since the names of these styles (e.g., ADDR for Address information) is arbitrary, you can define them any way you prefer.

The Privacy Field table (privfd_table) contains the database records and fields that are located in the groups from the Privacy table. Each group can contain as many records and fields as you want, but only records and fields that are accessible in entry programs are highlighted.

How to Access

The screen file for the Privacy Act Table is located in the following directory path:
\$CARSPATH/modules/common/screens/tpriv

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Creation Sequence

Create these tables in the following sequence: enter the privacy code in the Privacy table, then enter the corresponding table name(s) and column(s) in the Privacy Field table.

Screen Example

The following Privacy Act Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM: Query Next Previous View Add Update Remove Table Screen ...
Searches the active database table.          ** 1: priv_table table**
                PRIVACY ACT INFORMATION

    Privacy Code.....[      ]
    Description.....[                ]
    Comment Text.....

[                                                                    ]
-----
    Table Name.....
    Column Value.....
```

Field Descriptions

The following contains descriptions of the fields contained on the Privacy Act screen.

Column Value

The column name for the privacy code (e.g., addr_line1)

Comment Text

A blob field containing the explanation of the privacy restriction. It is used in web applications to provide the user more information about the restriction.

Description

The description of the privacy code.

Privacy Code

The privacy code (e.g., ADDR for name and address information).

Table Name

The table name applicable to the privacy code (e.g., id_rec for ID record).

Report Example

The following Privacy Act Table report may vary from your report format and content due to your institution's particular specifications.

```
Fri Jan 26 1996          CARS College          Page  1
13:50                   ENTRY PROGRAM PRIVACY ACT INFORMATION      tpriv
```

Privacy Code	Database Table(s) and Column(s)
ADDR Name and Address Info	id_rec.addr_line1 id_rec.addr_line2 id_rec.city id_rec.fullname id_rec.st id_rec.zip profile_rec.priv_code
BIRTH Birthday and sex	profile_rec.birth_date profile_rec.sex
PHON Name, Address and Phone	id_rec.addr_line1 id_rec.addr_line2 id_rec.city id_rec.fullname id_rec.phone id_rec.phone_ext id_rec.st id_rec.zip profile_rec.priv_code

Location: /disk06/carsdevi/modules/common/reports/tpriv
Revision: G.0 04/22/94 10:23:39
:39

Relationship Table

Purpose

The Relationship table (rel_table) contains the valid primary/secondary relationships.

Note: The base system contains many of the basic relationships that are necessary to implement the system. The real issue is whether or not your institution plans to use relationship records. If so, which offices will be responsible for maintaining them. The two primary users will probably be admissions and development.

Do not reverse a relationship order in another code.

How to Access

The screen file for the Relationship table is located in the following directory path:
\$CARSPATH/modules/common/screens/trel

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Relationship Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                         ** 1: rel_table table**

                                RELATIONSHIP TABLE

Code.....[      ]
Description....[      ]
Maintenance....[ ]
```

Field Descriptions

The following describes the fields contained on the Relationship Table screen.

Code

The relationship code. A code assigned to a relationship.

Example: BS (for brother/sister)

Description

The text describing the relationship code.

Example: Brother/Sister (for the relationship code BS)

Maintenance

Should related IDs have their addresses maintained? Use Y for yes and N for no. The default value is no.

Report Example

The following Relationship Table report may vary from your report format and content due to your institution's particular specifications.

Fri Jan 26 1996 14:01		CARS College RELATIONSHIP TABLE REPORT		Page 1 trel	
Code	Text	Maint	Active Date	Inactive Date	
AUE	Aunt / Nephew	N	01/01/94		
AUI	Aunt / Niece	N			
BB	Brother/Brother	N			
BE	Business/Employee	N			
BI	Brother / In-Law	N			
BS	Brother/Sister	N			
CE	Company / Employee	N			
COE	College / Employee	N			
CP	Church/Pastor	N			
FD	Father/Daughter	N			
FI	Father / In-Law	N			
FS	Father/Son	N			
GW	Guardian/Ward	N			
HW	Husband/Wife	Y			
MD	Mother/Daughter	N			
MEM	Member	N			
MI	Mother / In-Law	N			
MS	Mother/Son	N			
PC	Parent/Child	Y			
SC	School/Counselor	N			
SI	Sister / In-Law	N			
SP	School/Principal	N			
UNE	Uncle/Nephew	N			
UNI	Uncle/Niece	N			

* - Record is inactive according to active/inactive dates
 Location: /disk06/carsdevi/modules/common/reports/trel
 Revision: G.101.110.2 04/22/94 10:06:30

State Table

Purpose

The State table (st_table) contains all states and US territories.

Note: This table will probably not require updating from the base product. The zip codes specify the ranges for a given state and can be used as desired by your institution.

How to Access

The screen file for the State table is located in the following directory path:
\$CARSPATH/modules/common/screens/ts

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following State Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: st_table table**

                STATE TABLE

Code.....[ ]
Text.....[ ]
Description...[ ]
Low Zip.....[ ]
High Zip.....[ ]
```

Field Descriptions

The following describes the fields contained on the State Table screen.

Code

The state code. A code assigned to a state for reference purposes.

Example: SC (for South Carolina)

Description

The Federal Information Processing Standard (FIPS) code assigned to this state

Example: SC (for South Carolina)

High Zip

The high end of the zip code range for this state.

Low Zip

The low end of the zip code range for this state.

Text

The text describing the state code.

Example: South Carolina (for the state code SC)

Report Example

The following State Table report may vary from your report format and content due to your institution's particular specifications.

Code	Text	Low Zone	High Zone	Description
PR	Puerto Rico	00600	00999	PR
RI	Rhode Island			RI
SC	South Carolina	29000	29999	SC
SD	South Dakota	57000	57799	SD
TN	Tennessee	37000	38599	TN
TX	Texas	75000	79999	TX
UT	Utah	84000	84799	UT
VA	Virginia	22000	24699	VA
VI	Virgin Islands			VI
VT	Vermont	05000	05999	VT
WA	Washington	98000	99499	WA
WI	Wisconsin	53000	54999	WI
WV	West Virginia	24600	26899	WV
WY	Wyoming	82000	83199	WY
YK	Yukon Territories			YK

Location: /disk06/carsdevi/modules/common/reports/tst
Revision: G.201 07/07/92 15:41:37

Subscription Table

Purpose

The Subscription table (sbscr_table) is a variation of the Contact table that specifies recipients, other than students, who are to receive some communication (e.g. grade reports, student data sheets that contain billing information).

Note: This table is a variation of the Contact table and allows your institution to specify other recipients of selected information in addition to the student. Two common uses are for GRDRPT (grade reports) and SDSBILL (student data sheets with billing).

How to Access

The screen file for the Subscription table is located in the following directory path:
\$CARSPATH/modules/common/screens/tsbscr

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Subscription Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: ctc_table table**

                SUBSCRIPTION TABLE

When querying, enter "SBSC"..[      ]

Subscription Code...[          ]
Description.....[          ]
Run Code.....[          ]
Communication Code..[          ]
Ace Report.....[          ]
Routing.....[          ]
Span Waived.....[          ]
Reissued.....[          ]
Enrollment Status...[          ]
```

Field Descriptions

The following describes the fields contained on the Subscription Table screen.

Ace Report

The name of the ACE report using this subscription.

Example: ltracadrec (for academic records letter)

Communication Code

The communication code. This code identifies the type of communication this contact is.

Example: LTLB (for letter and label)

Description

The text describing the subscription code.

Example: DEANLIST (for the dean's list letters)

Enrollment Status

The enrollment status achieved with this contact. There is no entry to this field.

Example: ACCEPTED (for a subscription code of ACCLET acceptance letter)

Reissued

Can the contact occur more than once? (Y = yes; N = no) There is no entry to this field.

Routing

Is this an incoming or outgoing contact?
(I = incoming; O = outgoing) There is no entry to this field.

Run Code

The run code for this subscription.

Example: SINGLE

Span Waived

Is the normal span for days between contacts waived for this contact? (Y = yes; N = no)
There is no entry for this field.

Subscription Code

The code assigned to this subscription.

Example: DEANLIST (for the dean's list letters)

Report Example

The following Subscription Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992 16:10	CARS College SUBSCRIPTION TABLE REPORT		Page 1 tsbscr
Code	Text	Comm	Run Code
-----	-----	---	-----
ALUMNEWS	Alumni News Mailing	LABL	JN
PRESRPT	President's Annual Rpt	LETT	JN
Location: /sysdisk/carsgSQL/modules/common/reports/tsbscr Revision: G.100 08/21/91 19:33:31			

Suffix Table

Purpose

The Suffix table (suffix_table) contains codes for all valid suffixes (e.g., Esq for Esquire).

How to Access

The screen file for the Suffix table is located in the following directory path:
\$CARSPATH/modules/common/screens/tsuffix

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Suffix Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: suffix_table table**

                SUFFIX TABLE

Code.....[   ]
Title.....[   ]
Labels.....[           ]
Joint Labels...[           ]
```

Field Descriptions

The following describes the fields contained on the Suffix Table screen.

Code

The suffix code.

Example: ESQ (for esquire)

Joint Labels

The text used for printing when two IDs have the same suffix.

Example: Esquires

Labels

The text used for printing labels.

Example: Esq. (for esquire)

Title

Can this suffix be used with titles? Use Y for yes or N for no.

Report Example

The following Suffix Table report may vary from your report format and content due to your institution's particular specifications.

Code	Title	Text	Joint Text
ESQ	N	Esq	Esquires

Location: /disk06/carsdevi/modules/common/reports/tsuffix
Revision: G.50 07/07/92 15:41:41

Tickler Table

Purpose

The Tickler table (tick_table) specifies the ticklers used by your institution, even if they are not set up in the tickler structure.

Note: One entry in this table is required for the registrar even though a tickler process may not be established for the registrar's office. Few, if any, registrars establish specific tickler applications for their office, since the great majority of their processes do not lend themselves to such applications.

How to Access

The screen file for the Tickler table is located in the following directory path:
\$CARSPATH/modules/common/screens/ttick

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Tickler Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: tick_table table**

                                TICKLER TABLE

Code.....[      ]
Description.....[      ]

Minimum Contact Span..[      ]
Maximum Contact Span..[      ]
Maximum Review Span...[      ]
Mail Recipient.....[      ]
```

Field Descriptions

The following describes the fields contained on the Tickler Table screen.

Code

The tickler code. A code assigned to a tickler system for identification purposes.

Example: ADM (for admissions tickler)

Description

The text describing the tickler system referred to by the tickler code.

Example: Admissions Tickler (for the tickler code ADM)

Maximum Contact Span

The user's login name, who is to receive mail from the tickler program.

Example: admit (for admissions)

Maximum Contact Span

The maximum number of days which may exist between sending contacts to a person on this tickler system.

Example: 30 (for 30 days)

Maximum Review Span

The maximum number of days between reviews of a person on this tickler system.

Example: 30 (for 30 days)

Minimum Contact Span

The minimum number of days which may exist between sending contacts to a person on this tickler system.

Example: 10 (for 10 days)

Report Example

The following Tickler Table report may vary from your report format and content due to your institution's particular specifications.

Thu Jul 9 1992 16:12	CARS College TICKLER TABLE REPORT				Page 1 ttick
Tick	Description	Ctc Span	Rev	Mail	
ADM	Admissions Tickler	10/ 30	30	coord	
ADMG	Graduate Admissions	10/ 30	30	admit	
ALPR	Alumni Public Relations	7/ 30	7	mark	
DEV	Development	30/ 90	30		
FA87	Fall 87 Financial Aid	0/ 10	7	dave	
FA88	Fall 88 Financial Aid	0/ 10	7	dave	
FA89	FA89 Tickler	0/ 0	0		
FY87	Financial Aid 8788	0/ 30	7	dave	
FY88	Financial Aid 8889	0/ 30	7	dave	
FY89	8990 Tickler	0/ 30	7	dave	
FY91	9192 Financial Aid	0/ 10	7	gerry	
FY92	9192 Financial Aid	0/ 10	7	gerry	
FY94	9091 FA Document Track'g	0/ 30	7	dave	
HUNT	CARS Standard ADM tick	10/ 30	10	ken	
INV	Invitation to Open House	14/ 28	14		
KADM	StKate Admissions	5/ 30	30	harold	
MAIL	Direct Mailings	0/ 0	0		
REG	Registration	0/ 0	0		
SAFF	Std Services/Affairs	0/ 0	0		
SBSC	Subscription	0/ 0	0		
SP88	Spring 89 Financial Aid	0/ 10	7	dave	
SP89	Spring 89 Financial Aid	0/ 10	7	dave	
SP90	SP90 Tickler	0/ 0	0		

Location: /sysdisk/carsgSQL/modules/commgmt/reports/ttick
Revision: G.100 08/21/91 19:16:43

Title Table

Purpose

The Title table (title_table) contains the various titles that are used by your institution.

Note: The base product has a reasonably comprehensive set of titles. You should update the table as necessary for your institution.

You should use titles when building your ID records so that other applications will respond properly (e.g.,adr).

How to Access

The screen file for the Title table is located in the following directory path:
\$CARSPATH/modules/common/screens/ttitle

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Title Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  Current  Master
Detail ...
Searches the active database table.                ** 1: title_table table**

                TITLE TABLE

Code.....[      ]
Priority.....[      ]
Labels.....[      ]
Salutation.....[      ]
Joint.....[      ]
Gender.....[      ]
Display on Web.[      ]
```

Field Descriptions

The following describes the fields contained on the Title Table screen.

Code

The title code. A code assigned to a specific title.

Example: CAPT (for Captain)

Joint

Does this title address two individuals (e.g., Mr. & Mrs.) jointly? Use Y for yes or N for no. Default value is N.

Labels

The text used for printing on labels.

Example: Capt. (for Captain)

Priority

The title priority. The priority level assigned to a title in case the order of displaying two titles may make a difference. This is used for report sorting. The value of "0" (zero) is the highest.

Salutation

The formal usage of this title when used as a salutation in a letter.

Example: Capt. & Mrs. (for the title code CPMS)

Display on Web

A Y/N field dictating whether or not you want this field to display on the Web Admissions Application.

Report Example

The following Title Table report may vary from your report format and content due to your institution's particular specifications.

Code Text		Salutation	Prty	J	G	W
Mon Jan 5 1998 14:33		CARS College TITLE TABLE REPORT				Page 1 ttitle
*CAPT	Capt.	Captain	0	N	Y	
COL	Col.	Col.	0	N	Y	
COLM	Col. & Mrs.	Col. & Mrs.	0	Y	M	Y
CPMS	Capt. & Mrs.	Capt. & Mrs.	0	Y	M	Y
DEAN	Dean	Dean	0	N	Y	
DR	Dr.	Dr.	0	N	Y	
DRMS	Dr. & Mrs.	Dr. & Mrs.	0	Y	M	Y
FR	Fr.	Father	0	N	M	Y
GEN	Gen.	Gen.	0	N	N	
HG	The Hon.	Governor	15	N	Y	
HGMS	The Hon. & Mrs.	Governor & Mrs.	0	Y	M	Y
HJ	The Hon.	Judge	0	N	Y	
HJMS	The Hon. & Mrs.	Judge & Mrs.	0	Y	M	Y
HM	The Hon.	Mayor	0	N	Y	
HMMS	The Hon. & Mrs.	Mayor & Mrs.	0	Y	M	Y
LCDR	Lt. Cdr.	Lt. Cdr.	0	N	Y	
LT	Lt.	Lt.	0	N	Y	
LTC	Lt. Col.	Lt. Col.	0	N	Y	
MAJ	Major	Major	0	N	Y	
MISS	Miss	Miss	0	N	F	Y
MM	Mr. & Mrs.	Mr. & Mrs.	0	Y	M	Y
MR	Mr.	Mr.	0	N	M	Y
MRMS	Mr. and Mrs.	Mr. and Mrs.	0	Y	M	Y
MRS	Mrs.	Mrs.	0	N	F	Y
MS	Ms.	Ms.	0	N	F	Y

* - Record is inactive according to active/inactive dates
 Location: /usr/carsbetai/modules/common/reports/ttitle
 Revision: Released 12/12/97 18:57:20

User ID Table

Purpose

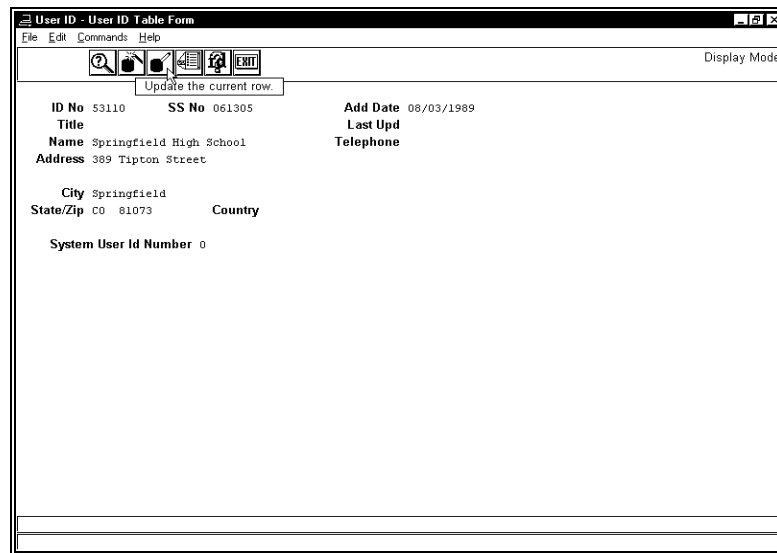
The User ID table (userid_table) contains user ID numbers and provides a link between the login ID number and the database id_no.

How to Access

The screen file for the User ID table is located in the following directory path:
\$CARSPATH/modules/common/progscr/identry/tuserid_1

Screen Example

The following User Id Table may vary from your table format and content due to your institution's particular specifications.



The screenshot shows a window titled "User ID - User ID Table Form" with a menu bar (File, Edit, Commands, Help) and a toolbar. The main area displays the following information:

ID No	53110	SS No	061305	Add Date	08/03/1989
Title		Last Upd		Telephone	
Name	Springfield High School				
Address	389 Tipton Street				
City	Springfield				
State/Zip	CO	81073	Country		
System User Id Number	0				

Field Descriptions

Add Date

The date the user was added

Address

The street address of the user.

City

The city in which the user lives.

Country

The country in which the user lives.

ID No

The user's ID number.

Last Upd

The date the ID record was last updated. (This field is display only.)

Name

The name of the user.

SS No

The social security number of the user.

State/Zip

The state in which the user lives, and the user's zip code.

System User ID Number

The user's system Id number.

Telephone

The telephone number of the user.

Title

The user's title.

Report Example

The following User ID Table report may vary from your report format and content due to your institution's particular applications.

```
Tue Jan 13 1998          CARS College          Page 1
09:40                  USER ID TABLE REPORT      tuserid
                          Sorted by fullname

  User      Active /Inactive
  ID        ID No   Name      Date      Date
-----
  999          0
  277    100000
  255    104575  ACCOUNTS PAYABLE CLERK
  207    104667  Bargo, Jerome D.
  247    104893  Smith, Steve
  191    104624  Jones, Robert F.
  212     22407  Loggia, Timothy
  224    104629  Davis, Paul
  131     20228  DeVry, Kenneth
  268    104781  Dorney, Doug
  130     20315  Severil, Philip
  151     28360  Roth, William
  225    104628  Birdsong, Montgomery
* 158     92766  Stuart, Andrea      01/01/93 01/01/98
  233    104651  Elbony, Tina
  246    104753  Lincoln, Thomas
  121     20240  Thackeray, Robert
  221    104625  Bryant, Toni R.
  159    104576  Kowalczyk, Edward
  199    104688  Simon, Arthur
  211    104623  McGraw, Stephen
  196    104855  Tyson, Marge
  217    104626  Simpson, Lance
  128     84987  Clark, Christopher

* - Record is inactive according to active/inactive dates
Location: /disk07/carsbetai/modules/common/others/tuserid
Revision: Released 08/20/92 18:06:47
```

Veteran Chapter Table

Purpose

The Veteran Chapter table (vetchap_table) contains codes for veteran chapters.

How to Access

The screen file for the Veteran Chapter table is located in the following directory path:
\$CARSPATH/modules/common/screens/tvetchap

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Veteran Chapter Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
                                                ** 1: vetchap_table table**

          VETERAN CHAPTER TABLE
Code.....[      ]
Description...[      ]
```

Field Descriptions

The following describes the fields contained on the Veteran Chapter Table screen.

Code

The code for this veteran chapter.

Description

The description for this veteran chapter code.

Report Example

The following Veteran Chapter Table report may vary from your report format and content due to your institution's particular specifications.

Fri Jan 26 1996
15:24

CARS College
VETERAN CHAPTER TABLE REPORT

Page 1
tvetchap

Code	Text	Active Date	/Inactive Date
AUTO	Automobile Veterans		

* - Record is inactive according to active/inactive dates
Location: /disk06/carsdevi/modules/common/reports/tvetchap
Revision: G.0.110.2 11/06/92 12:17:41

Visa Table

Purpose

The Visa table contains codes and descriptions of Visa codes recognized by the system. A visa is official authorization for travel within a specific country. An example of a visa code recognized by the CX System is F-1 for student in an academic or language program.

How to Access

The screen file for the Visa table is located in the following directory path:
\$CARSPATH/modules/common/screens/tvisa.

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Visa table screen example may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  Current  Master
Detail ...
Shows the next row in the Current List.          ** 1: visa_table table**

                VISA TABLE
Code.....[F-1 ]
Description...[Student in academic or language ]
Description...[program ]
Description...[ ]
```

Field Descriptions

The following are descriptions of the fields contained on the Visa Table screen.

Code

The visa code as given by Immigration and Naturalization Service. For example, "F-1" for Student in academic or language program.

Description

Text describing the associated visa code. There may be more than description line to accommodate lengthy descriptions. For example, "Student in academic or language program" for the code F-1.

Report Example

The following portion of a Visa Table report may vary from your report format and content due to your institution's particular specifications.

Mon Jan 5 1998 11:54		CARS College VISA TABLE REPORT		Page 1 tvisa	
Code	Text	Active / Inactive		Date	Date
-----	-----	-----	-----	-----	-----
	Blank				
1234	This is but a description of an undetermined length. So I guess that I can just keep going and				
A-1	Ambassador, public minister, career diplomat or consular, & members of immediate family				
A-2	Other foreign government official or employee, and members of immediate family				
A-3	Attendant, servant, or personal employee of alien classified A-1 or A-2, & imm. family members				
B-1	Temporary visitor for business				
B-2	Temporary visitor for pleasure				
C-1	Alien in transit				
C-2	Alien in transit to U.N. headqtr				

Zip Code Table

Purpose

The Zip Code table (zip_table) contains zip codes.

How to Access

The screen file for the Zip Code table is located in the following directory path:
\$CARSPATH/modules/common/screens/tzip

You can access this table from the Systems Management: Table Maintenance Menu or from the specific module table maintenance menu.

Screen Example

The following Zip Code Table screen may vary from your table format and content due to your institution's particular specifications.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
          ** 1: zip_table table**

          ZIP TABLE

          Code.....[          ]
          City.....[          ]
          State.....[    ]
          County.....[    ]
          Residence Code..[    ]
```

Field Descriptions

The following describes the fields contained on the Zip Code Table screen.

City

The city associated with this zip code.

Example: Cincinnati (for the zip code 45241)

Code

The zip code. The zip code assigned to the city or county.

Example: 45241 (for the city of Cincinnati)

County

The county code associated with this zip code.

Example: HAMI (for the county of Hamilton)

Residence Code

The five digit residence code associated with this zip code.

State

The state code associated with this zip code.

Example: OH (for Ohio)

Report Example

The following Zip Code Table report may vary from your report format and content due to your institution's particular specifications.

Zip Code		City	St	Cty	Res	Active /Inactive Date	Date
45140-1111		Loveland	OH	WARR	00000		
45056		Oxford	OH	BUTL	00000		
43123		Grove City	OH	FRAN	00000		
74135		Tulsa	OK	TULS	00000		
74012		Broken Arrow	OK	TULS	00000		
74014		Broken Arrow	OK	WAGO	00000		
74017		Claremore	OK	WAGO	00000		
45213		Cincinnati	OH	HAMI			
45242		Cincinnati	OH	HAMI			
45215		Cincinnati	OH				
00000 0000		No City					
77651		Port Neches	TX				
60187		Wheaton	IL				
60188		Glen Ellyn	IL	WARR			
62002		Alton	IL				
60540		Naperville	IL				
76201		Denton	TX				
75287		Dallas	TX				
75090		Sherman	TX				
77056		Houston	TX				
75979		Woodville	TX				
43606		Toledo	OH	LUCA			
77623		Beaumont	TX				
67898		Troy	MI				

* - Record is inactive according to active/inactive dates
 Location: /disk06/carsdevi/modules/common/reports/tzip
 Revision: G.0.110.2 04/29/94 11:36:48

SECTION 5 – JENZABAR CX MACROS

Overview

Introduction

This section describes CX macros, defines specific values used throughout the system. The macros enable you to change the available options and functionality of CX without having to modify C code. By modifying macros, you can customize your implementation of CX, and make the system easier to maintain.

This section provides reference information about macros, and the macros used to set up the common elements of CX.

What Is a Macro?

A macro is an instruction that causes the execution of a pre-defined sequence of instructions in the same source language. A macro consists of uppercase letters and underscores, and is used in place of a text string within source files. CX expands the macro to the longer text during the installation process for a file.

You set up and modify macros using UNIX commands that are part of the *make* processor. The *make* processor translates and expands the macro to the longer text during the installation process for a file. Macros are contained in a common area (library) on the system.

CX uses the following kinds of macros:

- Enables - allows you to enable or disable a CX module or module or feature
- DBS_COMMON - allows you to define database values in screens
- Periodic - allows you to make changes on a periodic basis

Macros can perform one of the following functions:

- Define defaults on a screen (`_DEF`)
- Define valid values in a field (`_VALID` or `_INCL`)
- Enable system modules (`ENABLE_MOD`)
- Enable system features (`ENABLE_FEAT`)
- Establish a valid value for an include

Configuration Table

You make changes to enable macros using the Configuration table. For more information, see *Configuration Table* in *Common Tables and Records* in this manual.

The Relationship Among Macros, Includes, and C Programs.

For all elements of the product other than C programs, m4 macros contain the definitions of features that have been designed to be modified for each institution. These macros, located under \$CARSPATH/macros, are passed through the m4 processor.

CX contains an alternative method for the setting of features in C programs. Macros in the source code of C programs are not passed through the m4 processor because the C compiler has its own preprocessor, the cc.

To provide the same apparent functionality to C programs, a section in the include source directory has been allocated for a type of include file that acts as an intermediary between the m4 processor and the cc preprocessor. In operation, m4 macros are defined whose output is a valid cc macro. These m4 macros are placed in the include files. Then the include files are translated and the appropriate cc macro is placed in the include file. The installed include file is included by the C compiler at compile time so that the result of the compilation is influenced by the m4 macro.

Benefits of Jenzabar CX Macros

Introduction

Use macros within source files for menu options, screens, and reports to reduce the need for editing source files when you make changes to such items as defaulted values, included sets of values, or report headings.

The CX base product contains approximately 1500 menu options that are distributed among 320 menus. You can use a macro to make a change in one location; then, use the *make* processor to incorporate that change throughout the entire CX product.

For example, if your institution wants to change the base product word "session" to "semester," make the change in the appropriate macro file and then use the *make* processor to incorporate that change throughout the product. Changing a macro is simpler than locating all of the source files using a particular string of text and then ensuring that all of the text strings are modified.

Benefits of Jenzabar CX Macros

The following lists and describes the benefits of macros.

Improved productivity

Macros, because of their abbreviated state, reduce the amount of text that you need to insert in a file. After you test a macro definition to make sure that it works properly, you can use the macro to reduce typographical and logical problems.

Improved and maintained consistency

Macros provide a means for ensuring consistency of implementation among multiple files. For example, you can use macros to set the format for ACE report output. You can also use macros to create a comment that appears on the comment line throughout an application with the same wording each time.

Simplified customizations

The *make* processor maintains changes that your institution makes to the macro files. You can merge these changes with new releases of the macro files that CX electronically distributes.

Contents of a Macro File

Introduction

The macros in CX are processed through a UNIX utility called m4, or the m4 processor. Each macro is contained in a macro file. The macro files are located in the following directory path: \$CARSPATH/macros.

A macro file contains the following components:

- Comments
- Macro definition lines

A macro file can also contain m4_include statements. For details on the m4_include statements, see m4_include statements in this section.

Example Macro File

Following is an example of a macro located in the following directory path: \$CARSPATH/macros/custom/common.

```

{
----- comment
-----
Common Enable Macro Definitions
-----
}
comments
{*** Defines whether FPS is enabled ***} -----3
{*** Normally, this will be set to "Y". This would ***}--
{*** be set to "N" only if all printing was done in ***}--
{*** central location and not in individual end-user ***}--
{*** offices. ***}--3
m4_define(`ENABLE_FEAT_FPS', `Y') ----- macro definition line
3--- m4 command
3----- macro name
3----- macro name
```

The above macro, m4_define(`ENABLE_FEAT_FPS', `Y'), indicates the following:

The system will enable (turn on) the Forms Production System (FPS) feature because the macro definition is set to "Y" for yes. Wherever the macro `ENABLE_FEAT_FPS' is used in a menu, the *make* processor will enable the feature when the file is installed.

Parts of a Macro File

The following lists each part of a macro file and provides the content and specification for each.

Comment

A statement that provides information on the source code following it. Comments begin with the left brace symbol ({} and end with the right brace symbol (}). Notice that some comments also contain three asterisks (***) between the text and braces to help you separate the comment from the macro definition line.

Note: The system does not process comments. Comments separate major sections of macro files, and each section begins with a comment.

Macro definition line

An m4 command, the macro name, and the macro definition. The definition line appears on the next line after a comment.

m4 command

A command that is processed by the m4 processor, and that uses the following format: *m4_command*. Leave no blank spaces between the m4 command (e.g., "m4_define") and the opening parenthesis.

Macro name

Any collection of only uppercase letters and underscores (`_`) that appear between a single back quotation (```) and a forward quotation (`'`).

- Enclose the macro name and its corresponding macro definition within a single back quotation (```) and a single forward quotation (`'`).
- Leave a comma and one blank space following the forward quotation (`'`) of the macro name.

Macro definition

A collection of one or more characters that the *make* processor substitutes in the source file when the macro is translated and expanded. Enclose the macro definition within a single back quotation (```) and a single forward quotation (`'`).

M4_Include Statements

In addition to the components listed on the previous pages, a macro file can also contain `m4_include` statements. `m4_include` statements define the other macro files that the *make* processor will include when it expands and translates the macro file in which you are currently working.

Note: An `m4_include` statement is not the same as the `include` that is discussed in *Setting Up Includes* in this guide.

Some examples of `m4_include` statements are located in the following directory path: `$(CARSPATH)/macros/user/common`. Following is an example of three `m4_include` statements located in the common file:

```
Example: m4_include(m4PATH/custom/client.m4)
            m4_include(m4PATH/custom/configure.m4)
            m4_include(m4PATH/custom/periodic.m4)
```

The above `m4_include` statements indicate the following:

When you access a macro file, the *make* processor searches the install path for the list of `m4_include` statements that could contain a macro definition, and then it searches for the macro definition in the macro file. The *make* processor searches the client file first, and if it cannot find the macro definition, it searches the configure file, then the periodic file, until it locates the macro definition in the actual file.

The Four Types of Macro Files

Types of Macro Files

The following describes the types of macro files that are located in each of the four macro subdirectories.

Custom macro files

The `$CARSPATH/macros/custom` directory path contains macro files that your institution can customize. The institution maintains the custom macro files using the *make* processor.

System macro files

The `$CARSPATH/macros/system` directory path contains macros that pertain to implementing the UNIX operating system. There is one system macro file for each hardware platform that supports CX (e.g., AIX for UNIX on the IBM platform).

CAUTION: The institution cannot customize the macros located in the system macro files. CX maintains the system macro files.

User macro files

The `$CARSPATH/macros/user` directory path contains macros that define many of the standard defaults that a menu user sees on a screen (e.g., the default for the state in which the institution is located).

Note: The institution customizes the macros in the user macro files during the initial installation of CX. Once you set up the macros in the user macro files, you should never have to modify them again unless your institution makes a major policy or procedure change.

Utility macro files

The `$CARSPATH/macros/util` directory path contains macros that pertain to implementing the Revision Control System (RCS), the *make* macros, and the macro processor (m4) macros.

CAUTION: The institution cannot customize the macros in the utility macro files. CX maintains the utility macro files.

Macro Files That the Institution Can Customize

An institution can customize the following macro files:

- The custom macro files located in the following directory path:
`$CARSPATH/macros/custom`
- The user macro files located in the following directory path: `$CARSPATH/macros/user`

Note: The custom macro files contain the macros that an institution can customize. Although the institution can modify the user macro files, CX discourages this since the institution can customize the macros in the custom macro files.

If an institution makes a change to a macro in any of the custom or user macro files, you must install the files and reinstall the menu options, menu screens, PERFORM screens, program screens, and reports that use the macro you have changed.

Any changes that you make in a macro file are recorded by the Revision Control System (RCS). The RCS maintains a history of all changes made to a macro file. You can review the changes before installing the macro file.

Macro Files That the Institution Should Not Customize

CX processes depend on the macros contained in the system and utility macro files. Any changes made to the system macro files or the utility macro files could corrupt an institution's entire system.

CAUTION: Do not change the macro files listed below:

- The system macro files located in the following directory path:
\$CARSPATH/macros/system
- The utility macro files located in the following directory path: \$CARSPATH/macros/util

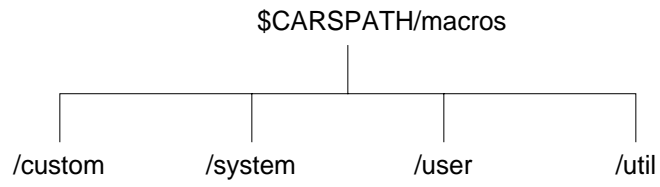
The Macro Directory Structure

How to Access the Macro Files

The CX macro files are located in the following directory path: `$CARSPATH/macros`.

Macro Directory Structure

There are four subdirectories in the `$CARSPATH/macros` directory path. The following figure shows the macro directory structure that is made up of these four subdirectories: `custom`, `system`, `user`, and `util` (utility).



Custom Macro Files

Descriptions of Custom Macro Files

This list describes macro files that an institution can customize. These macros are located in the following directory path: \$CARSPATH/macros/custom.

admissions

Macros that are unique to the Admissions module.

admrpt

Macros that are unique to reports in the Admissions module.

client

Macros that are unique to an institution. The client macros are strictly local customizations, meaning that they contain changes to the CX base product that are specific to the institution. CX never modifies macros in the client file.

Note: Use the client file to create a new macro that an institution wants to keep as a local customization; that is, the institution does not want CX to modify the macro using the *make* processor. The *make* processor does not modify local customizations.

CAUTION: The *make* processor translates and expands the macro definitions in the client file first, out of all of the custom macro files. If your institution selects a macro name in the client file that has the same name as a macro created by CARS, then the CX macro takes precedence and is translated and expanded.

comment

Macros used for comments that appear on the comment line of screens used throughout CX.

common

Macros that are shared between each administrative office at an institution.

configure

Macros that are used to configure an institution's database and to enable macros when CX is first installed. If an institution has not purchased the Financial Aid Packaging module, for example, then type an N as the macro definition for that module. The result is that none of the menu options associated with the Financial Aid Packaging module appear on the menus.

Note: Once the macros are set up in the configure file, an institution never needs to change them again unless the institution purchases additional modules in the future.

develop

Macros that are unique to the Alumni/ Development module.

financial

Macros that are unique to the Financial module (e.g., General Ledger, Subsidiary, Fixed Assets).

finrpt

Macros that are unique to reports in the Financial module (e.g., General Ledger, Subsidiary).

ltrwp

Macros that are used in the letter writing process for letters that an institution creates and sends.

periodic

Macros whose values change periodically, such as session, fiscal year, and calendar year. The macro definitions in the periodic file are defaulted into fields on the data entry screens, and contain the most current information that menu users see on the screens (e.g., FA9X for the current session of Fall 199X).

student

Macros that are specific to the student area (e.g., Financial Aid, Registration, Course Entry, Student Services).

sturpt

Macros that are unique to reports in the student area (e.g., Financial Aid, Registration, Course Entry, Student Services).

table

Macros that an institution uses for setting up standard default codes that menu users see in fields on screens (e.g., the default code for the Class field is FF for first-time freshman). These macros contain codes for fields in which Table Lookup windows are not available.

User Macro Files

Descriptions of User Macro Files

This list describes each macro file that is located in the following directory path:
\$CARSPATH/macros/user.

acct

Macros for customizing the general ledger account structure.

admrep

Macros for customizing profile reports in the Admissions module.

adr

Macros for customizing the name and address management (ADR) process.

arc

Macros for customizing ACE reports.

cmd

Contains the macros for customizing command scripts.

common

- Macros that are common to several types of files (e.g., frm.m4, opt.m4), and that provide the default values for several files.
- The m4_include statements that define the other macro files that the *make* processor is to include when it expands and translates the macro file in which you are working.

db

Macros for customizing the Informix data types.

doc

Nroff macros for formatting the documents located on CX

finrpt

Macros for customizing financial reports.

fonts

Macros for customizing the fonts on the laser printer.

fps

Macros for customizing the Forms Production System (FPS).

frm

Macros for customizing PERFORM screens

inc

Macros for customizing include files.

inf

Macros for customizing ISQL scripts.

lps

Macros for customizing ACE reports to produce a Letter Production System (LPS) data file.

ltb

Macros for customizing ACE reports that are used with the letter and label processing and formatting.

- ltr**
Macros for customizing nroff formats in ACE reports.
- ltrwp**
Macros for customizing the font type for letters.
- mnu**
Macros for customizing menu description files.
- oth**
Macros for customizing the formats for the ACT and ETS tapes used in the tape conversion process.
- prompt**
Macros for customizing menu option screen prompts.
- rep**
Macros for customizing ACE reports.
- sch**
Macros for customizing schema files.
- scp**
Macros for customizing script files.
- scr**
Macros for customizing application screen files.
- skl**
Macros for customizing skeleton files for home directories.
- srt**
Macros for sorting information on a page in reports.
Note: Sortpage is a program developed by Jenzabar that is an extension of ADR and is used to sort information in reports after CX has processed the report.
- sys**
Macros for defining system files.
- wp**
Macros for customizing the vi wp files.

Common Macros

Introduction

This section lists the following macros that are used in common by the applications of CX:

- Enable macros that enable shared features of CX
- Periodic macros that define values used jointly by applications in CX

Note: You make changes to enable macros using the Configuration table. For more information, see *Configuration Table* in *Common Tables and Records* in this manual.

Access

The common enable and periodic macros are located in the following file:
\$CARSPATH/macros/custom/common.

Enable Feature

A group of macros containing the word *enable* are used to turn on and turn off features of CX. Determining which of these macros to enable is a policy decision that an institution makes. The macros enabled in CX at an institution determine how to carry out your institution's policies and procedures.

Common Enable Macros

The following lists the common enable macros located in the *common* macro file.

Note: You make changes to enable macros using the Configuration table. For more information, see *Configuration Table* in *Common Tables and Records* in this manual.

ENABLE_FEAT_ALLOW_TRANS_CHG_FEE_BALANCE

Default setting: Y

Defines whether or not to allow a fee balance for a transcript charge in the Form Entry (forment) program.

ENABLE_FEAT_AUTOMODE

Default setting: N

Defines whether or not to enable the AUTOMODE feature in entry programs. AUTOMODE specifies that entry programs start in query mode and automatically select update or insert mode based upon the return status of the query function. When AUTOMODE is disabled, you can initially perform a query without the program automatically initiating update mode upon the return status of the query function.

ENABLE_CONTACT_TEXT

Default setting: Y

Defines whether or not to enable text blob fields in the Contact detail window, accessed in entry programs. When you enable this macro, you access via the blob field a text editor in which you can make unlimited comment entries.

ENABLE_FEAT_FORCEQUERY

Default setting: N

Defines whether or not to enable the FORCEQUERY feature in entry programs. FORCEQUERY specifies that entry programs start in query mode so that a user queries for an ID record before attempting to add the ID to the database. This feature reduces the possibility of duplicate ID records being added to the database.

Note: If you select Y, the system automatically changes to update mode immediately after you complete your query. It is recommended that you change this default to Y.

ENABLE_FEAT_FPS

Default setting: Y

Defines whether or not to enable the Form Production System (FPS). You enable FPS if printing occurs in individual end-user offices. You disable FPS if all printing occurs in one central location.

ENABLE_FEAT_IDPERMS

Default setting: N

Defines whether or not to grant permissions by office and by entry programs (those programs that utilize Library Entry (libentry) routines). If you enable the IDPERMS feature, entry programs perform lookups in the ID Permissions table (idperm_table) before allowing the update of an ID record (id_rec) or Profile record (profile_rec). If the office has the permissions to update the records, the program allows the update.

ENABLE_FEAT_IDS_ENTER_2_EXECUTE

Default setting: N

Defines whether or not to enable the IDS_ENTER_2_EXECUTE feature, which sets the Enter key as an execute key for the system-wide ID Query functionality.

Note: This feature is currently not implemented.

ENABLE_FEAT_LPS

Default setting: Y

Defines whether or not to enable the Letter Production System (LPS). You enable LPS if printing occurs in individual end-user offices. You disable LPS if all printing occurs in one central location.

ENABLE_FEAT_MULTI_SITE

Default setting: Y

Defines whether or not to set multiple site functionality when multiple sites are associated with the institution. Disable the MULTI_SITE feature when the institution has only one location of instruction, or does not differentiate between multiple locations of instruction.

ENABLE_FEAT_PREV_INTERACTIVE

Default setting: N

Defines whether or not the PREV-address logic in entry programs prompts the user before saving a previous (PREV) address in the Alternate Address record (aa_rec). If you enable the PREV_INTERACTIVE feature, entry programs prompt the user to respond yes or no when asked to save a previous address. If you disable the feature, entry programs automatically save a previous address.

ENABLE_FEAT_PREV_PHONE

Default setting: Y

Defines whether or not to invoke the PREV-address logic in entry programs when a user makes changes to phone number information. If you enable the PREV_PHONE feature, entry programs save a previous address in the Alternate Address record (aa_rec) when a user changes the phone number for an individual. If you disable the PREV_PHONE feature, entry programs invoke the PREV- address logic for address changes, but not phone number changes.

ENABLE_FEAT_TRANS_CHG_OVERRIDE

Default setting: N

Defines whether or not to enable the transcript charging override feature in the Form Entry (forment) program. When you enable the TRAN_CHG_OVERRIDE feature, a user can manually override a charge for a transcript.

Common Periodic Macros

The following lists the Common periodic macros located in the *common* macro file.

Accomplishment Table Macros

- ACCOMP_TYPE_DEF', `ACADEMIC'
- ACCOMP_TYPE_VALID', `SOCIAL,ACADEMIC,ATHLETIC,BUSINESS,SERVICE'
- ACCOMP_TYPE_INCL', `include=(ACCOMP_TYPE_VALID)'
- ACCOMP_TYPE_EG', `', eg: ACADEMIC, ATHLETIC'
Define accomplishment type values, the default value, and example values for the Accomplishments table (accomp_table). The system only accepts those values defined in ACCOMP_TYPE_VALID when adding an Accomplishment record.

Address Macros

- ADDREE_CODE_DEF', `S'
- ADDREE_CODE_VALID', `L,S'
- ADDREE_CODE_INCL', `include=(ADDREE_CODE_VALID)'
Define the values, the default value, and example values for the Addree Code Include. The default codes are: L for Label, and S for Salutation
- ADDREE_STYLE_DEF'
- ADDREE_STYLE_VALID', ``",F,I,M,N,P'
- ADDREE_STYLE_INCL', `include=(ADDREE_STYLE_VALID)'
- ADDREE_STYLE_EX' (F)ormal,(I)nformal,(M)atric,(N)ickname,(P)revious,blank'
Define the values, the default value, and example values for the Addree Style Include.
- ADR_JOIN_REL_DEF', `HW'
- ADR_JOIN_REL_VALID', `HW,PC," ''
- ADR_JOIN_REL_INCL', `include=(ADR_JOIN_REL_VALID)'
- ADR_JOIN_REL_EX', `(HW), (PC) or blank.'
Define the values, the default value, and example values for the Address Joining feature (ADR Joining).
- AA_PREV_MAINT_CODE', `PREV'
Defines the Alternate Address code for previous address.
- CTRY_DEF', `USA'
Defines the default Country value.
- CTY_DEF', `HAMI'
Defines the default County value.

Note: You should use the county that has the greatest number of inhabitants at the institution. The County value must exist in the County table (cty_table).

- INST_NAME', `CARS College'
 - INST_ADDR1', `4000 Executive Park Drive'
 - INST_ADDR2', `Cincinnati, Ohio 45241'
 - INST_ADDR3'
 - INST_CITY', `Cincinnati'
 - INST_ST', `OH'
 - INST_ZIP', `45241'
- Define the name and address of institution.

Note: These values are used by various forms (e.g. cash_rcpt), and screens.

- ST_DEF', `OH'
- Defines the default State value.

Note: You should use the state that has the greatest number of inhabitants at the institution.

- ZIP_DEF', `45014'
- Defines the default Zip Code value.

Note: You should use the zip code that will occur most frequently.

Business Macros

- BUS_TYPE_DEF', `CORP'
 - BUS_TYPE_VALID', `",CORP,PART,PROP'
 - BUS_TYPE_INCL', `include=(BUS_TYPE_VALID)'
- Define the values, the default value, and example values for business' types. The system only accepts those values defined in BUS_TYPE_VALID when adding an ID record for a business.

Campus Building/Facility Macros

- BLDG_DEF', `ADMN'
- Defines the default building code.

Note: Use a value that reflects the most commonly used building on the campus. This building must exist in the Building table (bldg_table).

- CAMPUS_DEF', `MAIN'
- Defines the default Campus value.

Communication Management Macros

- COMM_DEF', `LETT'
 - COMM_EG', ` , eg: (LETT)letters, (LABL)labels, (LTLB)both.'
- Define the default and example Communication code values.
- CTC_SPAN_WAIVE_DEF', `Y'
 - CTC_REISSUE_DEF', `Y'
- Define the default Contact span waived and reissue values.
- SBSCR_ACE', `ltrsbscr'
 - SBSCR_DEF', `ALUMNEWS'
 - SBSCR_OFFICE_DEF', `DEVL'
- Define the default subscription mailings values.
- TICK_ADM', `ADM'
 - TICK_LEAD', `LEAD'

- TICK_ALPR',`ALPR'
- TICK_DEV',`DEV'
- TICK_FA',`FA'
- TICK_ACAD',`ACAD'
- TICK_RECV',`RECV'
- TICK_OTHRECV',`ORCV'
- TICK_REG',`REG'
- TICK_SBSCR',`SBSC'
- TICK_MATRIC',`MAT'
- TICK_TRANS',`TRAN'
- TICK_EOPS',`EOPS'
- TICK_DSPS',`DSPS'
- TICK_DEF',`TICK_ADM'
- TICK_VALID',`TICK_ADM,TICK_LEAD,TICK_ALPR,TICK_DEV,TICK_MATRIC,TICK_TRANS,TICK_EOPS,TICK_DSPS,TICK_FA)'
- TICK_INCL',`include=(TICK_VALID)'
Define the Tickler program default values.

Note: All of the above values exist in the Tickler table (tick_table).

Date Macros

- ARC_DATE_DEF',`12/31/1985'
Defines the default archiving date value.
- DATE_FORMAT5',`mm/dd'
- DATE_FORMAT8',`mm/dd/yy'
- DATE_FORMAT10',`mm/dd/yyyy'
Define the default date formatting values.

Enrollment Status Macros

- ENRSTAT_ROUTE_DEF',`O'
- ENRSTAT_ROUTE_VALID',`O,S'
- ENRSTAT_ROUTE_INCL',`include=(ENRSTAT_ROUTE_VALID)'
- ENRSTAT_ROUTE_EX',`(O)ffice/outgoing or (S)tudent/incoming.)'
Define the default enrollment status routing values and includes.

Events/Scheduling Macros

- EVNT_TYPE_DEF',`SOCIAL'
- EVNT_TYPE_VALID' MUSICAL,DRAMA,ATHLETIC,SOCIAL,ACADEMIC,SERVICE,BUSINESS'
- EVNT_TYPE_INCL',`include=(EVNT_TYPE_VALID),'
Define the values, the default value, and example values for event types.
- GRPSCHD_TYPE_DEF',`SERVICE'
- GRPSCHD_TYPE_VALID' MUSICAL,DRAMA,ATHLETIC,SOCIAL,ACADEMIC,SERVICE,BUSINESS'
- GRPSCHD_TYPE_INCL',`include=(GRPSCHD_TYPE_VALID)'
Define the values, the default value, and example values for group event types.
- SCHD_STAT_DEF',`S'
- SCHD_STAT_EX',`(S)cheduled, (F)inished, (P)ostponed, (C)ancelled, (T)entative'
- SCHD_STAT_VALID',`S,F,P,C,T'
- SCHD_STAT_INCL',`include=(SCHD_STAT_VALID)'
Define the values, the default value, and example values for Activity/Visit, Event, Group and Scheduling Status. Valid values include:
 - S for Scheduled

- F for Finished
- P for Postponed
- C for Canceled
- SCHED_PLACE_DEF, `HS'
- SCHED_PLACE_EG, `, eg: (HS)High School, (CF)College Fair.'
- SCHED_PLACE_EX, `(CH)urch,(HS)High School,(CF)Coll. Fair,(CO)llege,(JC)Jr. Coll.,(JH)Jr. High'
- SCHED_PLACE_VALID, `HS,CH,CO,JC,JH,CF'
- SCHED_PLACE_INCL, `include=(SCHED_PLACE_VALID)'
Define the values, the default value, and example values for Activity/Visit Scheduling Place codes. Valid values include:
 - HS for High School
 - CH for Church
 - CO for College
 - JC for Jr. College
 - JH for Jr. High
 - CF for College Fair

Faculty Macros

- FACULTY_CONTRACT_DEF, `9'
- FACULTY_CONTRACT_VALID, `1 to 12'
- FACULTY_CONTRACT_INCL, `include=(" ",FACULTY_CONTRACT_VALID)'
Define the default faculty contract duration in months and include.

File Format Macros

- FILE_FORMAT_DEF, `stdlps'
- FILE_FORMAT_VALID, `rtf","stdlps","wordp"'
- FILE_FORMAT_INCL, `include=(FILE_FORMAT_VALID)'
Define file formats.

File Transfer Macros

- XFER_PROTOCOL, `ZMODEM'
Defines the file transfer protocol to be used when transferring files from the UNIX host system to a PC. ZMODEM for zmodem; XMODEM for xmodem; KERMIT for kermit; FTP for ftp; CSERV for QuickMate.
- XFER_REMOTE_DIR, `/wpwin'
Defines the receive directory for files that are downloaded using ftp as protocol with the xfer command from wpvi or from utility menu.
- XFER_REMOTE_HOST, `\$LOGNAME'
- XFER_REMOTE_USER, `\$LOGNAME'
Define the remote host and user for ftp used for downloading files in wpvi or via the utility menu option.

Form/Label Macros

- FORMENT_ALT_ID_NUMBER, `92394'
Defines the ID of a temporary id_rec for creating ad hoc alternate addresses. Form Entry allows you to specify an alternate ID (an ID other than the ID for whom the form is generated) without that individual actually having an id_rec in the database. To provide that functionality, however, *forment* requires an ID record that is used as temporary storage space for this alternate recipient name and address data. This macro specifies the ID number of that record. Make sure that the ID number you assign here is specially created only for *forment*.
- FORMENT_STU_EXIT_PASSWORD, `CARS'

Defines password used to allow the operator to exit student version of forment.

- FORMENT_PERM_CATEGORY', `FORMENT'
- FORMENT_PERM_CODE', `OVERRIDE'
Define Perm table category and perm code for allowing you to override the charges for transcript orders.
- FT_DEF', `FT_STANDARD'
- FT_VALID', `"FT_STANDARD", "FT_WIDE"``
- FT_INCL', `include=(FT_VALID)'
Define formtypes.
- FT_STANDARD'
Defines standard 80 column paper.
- FT_WIDE', `wide'
Defines 132 column wide paper.
- LBLFORM_DEF', `1up5x35'
- LBLFORM_VALID'`1up5x35,1up5x40,1up8x40,3up5x35,3up5x40,3up8x40,4up5x30,4up5x35,"3up5x35 1up8x40"``
- LBLFORM_EX', `1up5x35'
Define label form files.
- NROFF_TOTAL', `2000'
Defines Nroff limits for producing Letters/Labels.

ID/Profile Macros

- DENOM_DEF', `BAPT'
Defines the denomination default value. The value used must exist in the Denomination table (denom_table).
- ETHNIC_WHITE', `WH'
- ETHNIC_BLACK', `BL'
- ETHNIC_HISPANIC', `HI'
- ETHNIC_ASIAN', `AS'
- ETHNIC_AMERICAN', `AM'
- ETHNIC_NONRES', `NO'
- ETHNIC_UNKNOWN', `UN'
Define ethnic codes.

Note: The codes must exist in the Ethnic table (ethnic_table). If you want to add an ethnic code, you must define a new m4_define macro, and also add the value in Ethnic table. Additionally, you might need to modify applicable reports.
- HAND_DEF', `NO'
Defines the handicap default value. The value used must exist in the Handicap table (hand_table).
- ID_LEN', `6'
Defines the length of ID numbers, mainly used in Menu options. Do *not* change this value.
- MARITAL_DEF', `S'
- MARITAL_VALID', `S,D,P,W,M,T'
- MARITAL_INCL', `include=(MARITAL_VALID),'
- MARITAL_EX', `(S)ingle, (M)arried, (D)ivorced, single (P)arent, separa(T)ed, (W)idowed.'
Define the marital status default and valid values.

- OT_DEF', `18'
- PT_DEF', `12'
Define the Part-Time, Full-Time, Over-time default values.
- SEX_DEF', `M'
- SEX_VALID', `F,M'
- SEX_INCL', `include=(SEX_VALID)'
Define the sex default values
- SUFFIX_EG', `ESQ'
Defines the suffix default value.
- TITLE_DEF', `MR'
Defines the title default value. If you use this macro, ensure that it matches the SEX_DEF above by sex (e.g., TITLE_DEF of MR if SEX_DEF is M).
- VET_BEN_DEF', `N'
- VET_BEN_VALID', `D,V,N'
- VET_BEN_INCL', `include=(VET_BEN_VALID),'
- VET_BEN_EX', `(D)ependent, (N)on-veteran, (V)eteran'
Define the veteran's benefit values, the default value, and example values.

Interest Table Macros

- INT_TYPE_DEF', `ACAD'
- INT_TYPE_VALID', `SOCI,ACAD,ATHL,BUSI,RELI'
- INT_TYPE_INCL', `include=(INT_TYPE_VALID)'
- INT_TYPE_EX', `SOCI, ACAD, ATHL, BUSI, RELI'
Define the interest type values, the default value, and example values for the Interest table (int_table).

Involvement Table Macros

- INVL_TYPE_DEF', `ACADEMIC'
- INVL_TYPE_VALID', `SOCIAL,ACADEMIC,ATHLETIC,BUSINESS,SERVICE,DA_GVGCLU
B_MAN'
- INVL_TYPE_INCL', `include=(INVL_TYPE_VALID,COHORT_INVL_VALID)'
Define the involvement type values, the default value, and example values for the Involvement table (invl_table).

Printer Macros

- PRINTER_DEF', `\${CARSPRINTER}'
- PRINTER_VALID', `\${CARSPRINTERS}'
- PRINTER_INCL', `include=(PRINTER_VALID)'
Define the printers. Do *not* change these values.

Right To Know Macros

- COHORT_INVL_VALID', `ATHLAID,OTHER," ""
- COHORT_INVL_DEF', `ATHLAID'
- COHORT_CTGRY_VALID', `FFFT,FFPT'
- COHORT_CTGRY_DEF', `FFFT'
- COHORT_INVL_INCL', `include=(COHORT_INVL_VALID)'
- COHORT_CTGRY_INCL', `include=(COHORT_CTGRY_VALID)'
Define the default values for Right To Know processing.

School/Community College Macros

- SCH_CODE_DEF', `PU'

- SCH_CODE_VALID',`PU,PR'
- SCH_CODE_INCL',`include=(SCH_CODE_VALID),'
- SCH_CODE_EG',`, eg: (PU)blic, (PR)ivate.'
- Define the school code values, the default value, and example values.
- SCH_TYPE_DEF',`HS'
- SCH_TYPE_VALID',`HS,COL,CC,GRAD'
- SCH_TYPE_INCL',`include=(SCH_TYPE_VALID)'
- SCH_TYPE_EG',`, eg: (HS)High School, (COL)lege.'
- Define the school type values, the default value, and example values.
- MIS_DISTRICT_CC_ID',`100'
- MIS_DISTRICT_INCL',`include=(MIS_DISTRICT_VALID)'
- MIS_DISTRICT_VALID',`100,200,300,400'
- Define the Community College District College Identifier values, the default value, and example values used for MIS tapes.

Session/Academic Year Macros

- ACAD_YR_DEF',`ACAD_YR_CUR'
- ACAD_YR_VALID',`1900 to 2100'
- ACAD_YR_INCL',`include=(0,ACAD_YR_VALID)'
- ACAD_YR_EG',`, eg: ACAD_YR_DEF.'
- Define the academic year values, the default value, and example values.

Note: ACAD_YR_VALID should include a range of years that begins with the first year you plan to enter data and ends at some future year. For example, to enter transcript edit for FA 1982, the year must be within the prescribed range.

Ensure that ACAD_YR_INCL has the 0 value in it because certain sessions allow a year of 0.

- SESS_EG',`, eg: SESS_DEF.'
- SESSYR_EG',`, eg: SESSYR_DEF'
- SESSYR_DEF',`SESSYR_CUR'
- SESSYR_VALID',`SESSYR_PREV,SESSYR_CUR,SESSYR_NEXT, SESSYR_OTH)'
- SESSYR_INCL',`include=(SESSYR_VALID)'
- Define the academic Sessions values, the default value, and example values.

Site Macros

- ALL_SITES',`\${CARSSITE}'
- Defines the site value, which indicates to programs that all sites are to be included.
- SITE_DEF',`CARS'
- SITE_VALID',`null," ",SBVC,CARS,CHC'
- SITE_INCL',`include=(SITE_VALID)'
- Defines the valid Site code types

Telephone Number Macros

- INST_CASHIER_PHONE',`(513) 563-CASH'
- Defines the phone number for the Cashier, in
\$CARSPATH/accounting/forms/voucher/cash_rcpt, the old version of the cash receipt program.
- INST_BURSAR_PHONE',`(513) 563-5010'
- Defines the phone number for the Cashier, in
\$CARSPATH/accounting/forms/cashier/cash_rcpt, the current cash receipt used by the Cashier Program.

- **FA_PHONE**, `(513) 563-4542'
Defines the phone number the Financial Aid office, in `$CARSPATH/modules/finaid/forms/faentry/fatran`. You can print the number on the Financial Aid Transcript.
- **FA_DIRECTOR**, `CARS FA Director'
Defines the Director of Financial Aid's name, in `$CARSPATH/modules/finaid/forms/faentry/fatran`. You can print the name on the Financial Aid Transcript form.

Track Macros

- **TRACK_HOLD_DEF**, `S'
- **TRACK_SBSCR_DEF**, `D'
- **TRACK_VALID**, `A,C,D,F,S'
- **TRACK_INCL**, `include=(TRACK_VALID),'
- **TRACK_EX**, `(A)dmissions,(C)ommon,(D)evelopment,(F)inancial,(S)tudent'
Define track code default values, the default value, and example values.

Word Processing Macros

- **MERGE_WORDP_FILE_EXT**, `mrg'
Defines the filename extension for WordPerfect merge files, which are placed into the Merge subdirectory in `wpvi` when you generate merge data files instead of `nroff` letter files.
- **MERGE_WORDW_FILE_EXT**, `doc'
Defines the filename extension for Word for Windows merge files, the default value, and example values.

SECTION 6 – JENZABAR CX INCLUDES

Overview

Introduction

This section describes CX includes. An include is a statement that controls what operations are performed by the C code in a C program, and functions in one of the following ways:

- Defines a variable to equal a value for the *make* processor to use (or *include*) in a C program
- Defines a variable to turn on and off operations for the *make* processor to use (or *include*) in a C program
- Defines compilation values for an entry program.

An include is located in an include file. You can set up and modify includes from the UNIX shell using an editor. Using the *make* processor, CX expands and translates any macros referenced in the includes to the longer text. This is done during the installation process for a file.

This section provides the necessary information to set up and modify includes.

Policy Decision

Determining which includes to enable in CX is a policy decision that your institution must make. The includes you enable determine how to carry out your institution's policies and procedures.

Macro Dependency

Includes have a dependency on macros. Normally, you do not directly modify includes for the module. You must modify a corresponding macro value and then reinstall the include.

How an Include Works

Relationship Between a Macro, Include, and C Program

As the following figure suggests, an include functions as an intermediary between a macro and a C program.

---→macro-----→include-----→C program

An m4 macro cannot be used directly in a C program since the system does not process C program code through the m4 processor. Therefore, an include is used so that a C program can communicate and process a macro. An include statement in an include file contains the information for defining a macro using a syntax that a C program understands. C programs read and understand include files.

Contents of an Include File

Introduction

The includes on CX are processed through a UNIX utility called m4, or the m4 processor. Each application include is contained in a separate include file, and the include files are located in the following directory path: \$CARSPATH/include.

An include statement in a file contains the following components:

- An include command
- A variable
- A value (A value in an include statement is optional.)

Parts of an include file

Following is an example of an include statement that defines a variable equal to a value. The include is located in the following directory path: \$CARSPATH/include/custom/billing.

```
----- comment
|
/* ----- |
=====
      BILLING program compilation options.
=====
----- */
/* -----
      Estimated aid note symbol which can be displayed on
      the SDS BILL next to any financial aid with an
      "EA" aid amount status. |
|
|
----- */ +-----comment

-----include statement
|
#define EST_AID_SYMBOL ""
| | |
+--- include | +----- value
      command |
|
|
+-----variable
```

Note: An include that is used to define a variable to equal a value should always be placed outside of a comment (after the "*/" symbols). An include that appears outside of a comment will be substituted in the actual C program when you reinstall the C program. An include placed inside of a comment is not defined, and therefore will not be substituted in the actual C program.

For an example of an include that appears inside of a comment, see *Examples of Includes* in this section.

How to Interpret the Include

The example include on the previous page, `#define EST_AID_SYMBOL "*"` , indicates the following:

The include defines using an asterisk (*) as the estimated aid symbol in the Student Billing application. Because the example include appears outside of the comment, it will be substituted in the actual C program when you reinstall the C program in the following install path: `/src/stubill/billing`.

When you reinstall the C program, the *make* processor brings in the include file (`$CARSPATH/include/custom/billing`) containing the above include that defines using an asterisk (*) as the estimated aid symbol. Therefore, when menu users use the Student Billing application, they see an asterisk (*) as the estimated aid symbol.

CAUTION: Do not modify the C code in CX C programs unless you are experienced in using C code.

Description of the Parts of an Include File

The following lists and describes the content of each component of an include file and provides specifications for each component.

Comment

A statement that provides information on the source code following it.

Note: The system does not process comments. Comments separate major sections of include files, and each section begins with a comment.

Begins with the `"/*` symbol and ends with the `*/` symbol; both symbols are for C program comments.

Note: Some comments also contain five dashes (-----) between the `"/*` and `*/` to help you separate the comment from the include. The five dashes are optional.

Include command

The `#define` command that either defines a variable or defines a variable to equal a value. Leave one blank space between `#define` and the variable.

Value

Any collection of characters that the *make* processor substitutes in the source file when the source file is reinstalled.

Note: An include does not have to contain a value.

Enclose the value within double quotation marks (`"` and `"`).

Variable

Any collection of characters that comes after the include command. Leave one blank space between the variable and the value.

Examples of Includes

Introduction

An include functions in one of the following ways:

- To define a variable to equal a value
- To define a variable to turn on and off operations

Depending on the function of an include and whether or not an institution would like the include to turn on or turn off an operation in a C program, the include can appear inside or outside of a comment.

Example of an Active Include Outside a Comment

An include that defines a variable to equal a value always appears outside of a comment and is referred to as active. Placing an include outside of a comment indicates for the *make* processor to activate the include in the C program.

For an example of an active include, see *Contents of an Include File* in this section.

Example of an Inactive Include Inside a Comment

An include that defines a variable to turn off operations in a C program appears inside of a comment and is referred to as inactive.

Following is an example of an include that defines a variable for turning off an operation in a C program. Notice that the include statement is inside the comment. The include is located in the following directory path: `$CARSPATH/include/custom/billing`.

```
/* -----  
    Post students with a stuac_reg_stat = "B" (boarder) or "C" (confirmed).  
    Boarder status students must have a stu_acad_rec to be billed.  
#define POST_CONFIRM_ONLY  
----- */
```

Interpreting the Include Inside the Comment

The include in the previous example, `#define POST_CONFIRM_ONLY`, indicates the following:

The include defines turning off the operation in the Student Billing application for posting all charges and credits to a student's account only if a student is confirmed on campus. Because the include appears inside of the comment, it will not be substituted in the actual C program when you reinstall the C program in the following install path: `/src/stubill/billing/`.

When you reinstall the C program, the *make* processor brings in the include file (`$CARSPATH/include/custom/billing`) containing the above include which defines for the C program whether or not to turn off the operation for posting charges. Billing will always post. This macro defines which students it posts for (e.g., "post only confirmed students" versus "post all students").

Therefore, when menu users use the Student Billing application, the application does not post charges and credits to student accounts once the student is confirmed.

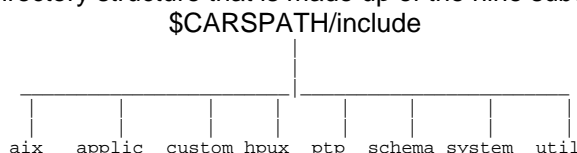
Nine Types of Include Files

How to Access the Include Files

The CX include files are located in the following directory path: \$CARSPATH/include.

Include Directory Structure

There are nine subdirectories in the \$CARSPATH/include directory path. The following figure shows the include directory structure that is made up of the nine subdirectories.



Types of Include Files

The following describes the types of include files located in the include subdirectory.

Aix

Includes that define the setup for the UNIX operating system on the IBM hardware platform.

Applic

Includes that define the setup for specific CX applications.

Custom

Includes that define the setup for CX applications and programs that an institution can customize.

Hpux

Includes that define the setup for the UNIX operating system on the Hewlett-Packard platform.

Ptp

Includes that define the setup for the process-to-process communications.

Schema

Includes that define how C programs know the data structure of CX.

Note: The *make* processor automatically processes the definition files (located in the install path) for all the schema files located in the \$CARSPATH/schema directory. However, the *make* processor does not automatically process the definition files for the Assessment Table and the Charge Table files located in the following directory path: \$CARSPATH/include/schema/student. CX has hard-coded both of these files to create the definition files.

System

Includes that define the characteristics of the operating system to C programs.

Util

Includes that define how some library functions that are used by CX programs work.

Include Files That an Institution Can Customize

Each institution can customize the custom include files located in the following directory path: `$CARSPATH/include/custom`.

If an institution modifies an include in any of the custom include files, then it must install the include file and reinstall the C program(s) that use the modified include.

If an institution modifies an m4 macro that is referenced in an include (e.g., the text "DA_GVGCLUB" is a macro in the following include: `#define DA_GCLUB "DA_GVGCLUB"`), then it must modify the m4 macro, install the macro file, and reinstall the include file that uses the macro and the C program(s) that use the include (in that order).

Include Files That an Institution Should Not Customize

Do not make any changes to the following include files unless you have experience in modifying include files.

- Aix
- Applic
- Hpux
- Ptp
- Schema
- System
- Util

An institution should never have to change an include in any of the above include files. However, an institution can reinstall any of these files if it changes a macro that is used in an include in one of the above include files. (In this case, the institution also needs to reinstall the include file(s) and C program(s) that reference the include.)

CAUTION: CX processes are dependent on the includes contained in the above include files, and any changes made to these files could corrupt an institution's entire system.

Custom Include Files

Descriptions of custom include files

This table describes each custom include file used in CX and located in the following directory path: \$CARSPATH/include/custom.

Acct

Includes for defining the values and functionality of the account structure.

Bgvoucher

Includes for defining the values and functionality of the background voucher program.

Billing

Includes for defining the values and functionality of the Student Billing application.

Bursar

Includes for defining the values and functionality of the Bursar application.

Cashier

Includes for defining the values and functionality of the Cashier application.

Ckslct

Includes for defining the values and functionality of the Check Select application.

Daentry

Includes for defining the values and functionality of the Donor Accounting Entry program.

Dirdep

Includes for defining the values and functionality of the Direct Deposit program.

Employ

Includes for defining the values and functionality of the Employee program.

F1099

Includes for defining the values and functionality of the 1099 program.

Faneed

Includes for defining the values and functionality of the Financial Aid Need Analysis application.

Finaid

Includes for defining the values and functionality of the Financial Aid application.

Fixpost

Includes for detailing the values and functionality of the Fixpost program.

Grading

Includes for defining the values and functionality of the Grade Entry application.

Libacct

Includes for defining the values and functionality of the accounting library.

Libbill

Includes for defining the values and functionality of the billing library.

Matric

Includes for defining the values and functionality of the Matriculation Entry program.

Payroll

Includes for defining the values and functionality of the payroll process and W2 forms process.

Phonebill

Includes for defining the values and functionality of an institution's telephone system process.

Progaudit

Includes for defining the values and functionality of the Auditing application.

Purch

Includes for defining the values and functionality of the Purchasing application.

Reglist

Includes for defining the values and functionality of the program used for printing class lists.

Trans

Includes for defining the values and functionality of the Transcript application.

Voucher

Includes for defining the values and functionality of the Voucher application.

W2tape

Includes for defining the values and functionality of the program that creates the W2 report and tape at the end of the calendar year.

Common Includes

The following lists the common CX includes.

#define ALL_SITE_CODE "ALL_SITES"

This site value indicates that all sites are to be included.

#define INST_STATE "INST_ST"

This defines the institution's state.

#define FORMENT_ALT_ID FORMENT_ALT_ID_NUMBER

This defines the FORMENT_ALT_ID used by the forment program to add alternate addresses to certain forms. Form Entry allows you to specify an alternate ID (an ID other than the ID for whom the form is generated) without that individual actually having an id_rec in the database. To provide that functionality, however, *forment* requires an ID record that is used as temporary storage space for this alternate recipient name and address data. This macro specifies the ID number of that record. Make sure that the ID number you assign here is specially created only for *forment*.

#define FORMENT_STU_EXIT_PASSWD "FORMENT_STU_EXIT_PASSWORD"

This defines the FORMENT_STU_EXIT_PASSWD used by the forment program to add alternate addresses to certain forms.

#define FORMENT_PERM_CTGRY "FORMENT_PERM_CATEGORY"

#define FORMENT_PERM_CD "FORMENT_PERM_CODE"

Perm table category and perm code for allowing operator to override the charges for transcript orders.

m4_keepif(ENABLE_FEAT_TRANS_CHG_OVERRIDE,~`Y~')

#define ENABLE_TRANS_CHG_OVERRIDE

This defines whether charge overriding is enabled in forment

m4_keepif(ENABLE_FEAT_ALLOW_TRANS_CHG_FEE_BALANCE,~`Y~')

#define ENABLE_ALLOW_TRANS_CHG_FEE_BALANCE

This defines whether a fee balance is allowed after charging for a transcript in forment.

m4_keepif(ENABLE_FEAT_IDPERMS,~`Y~')

#define ENABLE_IDPERMS

This defines whether or not an entry program (one that uses libentry) will employ the idperms functionality.

```
#define MERGE_WP_FILE_EXT    "MERGE_WORDP_FILE_EXT"
```

```
#define MERGE_WW_FILE_EXT    "MERGE_WORDW_FILE_EXT"
```

This defines the filename extension used for word perfect and word for windows merge files stored in the wpvi Merge directory.

```
m4_keepif(ENABLE_FEAT_MULTI_SITE,~`Y~')
```

```
#define ENABLE_MULTISITE
```

This defines whether or not libentry program will display the site in the detail windows option box. The site_rec will continue to be added in the background as required by the system, but the operators will not be able to select the option. This is only valid if the institution is a single site, as defined in the macros/custom/common macro listed below.

Setting Up Includes

What is the Process?

Each institution sets up includes while installing CX. The institution can modify includes when there is a new SMO release containing a new CX feature to enable, or when institutional policies or functions change.

The following shows the phases in the overall process of setting up include files.

1. Access the include files located in the following directory path: `$CARSPATH/include`.
2. Access an include file and modify the include(s).
3. Install the include file.
4. Reinstall the source program (C program) to make the include modifications known to the source program.

How to Set Up an Include

The following lists the steps to follow when you set up an include.

Note: Unlike macros, an institution cannot add includes to CX since the institution cannot change the C code in C programs.

1. Enter `cd $CARSPATH/include/custom` to access the directory containing the include files for implementation.

Example: `cd billing`

2. Enter `make co F=filename` to check out the specific file containing the include to be set up. (e.g., `make co F=billing`)
3. Enter `vi filename` to edit the file containing the includes (e.g., `vi billing`).
4. Use the arrow keys to move through the file and set up every variable in an include that you want to define and/or modify to as necessary.

Note: To define an include, place the include outside of the preceding comment. A comment begins with `"/*"` and ends with `"*/."`

5. Press `<Esc>`.
6. Enter `:wq` to exit and save the file.
 - Enter `make cii F=filename` to check in and install the file (e.g., `make cii F=billing`).
 - Reinstall all the files (e.g., reports, screens, programs) that reference the include.

SECTION 7 - FORM ENTRY PROGRAM

Overview

Introduction

This section provides reference information about the Form Entry (*forment*) program. The *forment* program has the following characteristics:

- Provides a means to execute table-specified processes to create forms or reports without using Contact records
- Produces Forms Order (*formord_rec*) records (whereas the normal Transcript process adds Contact records). The Forms Order record maintains information of all form orders placed on the system. The advantages of using the Form Entry program are:
 - You can view, update, and/or add holds for the student
 - This program shows where the official transcript was sent
 - This program maintains the transcript request history
- Uses the Form Order (*formord_table*) table to control who, where, and what type of form will be displayed and/or printed
- Enables menu users to display forms that a student previously requested
- Functions in two modes: Operator mode and Student mode
 - Is available to produce official transcripts using the Operator Form Requests menu option from the Registrar: Grading and Registrar: Transcript menus in Operator mode.
 - Must be implemented and monitored daily by a menu user when in Student mode. Student mode allows students who have a personal identification number to produce their own forms (e.g., unofficial transcripts and grade reports).

Program Features Detailed

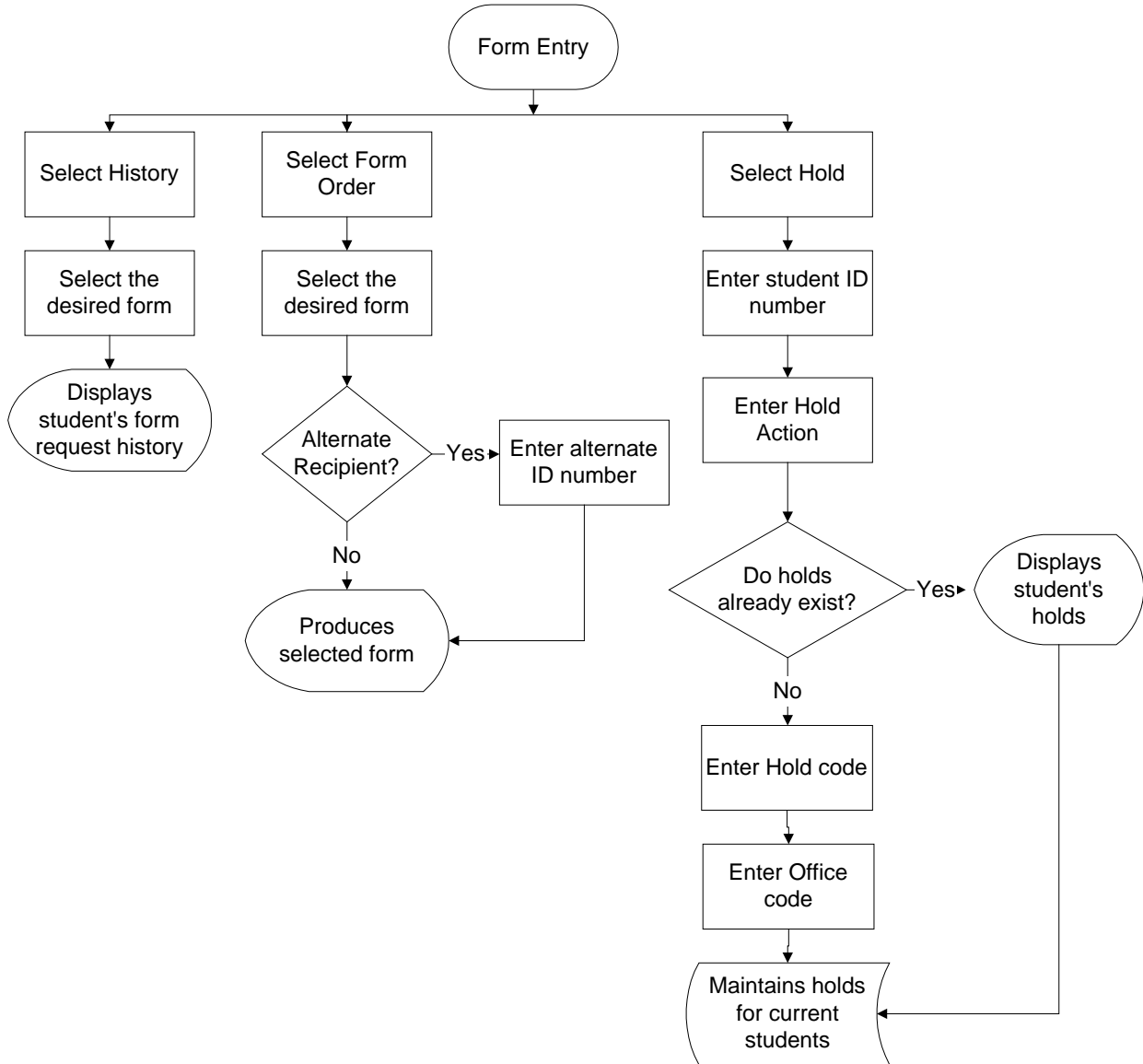
This section contains details about the following features of the Form Entry program:

- Process flow
- Parameters
- Program screens and windows

Process Flow

Diagram

The following diagram shows the flow of data in the Form Entry program.



Data Flow Description

The following describes the data flow in the Form Entry program.

1. The *forment* program presents options to the user.
2. The program processes the option selected.

In the case of the Form Order option, holds are first checked against the requested form. If no holds exist, the form is processed and the output generated.

Program Relationships

The following libraries are used by Form Entry.

- *libacct*
- *libbgv*
- *libbill*
- *libfee*
- *libgl*
- *Libreg*

Tables and Records Used

The Form Entry program uses the following Common tables and records.

Common tables and records

- *ctc_rec*
- *fac_rec*
- *formord_rec*
- *formord_table*
- *id_rec*
- *profile_rec*

Parameters

Introduction

CX contains parameters and compilation values for executing the Form Entry program. You can specify parameters to compile Form Entry in a specified manner at the time of execution.

Parameter Syntax

You can display Form Entry parameters by entering the following: **forment -**,

The following is the correct usage for running the Form Entry program from the UNIX shell:

forment -L sitecode [-o] [-F] -s session -y year -p program [-l printer] [-f billform]

Parameters that appear in brackets are optional. Parameters that do not appear in brackets are required.

Parameters

The following lists the parameters for running Form Entry.

-L sitecode

Required - Specifies the site at which to run the processes.

Example: forment -L CARS

-o

Optional - Specifies to run the processes in Operator mode.

-F

Optional - Specifies to use the fee collection option that calculates the current fee for the selected student.

-s session

Required - Specifies the session in which to run the processes.

Example: forment -s FA

-y year

Required - Specifies the year in which to run the processes.

Example: forment -y 2000

-p program

Required - Specifies the academic program in which to run the processes.

Example: forment -p UNDG

-l printer

Optional - Specifies the printer to use when executing the Fees option from the Operator mode of the Form Entry program.

Note: The default is *lpr*.

Example: forment -l lpt

-f billform

Optional - Specifies the billing form to use when executing the Fees option from the Operator mode of the Form Entry program.

Notes:

- The default is *SDS*.

- The *-f billform* parameter requires that you define Fee Collection requirements prior to using *formnt*.

Operational Modes

The following lists and describes the two modes in which Form Entry operates.

Student

If you do not use the “-o” parameter, CX runs the Form Entry program in Student mode. This means that students at the institution can use Form Entry to order their own unofficial transcripts and grade reports. In Student mode, Form Entry has fewer options and more security controls.

Notes:

- To exit *formnt* while in Student mode, press **<Ctrl-n>** (i.e., the command for starting the exit sequence), then enter the password defined in the macro `FORMENT_STU_EXIT_PASSWORD` located in the following directory path: `$CARSPATH/macros/custom/common`.
- The Student Form Request menu option is linked to the `profile_rec.password` field in the Profile record. The `profile_rec.password` field serves as a security number defined by the institution. It is the institution’s responsibility to create the student’s password number and notify the appropriate individuals.

Operator

The Operator mode enables you to query holds, update holds, and access the Fee Collection module to assess a fee for the request of an official transcript.

Program Screens and Windows

Introduction

Form Entry has screens and windows for performing the following interactive functions:

- Creating forms and reports
- Displaying a student's form order history

Access

The screen and window files for Form Entry are located in the following directory paths:

- \$CARSPATH/src/common/forment
- \$CARSPATH/modules/common/progscr/forment

Note: You can access windows from each program screen in Form Entry.

Screen Files and Table/Record Usage

The Form Entry screens and windows appear in the following files and use the indicated tables and records.

corrid

Contains the Alternate Recipient window.

Note: The *trans* program does not allow the menu user to select an alternate recipient. The system runs the entire transcript process, including the print file and printer selection.

Access: \$CARSPATH/modules/common/progscr/forment

Tables/Records:

- formord_rec
- st_table
- zip_table

frmord1

Contains the Student Form Request screen.

Access: \$CARSPATH/modules/common/progscr/forment

Tables/Records:

- formord_rec
- formord_table
- id_rec

frmord2

Contains the Operator Form Request screen.

Access: \$CARSPATH/modules/common/progscr/forment

Tables/Records:

- formord_rec
- frmtype_table
- id_rec

ordhist

Contains the Form Order History window.

Access: \$CARSPATH/modules/common/progscr/forment

Tables/Records:

- formord_rec
- formord_table

reqnum

Contains the Order Quantity screen.

Access: \$CARSPATH/modules/common/progscr/forment

Tables/Records:

- formord_rec

SECTION 8 – COMMON PROGRAMS

Overview

Introduction

This section describes the common programs of CX. Various products of CX use the common programs, which do not have product-specific functions. Certain common programs also provide solutions to system-wide integration issues, such as detecting duplicate ID records.

Common Programs in this Section

The common programs described in this section are as follows:

- ID Entry (identry)
- Duplicate ID Detection (dupid)
- ID Audit (idaudit)
- Database Administration (dbadmin)
- Sortpage (sortpage)

ID Entry Program

Introduction

To create and maintain ID records in entry program detail windows, you can use the ID Entry program (*identry*) by selecting the Add-ID command from detail windows that contain an ID number field.

Every CX user who can access any of the CX entry programs (i.e., programs that use the Library Entry program interface) can query, add, and update ID records in entry program detail windows. The CX entry programs include, for example, Student Entry (Stuentry), ID Entry (Identry), and Constituent Entry (Csentry).

Accessing the ID Maintenance Feature

You can access the ID Entry program from any detail window that contains an ID number field. If the Add-ID command appears on the command line, you can access the ID Entry program. For example, the following detail windows contain an ID number field and enable you to access the Add-ID command:

- First Relationship
- Second Relationship
- Employment/Work

The ID Add for Individual Screen

When you select Add-ID from a detail window that contains an ID number field, the ID Add for Individual data entry screen appears. You can use this screen to query, add, and update ID records for individuals, schools, or businesses.

Following is an example of the ID Add for Individual screen.

ID No	22465	SS No	301-88-9876	Add Date	01/23/1989
Title	MS Ms.	Last Upd		Telephone	513-791-8967
Name	Doe, Judy	Birthdate		Birthplace	
Suffix		Res: Country/St/Cty	USA BELL		
Address	9724 Ridgeway Ave.	City	Cincinnati		
State/Zip	OH 45241	Country	USA		
Deceased	N	Sex	M		
Correct Address	Y	Marital			
Alternate Address Code	PERM	Ethnic			
Privacy					
Occupation					
Denomination					

Note Continue to:
Page 2 for Business
Page 3 for School

Enter social security number. Format: ### # ####

Setup for this Feature

Your ability to use this feature depends upon the setup of each individual CX Library Entry application. Many entry application detail windows that contain an ID number enable you to use this feature.

Note: Consult your Jenzabar coordinator for information about the current setup of the CX entry applications.

Results of Selecting the Add-ID Command

When you select Add-ID from a detail window that contains an ID number field, the ID Add for Individual screen appears. You can use this screen to query, add, or update ID records. This screen appears in Query, Add, or Update mode, depending on how you select Add-ID command from the entry program detail window. From this screen, you can also access the School Entry and Business Entry screens.

The following lists the ways that you can use Add-ID to access the ID Add for Individual screen.

Note: You access the Add-ID command when the cursor appears in the screen's ID number field.

If you enter the Add-ID command from an ID number field that:

- Contains an ID number of zero, the ID Add for Individual screen appears in Add mode, and you cannot access Query or Update modes.
- Contains a non-zero ID number, the ID Add for Individual screen appears in the Query mode, and you can access the Update mode, but not Add mode
- Contains a non-zero ID number, and you select Finish while in Query mode, the ID Add for Individual screen switches to Update mode, and you cannot access Query or Add modes

Duplicate ID Detection Program

Introduction

Jenzabar designed the Duplicate ID Detection (*dupid*) program to assist you in maintaining the ID record (id_rec). The program provides the following:

- Ability to add new ID records with assurance that duplicates are not added
- Display of potential duplicates for analysis

The *dupid* program has the ability to modify and update data stored in the database; therefore, it should be used by:

- A Jenzabar coordinator
- A staff programmer with experience using CX
- A person responsible for maintaining the ID record

Dupid Terms

The following are definitions of terms and concepts used with the *dupid* program.

Test Function

A test function performs an evaluation on a section of data. *Dupid* performs each test on two records:

- Selected and Matched records for interactive mode
- Temporary and Matched records for review mode

The test function has a configurable weight and returns a level of confidence that the records are duplicates. Examples of test function:

- A test on name. The test function compares the names in both records and returns a value based on how closely the two names match.
- A test on sex returns either an equal value or a not equal value.
- An invalid test, which tests records with missing or invalid data. The invalid test does not affect the test function weight or the level of confidence.
- A valid test, the opposite of an invalid test, which tests good data to determine a confidence for the data.

The searching process compares the two records with a series of test functions. At the conclusion of the tests a normalized level of confidence and test weight are returned.

Weight

Some test functions are performed on pieces of data that have been determined to be more meaningful than others. A test on the social security number is a much more conclusive test than a test on street address. Thus the test function for social security numbers should have a higher weight than a test function for street address.

Confidence

The confidence is how closely the two records in the interactive mode or the review mode match. The level of confidence is a numeric value. The sum of all test functions performed on two records gives a raw level of confidence. This value is normalized and ranges from zero to one hundred. Program screens, documents and schema definitions sometimes abbreviate level of confidence as confidence or conf. Using test data at CARS, a confidence level greater than 70 or 80 is typically a good indicator a record is a duplicate. The program has options allowing the user to specify the minimum level of confidence.

Tests

The tests value indicates how much information was available to determine the confidence. A higher value indicates more information was used. The tests value is the ratio of how many test functions performed a valid test to how many test functions were performed. This value is normalized from a raw number to a range of zero to one hundred. Using test data at CARS, a test value greater than 40 is a good indication that enough data was available.

Normalization

This is a process where a number is normalized from some raw value to a value between zero and one hundred. Rounding errors may affect a normalized value by a few percentage points. It is unlikely that any normalized value will ever be exactly one hundred.

The example below shows some of the internal math involved in calculating the confidence and tests values.

Primary Data:			Secondary Data:		
Doe, John,, Jr.			Doe, Jonathan L.		
123 Main Street			123 Main St.		
Anywhere, OH 12345			Any Where, OH 12345-6789		
< No ss_no >			999-52-1238		
05/29/53			00/00/00		
			<--- Birthdate		
Test	Weight	Valid	Conf	Tests	Total

Name	15	Y	12	15	15
Birthdate	3	N	-	-	3
Address	10	Y	9	10	10
City	9	Y	9	9	9
State	5	Y	5	5	5
S.S.No.	25	N	-	-	25

Raw Totals:	^	^	35	39	67

The configured weight of a test function	---	Was there information for a valid test.	-----	Accumulated raw values from each test function	---
Normalized Confidence:	Raw Conf / Raw Tests * 100				
	35 / 39 * 100 = 89				
Normalized Tests:	Raw Tests / Raw Total * 100				
	39 / 67 * 100 = 58				

Match

A match is when the Selected record from the interactive mode or the Temporary record from the review mode is considered a possible duplicate of the Matched record from the database table id_rec. This is usually a one to many relation. One Selected or Temporary record should match one or more of the Matched records.

Match Table

The match record is a database table that stores potential matches between the temporary table, idtmp_rec, and the primary table id_rec. This table is called idtmp_match_rec and is created by the *dupid* program. Additional records are added to idtmp_match_rec each time the *dupid* program runs in background mode.

Program Arguments

You can use several options to enable or disable process features of *dupid*. The following shows how you specify these options:

Example: `dupid [-r] [-b] [-u] [-f First] [-l Last] [-c Confidence] [-t Test] [-x]`

The following lists the options and what they signify to *dupid*.

-r
Runs *dupid* work in review mode.

-b
runs *dupid* work in background mode.

-u
Runs *dupid* work in update mode.

-f (First)
The first ID number to be tested.

Note: The first and last parameters permit you to designate a range of ID numbers. *Dupid* tests only ID numbers in the ID record (*id_rec*) that fall between the first and last ID.

-l (Last)
The last ID number to be tested (0 = Last ID).

Note: The first and last parameters permit you to designate a range of ID numbers. *Dupid* tests only ID numbers in the ID record (*id_rec*) that fall between the first and last ID.

-c (Confidence)
The minimum confidence level required (0-100).

Note: This is the minimum normalized confidence value required for a record to be considered a potential duplicate.

-t (Test)
The minimum test weight required. 0-100

Note: This is the minimum normalized tests value required for a record to be considered a potential duplicate.

-x
Run using extended debug data

Note: When this option is turned on, the error handling function will print the table name and line number where the error function was called. This is useful when trying to find errors.

Dupid Modes

The following are the modes in which *dupid* runs.

Note: If a combination of the review, background and update options are specified, review mode takes precedence over background mode, which takes precedence over update mode.

Background Mode

Designed to be run overnight, this mode has no screen output. *dupid* will read from the Informix database table `idtmp_rec` and compare those records with the database table `id_rec` for possible matches.

Interactive Mode

The Interactive Mode is designed to be used in two ways:

- A single outside source check against the database table `id_rec` using the Input command
- An analysis of suspected duplicates from within the database table `id_rec` using the Query command.

Note: If you select the Input command, the Primary ID is set to zero.

Review Mode

After running *dupid* in background mode, the review mode gives you an opportunity to interactively confirm or reject all records which are considered duplicates.

Update Mode

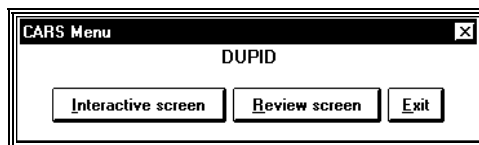
After running *dupid* in background mode and doing any necessary work in review mode, you can use update mode to update the database as described in stage two of the Command Options/Update section. This is useful when a large number of records are to be added and it is not possible nor desirable to tie up a terminal for an extended period of time.

Dupid Main Menu

When you execute *dupid* without specifying background mode, the DUPID Main menu appears. You use the menu options to select the mode in which to run the program. You select:

- **Interactive** to enter the Interactive mode
- **Review** to enter the Review mode
- **Exit** to Exit from the *dupid* program

The following is an example of the DUPID Main menu.



Database Tables Used by Dupid

The following tables are used by *dupid*:

ID record (`id_rec`)

Used in the following modes:

- In background mode, *dupid* tests the `id_rec` against a large number of new data about to be added to the system.
- In interactive mode, *dupid* tests the `id_rec` against itself (Query command) or a single outside source of new data (Input command).

ID Temporary record (idtmp_rec)

Stores new data to be added to the system.

ID Temporary Match record (idtmp_match_rec)

Stores potential duplicates between idtmp_rec and the id_rec. It also stores the classification between the Temporary and Matched data records from the Review mode.

Profile record (profile_rec)

Used to get birthdate and sex. It also performs an existence test for the data.

School record (school_rec)

Used as an existence test for the school record.

Business record (bus_rec)

Used as an existence test for the business record.

Church record (church_rec)

Used as an existence test for the church record.

First Relationship record (relation_rec)

Used to determine if two IDs which would normally be considered potential duplicates are in fact the same person. An example would be John, Doe Jr., a student. His father is also on a mailing list. Both addresses, zip codes, and many other factors are going to be identical. If both IDs being tested are linked by a relationship record, they will be rejected.

Not Duplicates record (nodup_rec)

Used for IDs which are not duplicates. Once the Primary ID and the Secondary ID are placed in this table, they will not appear on the screen as possible duplicates of one another again.

Modifying Table Definitions

CX allows clients to modify database definitions of the above tables. When you modify one of the tables, remember the following:

- If you change the length of any character field, you must recompile the following:
 - The *dupid* program
 - The *libdup.a* library
 - Associated program source code
- CX strongly recommends that the length of character fields in the idtmp_rec table match the length of their respective fields in the id_rec table.

Loading Data

Before running *dupid*, you must load new data into the ID temporary record (idtmp_rec). You can use several methods to load data, including:

- CX tool, Tape Conversion (See *Using Tape Conversion* in the *CX Implementation and Maintenance Technical Manual*)
- INFORMIX tools, which add records to a database table (See the appropriate INFORMIX documentation)

Note: Jenzabar recommends that you load a moderate number of records into the idtmp_rec until threshold values can be established because:

- The loaded data is slightly different
- A degree of intuition is initially required to establish the confidence level and tests passed

Running Duplicate ID Detection in Background Mode

Introduction

Background mode works best as either a scheduled process or as a background task in an interactive login.

Scheduling a Process

The following example procedure provides the commands you enter to schedule *dupid* as a background process. This procedure will begin the *dupid* program in background mode at eight in the evening. The minimum confidence level is set at 50. The minimum tests value is 20.

1. At the shell prompt, enter: **at 2000**

Note: 2000 equals 8:00 p.m.

2. Enter: **dupid -b -c 50 -t 20**

Refer to the *Program Arguments* section for more information on specifying parameters. In this example, you specified the following arguments:

- Background mode processing
- Confidence level to 50
- Test value to 20

3. Enter the End of File (EOF) command: **<Ctrl-d>**

The system displays the following messages:

- “warning: commands will be executed using /bin/sh”
- “job 651888001.a at Tue Aug 28 20:00:00 1990”

Starting from an Interactive Login

The following example procedure provides the commands you enter to immediately start *dupid* from the shell. This procedure will begin the *dupid* program in the background with a minimum confidence level of 65 percent. The minimum test defaults to zero.

1. At the shell prompt, enter: **dupid -b -c 65 &**

2. The system displays the following messages:

- At the start of the process: “[1] 12345”
- At the completion of the process: “[1] Done dupid -b -c 65”

Dupid Configuration

All parameters concerning configured weights are located in an Include table. Its location is `$CARSPATH/include/custom/dup`.

The items listed below are from the Include table and are the definitions for test functions which are either absolutely true or absolutely false. An invalid test returns `DUP_NULL`, which is defined as zero.

```
#define DUP_EQU_BIRTHDATE      (3)
#define DUP_NEQ_BIRTHDATE     (-1)
#define DUP_EQU_TYPE          (1)
#define DUP_NEQ_TYPE          (0)
#define DUP_EQU_SEX           (1)
#define DUP_NEQ_SEX           (-1)
#define DUP_EQU_STATE         (5)
#define DUP_NEQ_STATE         (0)
```

In the next list, the items are for test functions which return a degree of matching. If a function has a weight of fifty, then a return value of fifty is a perfect match. A return value of zero is totally non-matching. Some round-off errors may occur.

```
#define DUP_EQU_SSNO          (25)
#define DUP_NAME_WEIGHT       (15)
#define DUP_ADDR_WEIGHT_A     (10)
#define DUP_ADDR_WEIGHT_B     (7)
#define DUP_ZIP_WEIGHT        (12)
#define DUP_CITY_WEIGHT       (9)
```

Limitations

All ID records must have matching Soundex codes before *dupid* will test the ID when running *dupid* in interactive mode. The program, `SNDXINIT`, must be executed before using *dupid*.

System Demands

Note the following about system demands of *dupid*.

Background mode

In background mode, *dupid* uses a significant amount of system resources, including:

- Database disk I/O
- Processor time

Dupid is designed to use as little memory as possible while in background mode.

Note: Records are added to the database table `idtmp_match_rec` every time *dupid* is run in background mode.

If *dupid* is executed several times using the same data, redundancy and a large `idtmp_match_rec` table will occur.

Review mode

Running in review mode, *dupid* uses more memory and less CPU time. Much of the additional memory is used to store a dynamic list of the match table. Additional memory is used for screen functions.

Running Duplicate ID Detection in Interactive Mode

Introduction

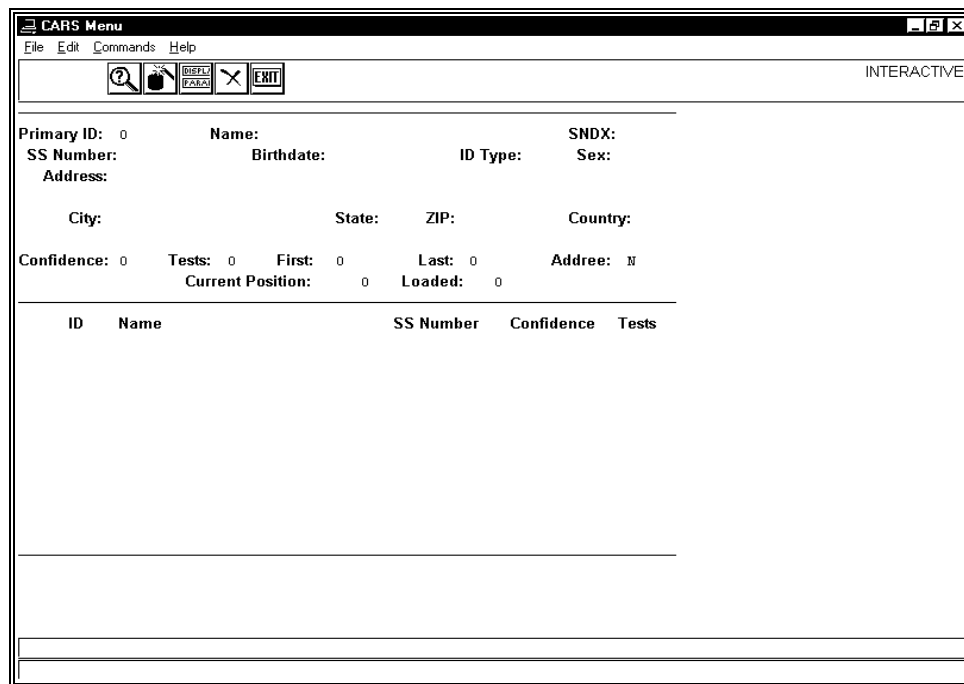
You run the Duplicate ID Detection (*dupid*) program in interactive mode to find individual ID records, which are suspected to be duplicates in the ID record (id_rec).

Jenzabar designed the Interactive Mode to be used in two ways:

- A single outside source check against the ID record (id_rec) using the Input command
- An analysis of suspected duplicates from within the database table id_rec using the Query command

Interactive Mode Screen Example

The following is an example of the *dupid* interactive mode screen.



Data Displayed On The Screen

The *dupid* interactive mode screen contains the following two primary areas of information on the screen.

Scrolling Area

The first area on the screen is the scrolling area where data from the match table is displayed. The data is sorted by the largest confidence value when loaded. Two primary fields in this section are as follows:

Current Position

Lists the position you are presently at in the Loaded records.

Loaded

Contains the number of records that are loaded into memory and not marked for deletion.

Note: The following data appears automatically when you use the Query ID command, and must be entered when you use the Input command:

- Name

- SS Number
- Birthday
- Sex
- Address
- City
- State
- ZIP
- Country
- ID Type

Note: ID Type lists the type of the Primary ID number (e.g., P for person, B for business, C for church, S for school).

Statistical Area

The second area on the screen is a statistical area. Each data field contains some statistical piece of information that will help the operator determine if the Primary and Secondary names are duplicates. Fields in this section are as follows:

Primary ID

From the Query ID command, lists the ID number from the database table `id_rec` and from the Input command, the ID number is set to zero.

SNDX

Soundex code is used to list the possible matches of the Query ID command or the Input command.

Fields Accessed with the Parameters Command

You can access the following fields through the Parameters command.

Note: After setting the parameters to a certain value, only those possible matches that are greater than or equal to that setting appear on the screen for the Input and/or the Query commands.

Confidence

Lists the minimum confidence value you want listed for the Matched Records.

Tests

Lists the minimum test value you want listed for the Matched Records.

First

Lists the starting value of your search in the database table `id_rec`.

Last

Lists the ending value of your search in the database table `id_rec`.

Adree

Asks you if you want nicknames of the Primary ID to be included.

Initial Screen Commands

The following are the commands that you can initially use in the *dupid* interactive mode screen.

Query ID

Selects a Primary ID as the selected record. It is used to check the database table `id_rec` against itself. Possible matches are listed on the screen sorted by confidence value.

Input

Allows you to manually enter data for the Primary ID. Use this when you are checking an outside source against the database table `id_rec`. Possible matches are automatically listed on the screen sorted by confidence value.

Note: *Dupid* sets the primary ID to zero.

Parameters

Allows you to set:

- Minimum confidence level
- Minimum test level
- First and last ID
- Addree

These parameter settings are set globally for both the Input and Query commands. After setting the parameters to a certain value, only those possible matches that are greater will appear on screen for the Input and/or Query commands.

Finish

Finishes the interactive mode and places you in the *dupid* main menu.

Exit

Exits from the *dupid* program. If you made changes, exiting without updating loses all changes. A safety prompt exists to help prevent lost data.

Screen Commands after Selecting Query ID or Input

The following are the commands that you can use after selecting the Query ID command and/or Input command.

Add

Adds the Primary ID to the database table `id_rec` for the Input command. You are prompted if the record needs a `Profile_rec` and also for an output table name for the added record to the database. The default is `/${HOME}/dupid.out`. `Dupid.out` needs to be checked after the Add command to see if there were any problems in adding the record to the database table `id_rec`.

Execute

Starts the actual search for the Query ID command for the Input command after a parameter has been changed. The Query ID and Input options set the Selected record and this option finds the Matched records. Possible matches are listed on the screen sorted by confidence value.

Sort

Sorts the Matched records by either name, confidence, social security number, or ID number for the Query ID command or the Input command.

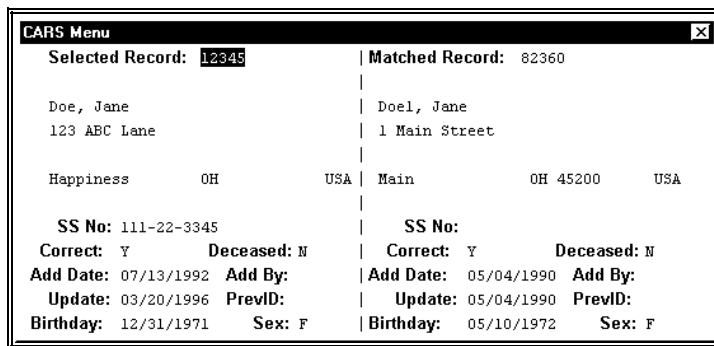
Detail

Pulls up detailed information about the two records for the Query ID command and the Input command.

Interactive Mode Detail Pop-up Window

The following is an example of the *dupid* interactive mode pop-up window. You access this window by selecting **Detail** from the *dupid* interactive mode screen.

Note: The primary ID appears in the window when you access it after selecting the Input command.



Command Options for Detail Window

The following are the commands that you can use in the Detail window.

Toggle ID

Toggles back and forth between the Selected and Matched records for the Query ID selection.

Update

Allows you to update the information presented on the window.

Note: You cannot update information from the Selected Record when coming from the Input command.

Non-dup

Places the Selected and Matched records from the Query ID command into the database table nodup_rec.

Finish

Finishes the window and allows you to select another ID to detail on.

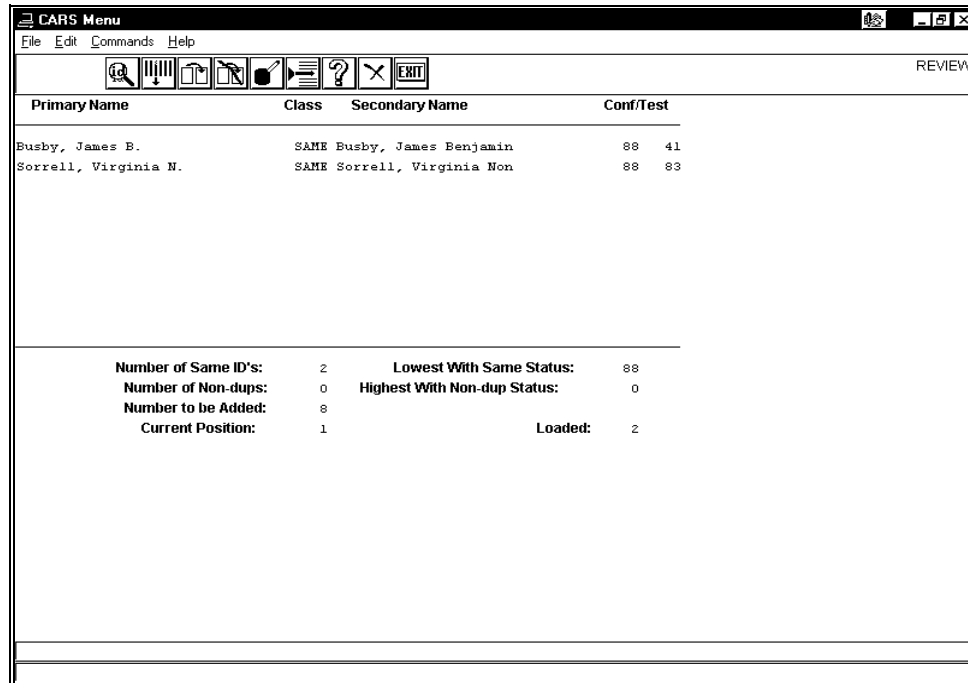
Running Duplicate ID Detection in Review Mode

Introduction

You run the Duplicate ID Detection (*dupid*) program in review mode after running the program in background mode. Review mode permits you to examine the records selected by *dupid* as potential duplicates.

Review Mode Screen Example

The following is an example of the *dupid* review mode screen.



The screenshot shows a window titled "CARS Menu" with a menu bar (File, Edit, Commands, Help) and a toolbar with icons for search, list, print, help, and exit. The main area is labeled "REVIEW" and contains a table with the following data:

Primary Name	Class	Secondary Name	Conf	Test
Busby, James B.	SAME	Busby, James Benjamin	88	41
Sorrell, Virginia N.	SAME	Sorrell, Virginia Non	88	83

Number of Same ID's:	2	Lowest With Same Status:	88
Number of Non-dups:	0	Highest With Non-dup Status:	0
Number to be Added:	8		
Current Position:	1	Loaded:	2

Data Displayed On The Screen

The *dupid* review mode screen contains the following two primary areas of information on the screen.

Scrolling Area

The first area on the screen is the scrolling area where data from the match table is displayed. The data is sorted by name when it is loaded. The Primary Name lists the name from the data file; the Secondary Name lists the name from the id_rec table.

Statistical Area

The second area on the screen is a statistical area. Each data field contains some statistical piece of information that will help the operator determine the impact of large scale operations like remove or update. Fields in this section are as follows:

Same IDs

Lists a count of how many times a classification contains the code SAME. Marking a record as being the same means that it will be removed from the table idtmp_rec. Remaining records may be added to the database table id_rec.

Non-dups

Lists a count of how many nodup_rec records will be added to the system. It is important to note that once a record has been marked as being the same, no nodup_rec record will be added.

To Be Added

Lists a count of how many records will be added to the id_rec table when the database is updated.

Current Position

Lists the position you are presently at in the Loaded records.

Lowest With Same Status

Contains the lowest confidence level of the record that has a classification of SAME.

Highest With Non-dup Status

Contains the highest confidence level of the record that has a classification of NON.

Loaded

Contains the number of records that are loaded into memory and not marked for deletion.

Screen Commands

The following are the commands that you can initially use in the *dupid* review mode screen.

Edit

Allows you to interactively confirm or reject which records in the idtmp_rec are duplicates. When *dupid* exits, the records will be processed according to the code you entered in the Class column. Enter one of the following:

SAME

The Primary and Secondary names are duplicates of one another and thus will not be added to the database table id_rec.

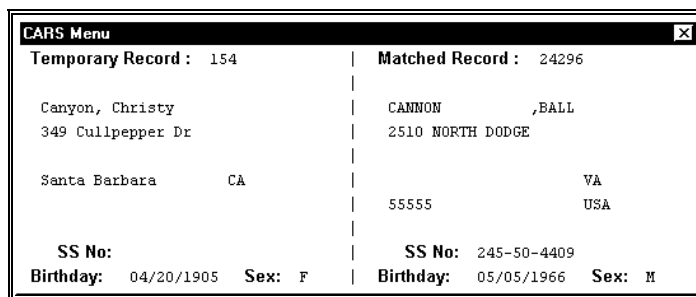
NON

The Primary and Secondary names are not duplicates and will be added to the database tables id_rec and nodup_rec. Once the Primary ID and the Secondary ID are placed in the nodup_rec table they will not appear on the screen as possible duplicates of one another again.

<blank>

The Primary and Secondary names are not duplicates and will be added to the database table id_rec.

In edit mode if you need additional information about the records, you can access the *dupid* review mode detail window. This window provides details about the two records being compared.



Global Submenu

Three items appear in the global submenu, including:

Remove

Globally deletes match records that have a confidence value less than some specified value. The user is prompted for this specified value. Records are marked as being removed but are not removed from memory at this time. This allows records to be restored if the database has not been updated.

Mark as Same

Sets the classification code to SAME for all records with a confidence greater than or equal to some specified value. The user is prompted for this specified value. SAME means the Primary and Secondary names are duplicates of one another and thus will not be added to the database table `id_rec`.

Mark as Non-Dup

Sets the classification code to NON for all records with a confidence less than some specified value. The user is prompted for this specified value. NON means the Primary and Secondary names are not duplicates and will be added to the database tables `id_rec` and `nodup_rec`. Once the Primary ID and the Secondary ID are placed in the `nodup_rec` table they will not appear on the screen as possible duplicates of one another again.

Note: Marking a record as <blank> means it will be added to the database table `id_rec`.

Update

Does the following:

1. Updates the database match table, `idtmp_match_rec`, with the data displayed on the screen and in memory. Records marked for deletion in memory will be removed from both disk and memory. Classification codes in memory will be stored to disk.
2. Stage two of this command will add records to the database table `id_rec` and add records to the database table `nodup_rec` if marked as 'NON'.
3. You are next prompted for an output table name for the added records to the database. The default is `$(HOME)/dupid.out`. You need to check `Dupid.out` after the Update command to determine if there were any problems in adding the records to the appropriate database tables.

Restore

Unmarks all records marked for deletion by the Global Submenu Remove option.

Help

Displays a brief help screen. This screen is located in `$(CARSPATH)/modules/common/progscr/dupid/helprev`.

Finish

Finishes the review mode and places you in the *dupid* main menu. If changes were made, exiting without updating will lose all changes. A safety prompt exists to help prevent lost data.

Exit

Exits the *dupid* program. If changes were made, exiting without updating will lose all changes. A safety prompt exists to help prevent lost data.

ID Audit Program

Introduction

Because CX uses identification numbers extensively, Jenzabar designed the ID Audit (*idaudit*) program to assist users in locating and removing problem ID records (*id_rec*). You can run *idaudit* to locate all the records with detached ID number(s) in the database and, if desired, remove all unwanted records. An ID number can reside in records other than the ID record (*id_rec*) because many CX programs use ID numbers during normal processing. The following scenarios can necessitate the use of *idaudit*:

- Users can accidentally enter an individual into the database more than once. The assigned duplicate IDs then appear in various records throughout the database.
- The institution no longer needs the records for an individual and wants to delete them

Note: You can run *idaudit* with or without command line options. *Idaudit* does not identify duplicate ID numbers.

The Process

The ID Audit program (*idaudit*) provides a means of removing old, unused, and unwanted ID records and records with detached IDs from the database. It uses the following process.

1. *Idaudit* selects and reports ID records that have a purge date earlier than the date the program is run (usually the system date).
2. *Idaudit* searches for and reports detached IDs, which are records in other files that no longer have an ID record associated with them.

Permissions

Idaudit does not attempt to delete any records unless you specify the *-p* (purge) or *-u* (delete) options. However, if you run *idaudit* with the *-u* or *-p* option, and *idaudit* cannot delete record(s) from a file, *idaudit* does the following:

- For the *-u* option, prints a "NOT DELETED" message beside the non-deleted record as well as indicating the deleted records
- For the *-p* option, prints a "NOT PURGED" message beside the un-purged ID number as well as indicating purged ID's, and informs you to run *idaudit* using the *-i* option on the particular ID number(s).

Note: An error message can occur if you do not have permission to delete certain records.

Running Idaudit Without Program Arguments

When you pass no options to *idaudit*, the program does the following:

1. Processes an informational run
2. Presents the data in a report as follows:
 - IDs with a purge date earlier than the run date (usually the system date) and the message: "Scheduled for purging"
 - Detached ID data showing:
 - The ID number
 - The field in the record that references the ID number
 - The record where the ID number is referenced, the number of times that the ID number was found in the record, and the *delete_allow* status of the *dbfile_rec*

Note: *Idaudit* does not delete any records from the database unless specifically instructed by using one of the delete/remove options described below.

Idaudit Program Arguments

You can use several options to enable process features of *idaudit*. The following examples indicate how you can specify the options for *idaudit*:

```
Example: idaudit [-v] {-u or -i ID1 [ID2 ID3 ...] or -p [-d MM/DD/YY] }  
            idaudit [-v] [-s ID1] [-e ID2] [-f FIRST_RECORD] [-l LAST_RECORD]
```

The following lists the options and what they signify to *idaudit*.

-i (Information)

Specifies that *idaudit* search for specific ID numbers and report all occurrences of these numbers. You can use this option to locate all instances of duplicate IDs in the database.

For example, if the ID numbers 10583 and 13495 refer to the same individual, enter the following: **idaudit -i 10583 13495**

Idaudit will produce a report that provides the primary keys and their values for every record in the database where these ID numbers are found. You can examine these records and manually transfer the necessary data from one ID to another.

CAUTION: Take care in selecting what ID number to use. Please call Jenzabar before making the data transfer. Some records may require some additional changes or processing to maintain the data integrity of CX. There may also be preferred ways of doing the transfer.

Note: You can mark an ID for purging, and then later run *idaudit* using the -p option to purge the ID.

-u (Update)

Signals *idaudit* to update (delete) those records in the database that reference an ID number that is no longer in the ID record *and* have the allow_delete row of the dbfile_rec marked with a Y or y. If the allow_delete row of the dbfile_rec is marked N, or is blank, *idaudit* places the following in the report:

- The specific ID number
- The field in the record that references the ID number
- The record where the ID number was found
- The message: "NOT DELETED"

You can use the -f, -l, -s, -e options in conjunction with the -u option (ex: *idaudit -u -f alpha_rec -l beta_rec*).

CAUTION: Use this option with extreme care, and only after the general report run of the process and a full (Level 0) backup. Jenzabar recommends that you schedule an overnight run of *idaudit* when using this option.

-p (Update Purge IDs)

Deletes those IDs in the ID record that are marked for purging and that are not referenced anywhere else in the database. If *idaudit* finds an ID referenced elsewhere in the database, a "NOT PURGED" message is printed beside the particular ID number in the report. For more information on an ID with this message you can run *idaudit* with the -i option.

Note: You can use the -d, -s, -e options in conjunction with the -p option (ex: *idaudit -d 11/11/95 -p -s 100 -e 105*).

CAUTION: Entry order of options is important when you use -d in conjunction with -p. Specify -d before -p. Jenzabar recommends that you schedule an overnight run of *idaudit* when using this option.

-d (Date)

Specifies a run date. Use this option in conjunction with -p. *Idaudit* ordinarily uses the system date as the date to check the purge dates in the ID record (id_rec). You can use this option to delay the deletion of records marked for purging. Enter the date in the standard mm/dd/yy or mmdyy format.

Note: *Idaudit* purges an ID only if:

- The ID is not referenced anywhere else in the database
- The purge date is prior to the date of the process

CAUTION: The -d option is *not* used to schedule a purge to be run at a later time/date. Entry order of options is important when you use -d in conjunction -p. Specify -d before -p.

-v (Verbose)

Outputs progress messages indicating the various phases of operation as well as the records and keyed fields used in accessing the database.

Note: The verbose option prevents other processes from being initiated from the same terminal/PC during the operation of the program.

-s (Starting ID Number) and -e (Ending ID Number)

Specifies a range of ID numbers. You can use these options separately or in conjunction. Unless you specify these options, *idaudit* examines the entire ID record and considers all ID numbers as initially valid candidates for processing. You can use these options to examine portions of the ID record.

Note: When you use the -s and -e options, the report provides *purge* and *detached* data on the IDs in the specified range.

-f (First Filename) and -l (Last Filename)

Specifies a range of filenames. You can use these options separately or in conjunction. Unless you specify these options, *idaudit* searches all the files in the database during ID processing. You can use these options to examine a specified range of records.

Note: When you use the -f and -l options, the report provides *purge* data on all IDs and *detached* information on only the records in the specified range.

Running ID Audit

Processing Time

The processing time of *idaudit* can vary depending on the following:

- The choice of command line options used with *idaudit*
- The number of ID records in the database
- The total number of records in the database
- The system load

Because of the above considerations, *idaudit* could take anywhere from fifteen minutes to more than a day to finish. Jenzabar recommends that *Idaudit* not be run during the day since database activity in the ID record and in other database files during the run could produce inaccurate reports.

Note: Depending on the number of records in the file that *idaudit* is reading, it could take hours just to look through one file, and the CX database has well over 250 files to be considered.

Processing Notes

Remember the following about running *idaudit*:

- Jenzabar recommends that you run *idaudit* without command options on a quiet system. The *-s*, *-e* options will shorten the run time and output. You should then carefully review the data on the report.
- When finished searching through the entire database, *idaudit* writes a report to a file of the format, *mm-dd-yy.hhmmss*, in the *\$CARSPATH/audit/common/idaudit* directory.
- If *idaudit* discovers a problem with the ID numbers in a file, the program may not be able to finish a run. This error should only occur when many records exist in the database that reference nonexistent IDs, or when many IDs need to be purged. If the report contains no data, check for the report files in the */tmp* directory under *idaudit*.

Note: See below for more information about the *idaudit* report.

- Files exist in the database that should never be deleted. To protect these files, *idaudit* only deletes those files which have *Y* or *y* in the *allow_delete* field of the file's *dbfile_rec*. The only exception to this is the ID record. Jenzabar recommends that you contact Jenzabar before you run *idaudit* with the update option.
- If you have any questions about selecting one ID number when duplicate ID numbers exist for an individual, or any other questions about the use of *idaudit*, contact Jenzabar for further instructions.
- If a fetch error occurs, run the report again and/or attempt to determine who has the record locked. The process will complete when a fetch error occurs, but the data may be incomplete.

The Idaudit Report

Iaudit creates a report that contains information on IDs. The report's appearance is the same for most *idaudit* run options; only the `-i` option produces a report that is appreciably different.

In general, the report's features are as follows:

- The start time for the *idaudit* run appears at the beginning of the report
- The end time for the *idaudit* run appears at the end of the report
- Prints *purge record* data first, followed by *detached record* data, sorted by ID number then by record
- If the same ID appears in a detached record more than once, the report notes the total number of occurrences (e.g., (14 times)
- The report also reflects the status of the `dbfile_rec.allow_delete` field for each of the detached records by listing a (Y) or (N) after the record. You can only delete those records indicated with a y or Y.

All runs of *idaudit* create a file in the `audit/common/idaudit` directory, and four files on the `/tmp` directory. Options `-d` and `-p` create their output directly in `audit/common/idaudit` even though they also create the files in `/tmp`. *Iaudit* run without options and with options `-i` and `-u` use the files in `/tmp` to create the report before the data is deposited in the file in `audit/common/idaudit`.

Note: You can identify the *idaudit* report files in the `/tmp` directory by their name:
idaudit.#####.

The report(s) is named for the year, hour, minute, and second that the report was run, arranged in `yy, hh, mm, ss` format.

Running Idaudit With the Update Options

Running *idaudit* with either of the two update options has the potential to create major changes in the database. Jenzabar recommends that you do the following:

- Before running *idaudit* with either `-u` or `-p`, you first run *idaudit* without either of these options
- Carefully review the report to find exactly what will be deleted
- If possible, query the records that the report gives and make sure that they really should be deleted
- If it is possible, check each ID and record to insure of the validity of the report

CAUTION: Remember that when *idaudit* deletes the record, you can only recover the record by restoring the data file from a backup. Do *not* run *idaudit* with an update option until you have thoroughly examined the output of your initial run of *idaudit* without options and have run a full (level 0) backup.

The allow_delete Flag

CX contains records in the database that you should never delete. To guard against unwanted deletion, *idaudit* checks the allow_delete field in each record's Database File record (dbfile_rec). Unless the field's flag is set to Y or y, *idaudit* does not delete any records from that file.

Note: This processing feature assumes that the records, dbfile_rec and dbfield_rec, have been built and maintained on the system.

With the -p command, *idaudit* will delete ID records that have been marked for purging and that are not linked to any other records in the database, even though the allow_delete field in id_rec is set to N or n. The allow_delete flag is only meant to prevent the accidental deletion of records during a large scale update of the database, not to prevent the deliberate deletion of records that have been assigned a purge date.

Note: *Idaudit* will not delete those records with the allow_delete flag set to Y that also have a prev_name_id flag that refers to an ID no longer in existence.

Records like the Profile records (profile_rec) and the Education records (ed_rec) in the system no longer have meaning when associated with an ID record that is deleted. These records can safely have allow_delete flags of Y. Other records that have additional meaning, like the Subsidiary Account record (suba_rec) and the Course Work record (cw_rec), should have allow_delete set to N.

If you are unsure which records should and should not be deleted, please contact Jenzabar before running *idaudit* in an update mode.

Crash Recovery

If a crash occurs while running a version of *idaudit*, you need to:

1. Determine where the process stopped. Do the following:
 - If you ran *idaudit* with the -d, -p options, look in: audit/common/idaudit
 - If you ran *idaudit* with the -i, -u, or with no options, look in /tmp
2. Remove the files from both directories. If partial data is appropriate for your use of *idaudit*, you can print the output from the -d, -p options run before removing the files.

Merge ID Program

Introduction

This section describes Merge ID (*mergeid*), a program that allows users to maintain ID records by merging information from duplicate ID records into one ID. When you identify the existence of two IDs for the same person, you designate one of the IDs as the primary ID and the other as the secondary. The primary ID should be the ID that you want to retain. Mergeid will locate every occurrence of the secondary ID and merge it into the primary ID.

This section contains information about the following:

- Features
- Terms
- Modes
- Procedures

Merge Logic

The foundation of the merge process is to change the duplicate (secondary) id number to the valid (primary) ID number across all ID columns within the database. When the column holding the secondary ID is not part of an index on the table (such as `ctc_rec.corr_id`), then the logic just updates the column to the primary ID. When the column is part of a unique or logical index (such as `profile_rec.id`), then the logic is to remove the secondary row with the possibility of using the COPY_1ST option to fill in any blank or NULL columns of the primary row with the values of the secondary row before it is deleted.

To enhance performance, foreign key logic is provided. This functionality assumes that certain ID fields obtain their value only from ID fields in a specific table, and if that table does not have an entry for the ID being processed, further searching is not performed. For example, if an ID being merged does not have a `sch_rec`, then the `ed_rec.sch-id` field would not be searched for the ID.

Overview of the Process

Below is a broad overview of the steps to take in the mergeid process.

1. Define merge types and table and column level merge actions.
2. Enter one or more IDs thought to be duplicates into the Merge ID Interactive screen. This step must be done in interactive mode.
3. Run the merge preview process.
4. Preview the results in interactive mode.
5. Update the database with the results of the merge.

You access the mergeid options from the System Management: Data Dictionary Menu.

What is an ID Column?

The Merge ID process considers every table containing an ID column. It examines the database's system tables (systables and syscolumns) to find ID columns. A column is considered an ID column if it matches the following criteria:

- The type is smallint, integer, or serial, AND...
- Ends with the three characters "_id", OR...
- The column name is id_no, id_used_by, id_add_by, or subs_no, OR...
- The column name is id, prim_id, sec_id and is not part of the of the id_rec or mergeid_rec tables

ID columns meeting any of the above criteria can be specifically excluded from the Merge ID process by adding an entry in the mergefield_rec with a merge type of IGNORE..

Merge ID Features

The *mergeid* program contains the following features:

Batch and Interactive Modes

The Merge ID process runs in either interactive or batch mode. You must establish ID pairs and review merge action results interactively. However, you may perform the following steps interactively *or* by letting the process perform them in batch mode:

1. Generate merge actions
2. Perform the merge actions and update the database

Preview Merge Action Results

Running *mergeid* in interactive mode allows you to preview the results of the merge actions before choosing to update the database with the changes to the ID records. This allows you to detect data conditions that would result in an illogical data condition but would be technically accurate based on the data dictionary merge table setup.

Invoke Audit Programs When Necessary

When running *mergeid* in interactive or batch mode, you can inform the process to run *saaudit*, *daaudit* or *trans*. This will reconcile differences between totals and correct any balances in summary records that may have occurred as a result of running *mergeid*.

If the Merge ID process does not run an audit after performing a merge or if the audit fails, you can manually run *saaudit*, *daaudit*, or *trans* and reconcile the results. The *mergeid* program will send messages (online in interactive mode, by e-mail in batch mode) regarding the audits whether they were successful or not. From the message returned, you can determine whether you need to run an audit manually.

Error conditions are corrected outside the Merge ID process through the *mergeid* table setup or appropriate data modifications. For a viable audit trail, the actions taken by the Merge ID process must be accepted in their entirety.

Merge ID Terms

The following are definitions of terms and concepts used with the *mergeid* program:

Unique Index

A unique index is a database concept stating that a column, or group of columns, in a table cannot have more than one occurrence of a specific value or combination of values. For example, in the sample record below the unique index = SSN + Name. There can be only one occurrence of that exact combination. Even though the values in the two records are similar, they are not the same, and therefore, qualify as being unique.

NAME	SSN	ADD1	ADD2	STATE	ZIP
Dave Smith	540-54-9871	17 Elm St.		OH	45208
David Smith	540-54-9817	17 Oak St.	Apt. 2	OH	45208

Logical Index

A logical index is a concept whereby a column (or a series of columns) is viewed by mergeid as a unique index even though the unique index is not specified at the database level. (This usually occurs if the index was not required for normal processing against the table.) A table with a serial number would by virtue of the serial number be considered unique, so the database would technically allow duplicate values to exist. However, the ID information may need to be considered by the process and merged. You may create a logical index that causes mergeid to “overlook” the serial number and then consider the fields. For example, in the sample aid_rec below the unique index is AID# + SSN. The mergeid process would not detect any duplicate values because the aid #s are serially assigned and therefore all different.

AID #	SSN	YR	CODE	AMT
1	549-88-0734	9899	pell	800
2	549-88-0734	9899	staf	2000
3	549-88-0734	9899	pell	800
4	549-88-0743	9899	perk	400
5	549-88-0743	9899	staf	19999

However, there are duplicate records in the table. The information in the shaded columns below exists for the same person as evidenced by the SSN column. For the merge process to detect this, you would need to create a logical index which does not include the Aid# column. The logical index could be ssn+yr+code. Using that information it's obvious that two aid records exist with the same values.

AID #	SSN	YR	CODE	AMT
1	549-88-0734	9899	pell	800
2	549-88-0734	9899	staf	2000
3	549-88-0734	9899	pell	800
4	549-88-0743	9899	perk	400
5	549-88-0743	9899	staf	1999

Merge Index

The merge index is the index chosen by mergeid to merge a table. Several indexes may exist for the same table. The logical index has the highest priority and if it exists, it will be used by mergeid. A unique index has the next highest priority and will be used if it does not contain a serial field. If neither index exists for a table, then duplicate values are not considered.

Duplicate

The Merge ID process examines the value(s) of the column(s) of the primary index and searches the database for rows that match on the secondary ID. For each row found for the secondary ID, the Merge ID process searches the same table for a match on the primary ID. If the primary ID search results in a row found, a duplicate has been detected. The existence of a duplicate is the basis upon which the Merge ID process determines the actions to take.

Serial Number

A column with a serial number always contains a unique value. In other words, a serial column has a unique constraint (and usually it is indexed). The Merge ID process uses the serial column of a table (if one exists) to identify a single row to be updated. Since an index usually exists on a serial column, this approach saves time during the update phase of the merge process. This approach ensures that a single row is updated, and also saves space because only one "where" condition must be stored in the database for the audit trail. Also the audit trail will be easier to follow--a row's identity will change when its ID column is updated, but its serial number will remain the same.

Merge ID Tables and Records

Introduction

This section provides reference information about common tables and records associated with the Merge ID program. They are located in \$CARSPATH/common/schema.

Merge Type table (mergetype_table)

This table identifies the types of merge actions supported by *mergeid* and defines the parameters for the functions of the program. It controls the behavior of *mergeid* and the effect its actions have at the table and column level. Merge types define the actions that are performed on tables and records when you run *mergeid*. They are contained in the Merge Type table (mergetype_table). The following is the mergetype_table PERFORM screen.

```
PERFORM:  Query  Next  Previous  View  Add  Update  Remove  Table  Screen  ...
Searches the active database table.                ** 3: mergefield_rec table**

                                MERGE TYPE TABLE
Merge Type Code..... COPY_1ST
Description..... Copy Data, then Discard
Ignore Table/Column..... N
Logical Index..... N
Copy First..... Y
Special Algorithm..... N
Audit to Run..... N
Applicable to Table..... Y
Applicable to Column..... Y
=====
                                TABLE-LEVEL MERGE ACTIONS
Table Name..... adjust_table
Merge Type Code..... IGNORE      Do Not Merge
=====
                                COLUMN-LEVEL MERGE ACTIONS
Table Name.....[accommod_rec      ]
Column Name.....[accommod          ]
Merge Type Code.....[LOGICAL ] Include in Logical Index

109 row(s) found
```

The mergetype_table screen is divided into three main sections:

- Merge Type Table
- Table-Level Merge Actions
- Column-Level Merge Actions

Merge Type Table Section

The Merge Type Table section lists the seven merge types included with the *mergeid* program. The Merge Type Code and description are listed first, followed by seven yes or no qualifiers which further define each merge type and its behavior. The *mergeid* program supports the following seven merge types:

COPY_1ST

The COPY_1ST merge type controls whether or not a value from a column in the secondary ID will overwrite a null value in the corresponding column in the primary ID. For example, in the sample pair below the COPY_1ST merge action would cause the null in the ADD 2 column for Dave Smith to be overwritten by the Apt. 2 value from David Smith.

NAME	SSN	ADD1	ADD2	STATE	ZIP
Dave Smith	540-54-9871	17 Elm St.		OH	45208
David Smith	540-54-9817	17 Oak St.	Apt. 2	OH	45208

DAAUDIT

The DAAUDIT merge type stipulates that the Donor Accounting Audit program be run for the specified tables after the merge has completed.

FINANCE

The FINANCE merge type controls how mergeid deals with financial records. The Finance merge action is a special algorithm created especially for dealing with financial records.

IGNORE

The IGNORE merge type specifies certain columns within tables that will be ignored by the merge process.

LOGICAL

The LOGICAL merge type defines the columns that make up the logical index of a table.

RUNTRANS

The RUNTRANS merge type stipulates that the Run Transcript Audit program be run for the specified tables after the merge has completed.

SAAUDIT

The SAAUDIT merge type stipulates that the Subsidiary Audit program be run for the specified tables after the merge has completed.

Merge record (merge_rec)

This table tracks the primary key of each row to be updated or deleted by *mergeid*. It serves as an audit trail.

Merge Details record (mergedtl_rec)

This table tracks columns of a row to be updated as identified in the merge_rec. It also stores every non-null column value that has been deleted by the process. It serves as an audit trail.

Merge Field record (mergefield_rec)

This table tracks columns that require exceptional processing by *mergeid*, such as excluding particular fields as ID fields, and determining logical and foreign keys.

Merge File record (mergefile_rec)

This table tracks tables that require exceptional processing by *mergeid*, such as ignoring specific tables, determining foreign keys, and running audits.

Merge ID record (mergeid_rec)

This table tracks the primary and secondary IDs to be merged by *mergeid*.

Merge Detail Blob Table (mergedtl_blob)

This table tracks large binary or text columns (blobs) that were deleted. Rather than overwriting a blob column through an update process, *mergeid* reports the blob column that would have been updated. The user can then determine which picture, photo, or text to replace.

Configuration Macro

Permissions to update live data using the Merge ID process are restricted. You can use the Configuration table entry `MERGE_ID_UPDATE_GROUP` or create a new group to determine user privileges. The `MERGE_ID_UPDATE_GROUP` table entry controls whether or not the update option displays on the menu. The financial tables have additional restrictions and grant delete permissions only to the database super user. The Merge ID process must be run with `set-user-id` enabled.

Running Merge ID in Interactive Mode

Introduction

Run *mergeid* in interactive mode to establish ID pairs and review merge action results before actually updating the database with the new information. You use the following screens in interactive mode:

- Merge ID Interactive screen
- Merge ID List screen
- Merge Table List window
- Expanded Merge Item window

The following sections describe these screens.

Entering ID Pairs for Merge Processing

The steps below describe the procedure for entering ID pairs into the Merge ID Interactive screen for merge processing.

1. From the System Management: Data Dictionary main menu select Merge ID--Interactive. The Merge ID--Interactive parameter screen will appear asking you if you would like the appropriate audit process to be run when you run the merge process.
2. Enter **Yes** or **No** and select **Finish**. The Merge ID--Interactive screen appears.
3. While in Query mode, do one of the following:
 - If you know the Primary ID, enter it now and the corresponding Secondary ID will appear on the Merge ID screen
 - Select <CTRL-T> to access the ID Lookup screen and query an ID
 - Enter 0 to view the current Merge ID list of qualifying IDs
4. When you have reached the Merge ID results screen using one of the above methods, select **Add** from the ring menu. The Merge ID--Interactive screen will appear in Add mode:

```
ADD:  ESCAPE finish.  CTRL-C cancel.

                                MERGE ID--INTERACTIVE                                0 of 0
Primary ID.....[0              ]
Secondary ID.... 0
Name Prior to Merge.....

Enter ID or zero (0) to select all qualifying ID#'s. Use CTRL-T for table lookup
```

5. Enter the Primary ID in the Primary ID field and press **Enter**.
6. Enter the Secondary ID in the Secondary ID field and select **Finish**.

If you need to interrupt your *mergeid* process, set the Status to Hold for the IDs you have already processed until you are ready to perform the merge with update.

Merge ID Interactive Screen

Introduction

You use the Merge ID Interactive screen to enter the primary and secondary IDs for processing.

The following is an example of the Merge ID Interactive entry screen:

```
MERGE ID: Query Add Next Previous Id-list List eXpand ...
          MERGE ID--INTERACTIVE                               1 of 99
Primary ID..... 1253395   Babbette, Borris (MERGED:112286)
Secondary ID... 1258449   Babbette, Borris                     ON HOLD
Name Prior to Merge.....

UPDATE audin_rec
SET   id           = 1253395
WHERE id           = 1258449
AND   subaud       = ELECU93

Query an existing pair of IDs to be merged.
```

Data Displayed on the Screen

The interactive Merge ID Interactive screen in update mode displays the following two main areas of information:

Entry Area

The first area on the screen is the entry area where you enter the IDs. There are two principle fields in the entry area:

Primary ID

In this field, enter the ID that you want to save and into which you want to merge all column information. All column values (that are not blank or null) in records with the primary ID will override column values in records with the secondary ID. If a column has a blank or null value, the value from the secondary ID will be used if the merge type is COPY_1ST.

Secondary ID

In this field, enter the ID whose records will be permanently removed or merged into the primary ID.

Results Area

The second area on the screen is the results area. The results of a merge appear here as a set of SQL statements describing the changes made by the merge. These results reflect how records will be amended when you choose to update the database.

Example: In the example below the process will update the ctc_rec by setting the ID to 1253395 in the record that it found to have a ctc_no of 1153851.

```
UPDATE ctc_rec
SET id = 1253395
WHERE ctc_no = 1153851
```

Commands on the Merge ID Interactive Screen

The following are the commands that appear on the Merge ID Interactive screen and their descriptions:

Add

Displays the Merge ID Interactive screen in Add-mode.

Expand

Displays the Expanded Merge Item window that lists records with merge activity. It includes complete values for each record of the merge process for the primary and secondary IDs.

Id - List

Displays the Merge ID List screen that lists ID pairs entered into the merge id entry screen. It also lists the merge status of the IDs.

List

Displays the Merge Table List window that presents a summary of records that have been updated and deleted by the active merge.

Next

Displays the next merge action results for the current pair of IDs.

Previous

Displays the previous merge action results for the current pair of IDs.

Query

Displays the Merge ID Interactive screen in Query mode which allows you to query and locate a new pair of IDs. A query on 0 (zero) will bring up the Merge ID List screen.

Merge ID List Screen

Introduction

The Merge ID List screen displays a list of the ID pairs that have been entered in the Merge ID Interactive screen and their statuses. The value in the config_table for MERGE_ID_DISPLAY_AGE determines how long IDs appear on this list. Below is an example of the screen:

```
Select Letter Next to Choice.  CTRL-C cancel.                CTRL-F forward.  CTRL-B back.
-----
                                MERGE ID LIST
-----
  Prim ID  Primary Name                Sec ID  Secondary Name                S
-----
a. 112286  Babbette, Borris                    1253395  Babbette, Borris (MERGED: D
b. 1253395  Babbette, Borris (MERGED: 1258449  Babbette, Borris            H
c. 1266990  Boyd, John T.                        1266964  Jackson, Hailey (MERGED:1 D
d. 20270    Litvee, Deborah L. A.                20250    Litvee, Duane T.            Q
e. 20250    Litvee, Duane T.                      20205    Litvee, Dwayne T.          Q
f. 23317    SMITH ,DICK                          23318    SMITH (MERGED:23317)       D
g. 22380    Costa, John P.                       22420    Costa, John (MERGED:       D
h. 12345    Doe, Jane                             12346    Robinette, Duane Lee       V
i. 20339    Evilsizor, Jeremy R.                 20225    Evilsizor, Jerem (MERGED: D
j. 20202    Evilsizor, Michele B.               48350    Evilsizor, Dougl (MERGED: D
k. 1258239  Krekeler, Bill A                     1258240  Krekeler, Bill (MERGED:    D
l. 1258251  Krekeler, William G.                 1258256  Krekeler, William H.       Q
m. 1258456  Krekeler, William M.                 1266941  Krekeler, William N.       Q
n. 48353    Moore, Kelly R.                      1242932  Moore, Kelly (MERGED:48353 D
-----
```

Fields on the Merge ID List Screen

Below is a list of the fields on the Merge ID List screen and their explanations:

Prim ID

The ID that has been entered as the primary ID into which all duplicate information will be or has been merged (depending on the status at this point).

Primary Name

The name associated with the Prim ID.

Sec ID

The secondary ID that will be or has been marked as not valid and whose information will be or has been merged into the primary ID.

Secondary Name

The name associated with the secondary ID.

S

A code identifying the status of each entry.

Merge Table List Window

Introduction

The Merge Table List window displays a summary of records that were updated and deleted in the merge process. It displays the record name and the number of records that were updated or deleted.

MERGE TABLE LIST		
Table Name	#UPD	#DEL
a. aa_rec	2	0
b. acadsum_rec	1	0
c. aid_rec	10	0
d. ctc_rec	26	0
e. edepell_rec	2	0
f. faedit_rec	1	0
g. fana_rec	2	0
h. faneed_rec	3	0
i. gle_rec	8	0
j. hold_rec	2	0
k. naf_rec	3	0
l. profile_rec	0	1
m. prog_enr_rec	0	1
n. prsm_id_rec	1	0

Column Descriptions

The following lists the columns on the Merge Table List window:

Table Name

The name of the table or record being reported.

#UPD

The number of rows that were updated by the merge for the primary ID.

#DEL

The number of rows that were deleted by the merge for the secondary ID.

Expanded Merge Item Window

Introduction

The Expanded Merge Item window is a snapshot of the record and its current values. If you view it before you update the database, the values will display as they exist before the merge. After the merge has been processed, you cannot view this window.

To print the contents of this screen, press **<CTRL-N>**.

The following is an example of the Expanded Merge Item window.

EXPANDED MERGE ITEM		
site_rec	Primary Row	Secondary Row
site_no	N/A	1647
id	N/A	112286
beg_date	N/A	05/02/1996
site	N/A	CARS
home		

Fields on the Expanded Merge Item Window

The following are the fields on the Expanded Merge Item window and their descriptions:

<record name>_Rec

The active record whose columns have experienced merge activity.

Primary Row

The values contained in each row associated with the Primary ID.

Secondary Row

The values contained in each row associated with the Secondary ID.

Running Merge ID in Batch Mode

To run *mergeid* in batch mode, go to the System Management menu. Select Data Dictionary, and then select the Merge ID – Batch option. You can run batch mode for an individual pair of IDs or all pairs with a status of Q. You can preview the merge results in interactive mode prior to updating the database by setting the Update option to N.

When batch processing completes, *mergeid* sends an e-mail containing any error messages, the number of actions saved, and the process command line.

Database Administration Program

Introduction

The Database Administration program (*dbadmin*) maintains user access and permissions. You can do the following using *dbadmin*:

- Add and remove user logins
- Update user login permissions and connection levels to CX
- View users' permissions to database tables
- View a table's permission settings for groups and users
- View users' permissions to stored procedures
- View a stored procedure's permission settings for groups and users
- Run audits to locate and correct problems with database tables

Note: Jenzabar designed *dbadmin* to be a fast alternative to the process of adding a user where you must run a *make* build on every table just to add one user. If you need to add 12 or more new users, using *make* processor may be a faster alternative.

Program Arguments

You can use several options to execute *dbadmin* from the shell. The following example shows how you can specify the options for *dbadmin*:

Example: `dbadmin [-d database] [-s output] [-a audit] [-z]`

The following lists the options and what they signify to *dbadmin*.

-d (database)

The name of an Informix database

Example: `dbadmin -d train`

Note: The above example selects a database named *train*. To get full access to *dbadmin* functions, the user running *dbadmin* must have DBA access to that database. Otherwise, a restricted subset of functions is allowed.

Note: *Dbadmin* selects the default database from the environment variable, `CARSDB`.

-s (output)

Display SQL statements as they are executed [Y/N]

-a (audit)

Run the specified audit script

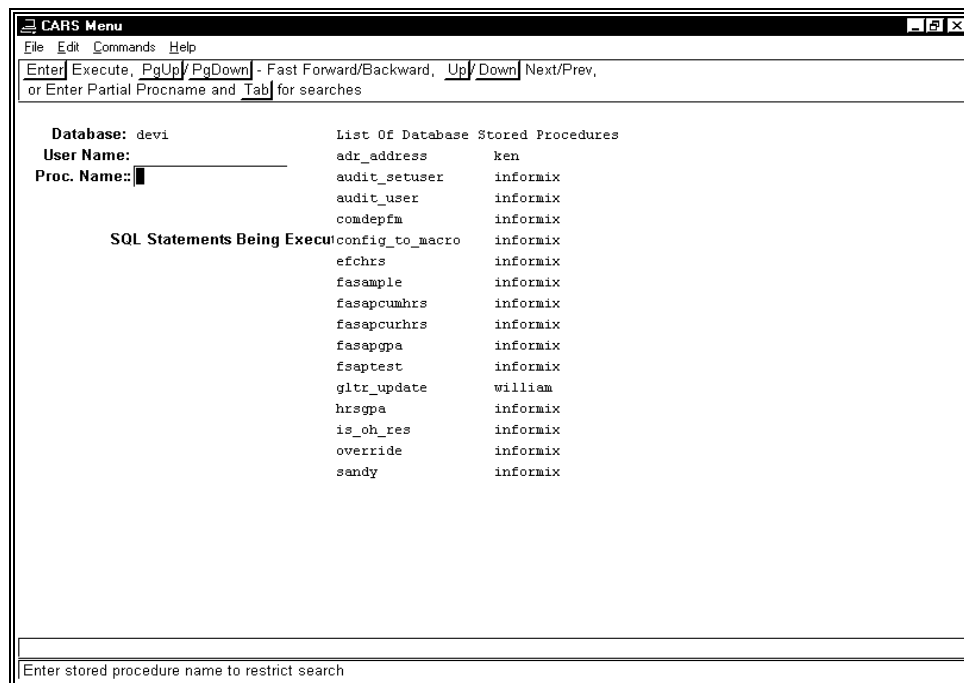
-z

Allow *dbadmin* to run as non-informix user

Dbadmin Screen

The *dbadmin* screen's User Name and Table Name fields perform partial name lookups on names. For example, if you want to choose the `stu_serv_rec` table, you can enter **stu** in the table name field., and select **Finish**. *Dbadmin* locates the first table that begins with `stu`. You can then move up and down the list by pressing the up and down arrow keys.

The following is an example of the *dbadmin* screen after you select **Permissions For Procedures** in the Procedures submenu. The screen displays a list of users and their permissions for the stored procedure displayed in the Proc. Name field.



Menu Options

This section describes the menu options that you use to initiate the features of *dbadmin*.

Note: You access the menu options from the Commands menu.

Database

Allows you to select a database. When you select Database, the cursor advances to the Database field. You can specify the database name or select **Lookup** to view a list of database names.

Note: *Dbadmin* selects the default database from the environment variable, `CARSDB`.

Users

Activates a submenu. This submenu is only available to DBA users of a database. The submenu options are described below.

Add User

Adds a new user to the database. The user will be asked to enter a login name. The login name must exist in UNIX file `/etc/passwd` and the login name must not exist in the database table `sysusers`.

Note: *Dbadmin* prompts for the three access level (Connect, Resource or DBA) to grant to the user. Then, the program prompts whether or not to grant permissions for all the tables on the database.

See *The GRANT statement* in Section 2 of the *Informix SQL Reference Manual* or more information about the access levels of Connect, Resource and DBA.

Remove User

Removes a user from the database. If a user leaves the school or is transferred to some other department where that user does not use CX, the CX system coordinator should disable the login and the database access for security. This works exactly opposite to the Add User option discussed earlier. First the connection will be revoked, then the operator will be prompted if all table permissions should be revoked.

Update Perms

Re-grants table access permissions for a user. This is designed to do the same tasks as the second half of Add User. This presumes that a user has already been granted access to the database, and you want to re-grant table access for that user.

Note: *Dbadmin* checks the */etc/group* file when updating user permissions. If you remove a user from a group, run this option to revoke any table permissions that are based solely on that group.

Modify Connect

Changes the access level for a user. For example, if Joe has Connect level access and needed Resource level access, this is the option to use. *Dbadmin* will ask which user need to be changed then prompt for which of the three access levels (Connect, Resource, DBA) the user should be granted.

List Users

Displays a detail window of users who have access to the current database. The window shows the login name and the access level.

Tables

Activates a submenu. The options are functions that relate to table permissions. The submenu options are described below.

Permissions For Table

Examines the real privileges that a specific user has for a specific table. When you select Permissions For Table, the cursor advances to the Table Name field. Enter the following:

- The table name
- The user name

The output also includes entries for the public user since a user also has privileges through public permissions. You can scroll through the output with Up and Down buttons.

View Systabperm Entries

Displays a table's group and individual user permissions. When you select View Systabperm Entries, the cursor advances to the Table Name field. Enter the table name.

Note: The information comes from the same tables used by *dbadmin* to grant permissions to new users. You can use this option to verify that *make* is correctly running on schemas. You can scroll through the output with Up and Down buttons.

Procedures

Activates a submenu. The options are functions that relate to stored procedure permissions. The submenu options are described below.

Permissions For Procedures

Examines the real privileges that a specific user has for a stored procedure. When you select Permissions For Procedures, the cursor advances to the Proc. Name field. Enter the following:

- The procedure name

- The user name

You can scroll through the output with Up and Down buttons.

View Sysprocperm Entries

Displays a stored procedure's group and individual user permissions. When you select View Sysprocperm Entries, the cursor advances to the Proc. Name field. Enter the procedure name. You can scroll through the output with Up and Down buttons.

Audits

Starts the Audit process. See below for more information.

Options

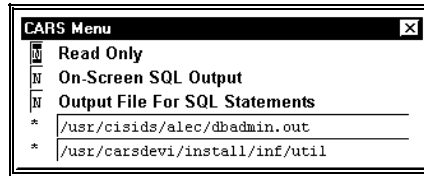
Displays a pop-up window for you to set various switches and define file locations. See below for more information.

Options Pop-Up Window

The options pop-up window allows you to set various switches and define file locations.

Note: The defaults for these options come from the screen located in `modules/util/progscr/dbadmin/options`.

The following is an example of the Options pop-up window.



Options Pop-Up Window Fields

The following are the fields in the Options pop-up window.

Read Only

Specifies if *dbadmin* is allowed to modify the database.

Note: Users learning to use *dbadmin* should set this field to Y.

On-Screen Output

When set to Y, displays SQL statements, which are sent to the database, on the screen.

Setting the Output File

When set to Y, sends the SQL statements, which are sent to the database, to the file name specified in the next field.

First Unlabeled Field

Use to specify an output file name for SQL statements sent to the database.

Second Unlabeled Field

Use to specify the default path to use to search for *dbadmin* audit scripts.

Audit Processing

Audits locate and correct tables that might have invalid values or extra rows and cleans up `/tmp` directories. This option is only available to DBAs. The following occurs in an audit process:

1. The audit process searches for scripts in its search path and displays the list in a dialog box.
2. You select the audit desired and *dbadmin* begins searching for rows that meet the audit criteria.
3. *Dbadmin* displays the rows selected by the query, if any, in the detail window for your

review.

4. After you review the data returned by the query, you select **Finish**.
5. *Dbadmin* notifies you whether or not to repair the records. Depending on the audit, you might be prompted to review each record in the query.
6. If you respond yes to repair the records. The repair operates on each row of the query, updating or deleting items that the query detected.

Audit Scripts

An Audit script contains five parts. The parts are described below.

Audit Keyword

Informs *dbadmin* that the script is an audit script. This prevents *dbadmin* from getting confused by real Informers. This part must be the first keyword. The other parts can be in any order.

Name of the audit

The name of the audit. It is displayed in the selection menu box.

Query Statement

A single, complete SQL select statement that retrieves information about corrupted data.

Repair Statement

A complete SQL statement that corrects one row. This statement may be called for every row returned by the select statement. String substitution is performed on the statement before execution. Values from selected columns are substituted where column names are enclosed in square brackets.

For example, the following repair statement:

- update my_table set valid = "Y"
 - where tabid = [tabid] and username = "[username]";
1. A query returns the following rows:
 - select * from my_table where valid = "N";
 - tabid username perms valid
 - 203 |eric |su-id--|N
 - 203 |ken |su-id--|N
 - 213 |john |su-----|N
 2. The repair statement expands to the following:
 - update my_table set valid = "Y"
 - where tabid = 203 and username = "eric ";
 - update my_table set valid = "Y"
 - where tabid = 203 and username = "ken ";
 - update my_table set valid = "Y"
 - where tabid = 213 and username = "john ";

Note: If you use the M option under the prompt section, the repair sends a message that *dbadmin* cannot perform a repair.

Prompt

Specifies how to handle the repair. You can specify one of four values, which are:

- Y (indicates that you should be prompted on every cycle)
- N (indicates to not notify you whether or not the records need to be repaired)
- U (allows you to choose Y or N at run time)
- M (displays informative messages from the repair section when no repair is possible)

Example Audit Script

Audit scripts are created by default in \$CARSPATH/modules/util/informers. The following is a sample audit script that looks for systabperm rows that no longer refer to a valid table.

```
audit
name = 'Extra Systabperm Entries'
query = '
select username, usertype, tabid from
    systabperm
where
    tabid not in (select tabid from systables) or
    tabid < 100;
,
repair = '
delete from systabperm where
    username = "[username]" and
    usertype = "[usertype]" and
    tabid = [tabid];
,
prompt = 'U'
Parts of an audit script
```

Additional Table

Dbadmin creates the following table based on /etc/passwd.

```
create temp table passwd
(
    username    char(8),
    uid         int,
    gid         int
) with no log;
```


Schedule Entry Program

Introduction

To create and maintain Schedule records for individuals or entities, you can use the Schedule Entry program (*schdentry*). Like other entry programs, it retrieves and maintains information by ID number, and enables users to track scheduled activities (e.g., admissions counselors' trips to high schools or the registrar's appointments with students).

Similar to other CX entry library programs, Schedule Entry displays a query screen from which the user accesses the ID lookup feature. Typically, Schedule Entry does not permit the entry of ID records, but with appropriate permissions it is possible for users to add IDs.

Windows Available in Schedule Entry

Schedule Entry offers access to many of the same records as other entry programs, and enables users to view or update record information through detail windows. Detail windows available from Schedule Entry are:

- Address runcodes
- Contacts
- First relation
- Second relation
- Schedule activities
- Other address
- Other name

Of these windows, the Schedule Activities window is unique to Schedule Entry. It displays the contents of the Schedule record (*schd_rec*).

Records Used in Schedule Entry

Records used in Schedule Entry are:

- *aa_rec*
- *addree_rec*
- *adr_rec*
- *ctc_rec*
- *profile_rec*
- *relation_rec*
- *relsec_rec*
- *schd_rec*
- *userid_rec*

Setup Issues for Schedule Entry

Consider the following table setup and permissions issues for implementing Schedule Entry:

- If you enable users to add id recs through Schedule Entry, you must have already set up the standard common tables (i.e., *st_table*, *title_table*, *ctry_table*, *zip_table*, *aa_table*).
- To use Schedule Entry most effectively, you must also establish the tables used by the scroll screens; none, however, relate directly to Schedule Entry.
- To add records through Schedule Entry, a user must be a member of one of these groups:
 - student
 - admissions
 - development
 - carsprog
- Only members of the carsprog group can delete Schedule records.
- Only members of either the carsprog or addid group can add ID records.

Sortpage Program

Introduction

Jenzabar created the Sortpage program (*sortpage*) to handle the sorting of output when records get out of their original order. The following example explains a use for *sortpage*:

- The ACE report sorts the output by zip code from the zip code value in the ID record (id_rec)
- The Address program (ADR) takes the ACE report output and inserts the correct address for an individual, which may not be the address in the associated ID Record (id_rec).
- The output, therefore, is no longer sorted in the original zip code order.

Sortpage uses the UNIX sort utility and values supplied by ACE to do the sorting.

Macros You Must Set

You must set the following macros to allow for placing the *sortpage* script into any ACE report used for letter/label production.

SRT_DEFINE

Defines the variables needed for the ACE report to function using *sortpage*. To use *sortpage*, you must place SRT_DEFINE in the define section of ACE (See Informix manual on ACE reporting).

Note: The macros are structured so that the SRT_DEFINE macro can be included at the top, even though *sortpage* will not be used. The key macro for inclusion is the following macro, SRT_SORT_BY.

The following is the expanded version of SRT_DEFINE:

```
function _dbtype
function _dbsize
variable _srt_type      type integer
variable _srt_long      type long
variable _srt_double    type double
```

SRT_SORT_BY

Used to tell *sortpage* what fields are to be used in sorting. You can use SRT_SORT_BY in addition to or in place of the ACE sort clause. The format of the macro is

SRT_SORT_BY(VALUE1 [descending],...,VALUEn)

Note: The optional word 'descending' reverses the order, ascending order is the default. An example of a SRT_SORT_BY clause would be:

SRT_SORT_BY(ADR_ZIP_VALUE,ADR_NAME_VALUE)

The macro parameters, ADR_ZIP_VALUE and ADR_NAME_VALUE are keywords to the SRT_SORT_BY macro. These are replaced by ADR commands to include the named values from the output of ADR on the ID number processed. For example, the record being processed by ADR has the ID number of 100, and the zip code from the ID record is 45056. ADR checks and finds an alternate address for that ID and replaces the address with a zip code of 89000. The macro ADR_ZIP_VALUE will pass the zip code supplied by ADR, 89000, instead of the home address zip code, 45056.

Note: The SRT_SORT_BY macro expands to nothing in ACE, but rather is used later by the SRT_HEADER and SRT_VALUES macros.

The *sortpage* macros within ACE are set up to automatically include everything needed by *sortpage* if the macro SRT_SORT_BY is found in the ACE report.

Note that even if the macro is commented out, the sort information will still be included since the macro pre-processor does not check for ACE comments.

SRT_HEADER

Automatically included with the LTR macros, expands to create the necessary first page header information needed by *sortpage*. The number of sort fields and length are defined by this macro. This also begins the sorting process, which continues when the program reaches the SRT_END macro.

You can subsort within one ACE report output by using multiple combinations of SRT_HEADER and SRT_END. The actual ACE commands generated are as follows:

```
print "$$$", "fields", ":";
print "10", "", "";
print ",";
print "32", "", "";
print
print "$$$", "names", ":";
print "zip", "", "";
print ",";
print "name", "", "";
print
```

The above produces the following output:

```
$$$fields:10,32
$$$names:zip,name
```

SRT_VALUES

Expands to put the actual values to pass to *sortpage*. The macro expands to the following ACE commands (using our zip,name sort example):

```
print "$$$", "values", ":";
print "&&&", "value", ":", "zip";
print
print "&&&", "value", ":", "name";
print
```

The above produces the following output:

```
$$$values:
&&&value:zip
&&&value:name
```

Note: The '&&&value:' is interpreted by the ADR program and replaced by the actual values passed back from ADR. Thus, from the example, the output passed to *sortpage* would look like:

```
$$$values:
89000
Smith, John R.
```

SRT_SORTBREAK

Instructs *sortpage* that this is the end of a group of data. The data are sorted then printed. Text from this point until the next values command is copied directly to stdout. The actual ace commands generated are:

```
print "$$$", "sortbreak", ":"
```

SRT_END

Automatically included by LTR macros, marks the ending point of sorting. The actual ace commands generated are:

```
print "$$$", "end", ":"
```

The above produces the following output:

```
$$$end:
```

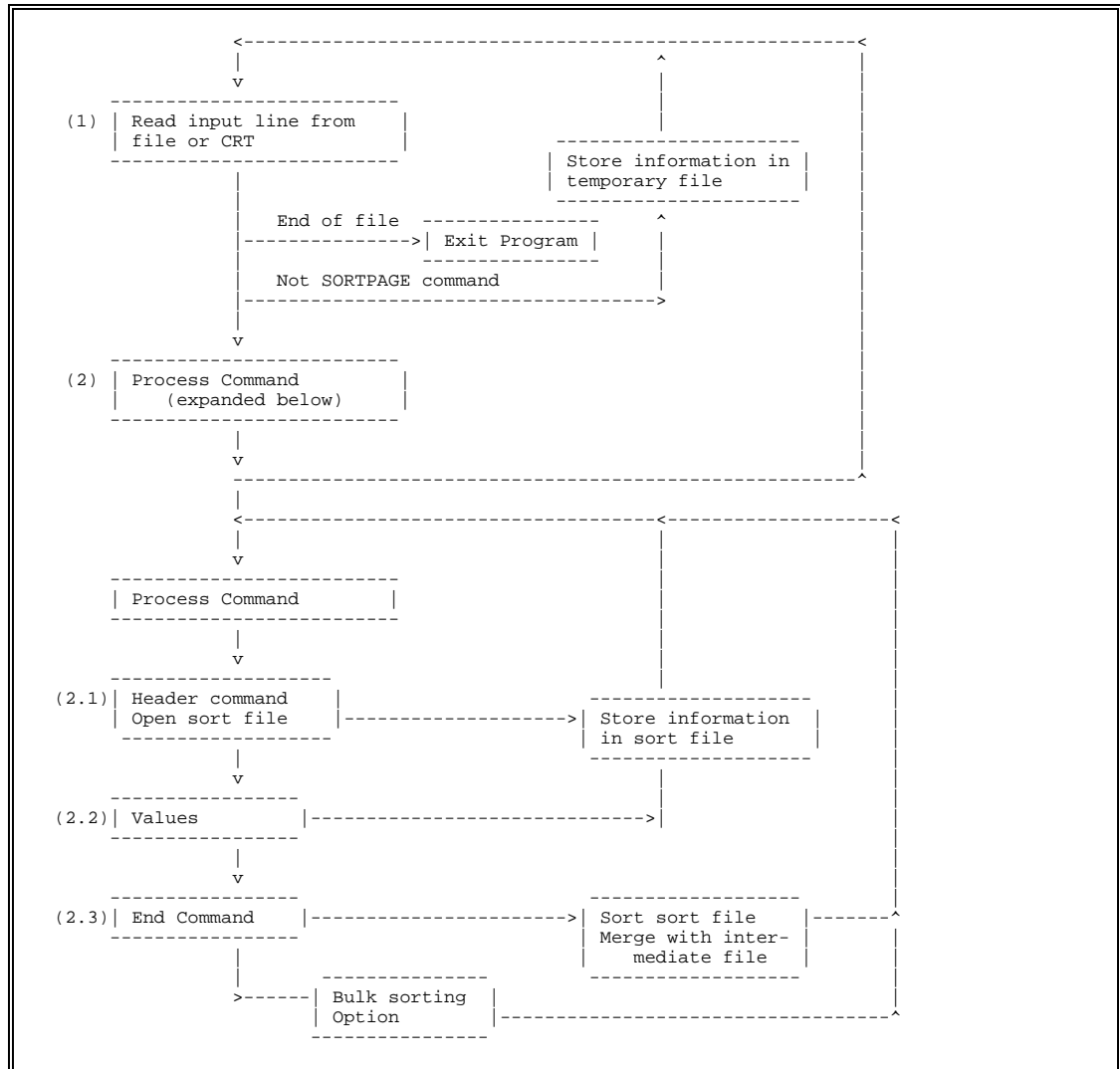
Sample ACE Report

The following is an example ACE report that uses *sortpage* macros.

```
define
.
.
SRT_DEFINE
end
.
read into a
.
.
end
SRT_SORT_BY(ADR_ZIP_VALUE,ADR_NAME_VALUE)
format -----
first page header -----|
. |
. |
SRT_HEADER | -----> Combined into one
on every record | macro for ADR
. |
SRT_VALUES | LTB_FORMAT
. |
on last record -----|
. |
SRT_END | -----> Combined into macro
. |
. |
LTB_LAST_REC
end -----
```

Program Flow

The following diagram summarizes the logic flow followed by *sortpage*.



Sortpage Processing

The system passes information to *sortpage* until an End of File is reached, at which time the intermediate file is closed. The file is not sorted at this time unless an End command had already been reached.

- If no End command was passed, *sortpage*, does not perform sorting.
- If the line is not a *sortpage* command (*sortpage* command must be on a separate line than the other data lines), *sortpage* places the line in a temporary file for later merging with sorted information.

Sortpage Process Commands

The following are the commands executed by the *sortpage* command.

Process Command

Executed if a *sortpage* command is found.

Header Command

Opens the sort file and defines the lengths for each sort field.

Values Command

Defines the value for each sort field for the record, and places the value in the sort field.

End Command

Sorts the sort file and merges the temporary file (holding the non-*sortpage* input) in with the sorted information, which is passed to the output file or device.

Note: If the End of File has not been reached, Step 1 is continued until all lines have been read and processed.

Bulk Mailing Mode

Sortpage has a bulk mailing sorting mode, based on Postal Service guidelines for third class mail. To use this mode, *sortpage* must have zip and state specified in the sort key as ascending keys. You can also add additional keys, (e.g., the resource) either before or after state and zip.

Sortpage will create packages of ten pieces or more of data that have the same:

- Five digits
- Three digits
- State
- Leftover values

Note: The system sends mail to the user with a report of the size, type and contents of each package.

The following is an example of bulk mail sorting output.

```
Sample Mail File:
Information:
 19 5-Digit      'D'-Package  01002
 20 5-Digit      'D'-Package  01201
 44 3-Digit      '3'-Package  010xx
 17 3-Digit      '3'-Package  012xx
 16 3-Digit      '3'-Package  013xx
 11 State        'S'-Package  ME
 11 State        'S'-Package  CT
 9  Mix State    'MS'-Package
```

Setting Up Bulk Mail Sorting

You can set up bulk mail sorting using one of the following methods.

- Add the -b option to the *sortpage* command line
- Add the macro SRT_BULKMODE before or just after the SRT_HEADER macro.

Note: The system automatically uses the SRT_BULKMODE if the tctc_bulkmode field contains any one of {S, P, B, Y or X}.

Program Error Messages

Sortpage sends error messages through user mail. The following error messages can occur while running *sortpage*:

- “Cannot open temporary file.”
- “Cannot open sort file.”
- “Could not sort input”
- “Fields exceed Names”
- “Names exceed Fields”
- “The number of fields in the fields record is different from the names record.”
- “The temporary file being used for intermediate storage of all the lines of input could not be opened (created). Probably due to permissions.”
- “The temporary file being used for sorting, i.e., the file having the fields and values in it, could not be opened (created). Probably due to permissions.”
- “The UNIX sort command failed for some reason. Possibly the lengths were incorrect or the wrong data type was specified.”

Crash Recovery

Since *sortpage* does not access the database directly, but rather files storing data read from these files, the database will remain intact if the system crashes while running *sortpage*. The final sorted output file may not have finished, so the *sortpage* option may be restarted from the beginning. The input file passed to *sortpage* is never updated; all information is passed to temporary files for subsequent sorting.

SECTION 9 – JENZABAR CX ENTRY LIBRARY

Overview

Introduction

This section describes the uses of the Entry Library program. The design of CX Entry Library (libentry) program provides the ability to customize entry programs to fit your institution's individual requirements. CX products perform data retrieval and database maintenance using the entry library. Data retrieval and database management is controlled for each individual entry program through the use of a set of arrays that describe the following for the library:

- What tables are to be used
- What the relationships are between the tables
- What tables have detail windows associated with them
- What the update processing order of the tables should be

You make the above specifications to the Entry Library program's definition file, the *def.c* file. In addition, you can specify special processing conditions for a table or column within the array definitions. In the *def.c* file, you can also:

- List the tables and detail windows (or scroll screens) accessible by the program
- Specify table relationships and other special processing considerations

Adding Tables for Use in Entry Library Programs

Introduction

By modifying the `def.c` file for an Entry Library program, you can do the following with tables:

- Add a table
- Add a scroll table (detail table)
- Control the display of table data

Adding a Table

To specify a table for use in an Entry Library program, you must add a line to the `filename` array of the program's `def.c` file. When you specify a table in the `filename` array, remember the following:

- For each component column in the specified index, specify a line in the common fields array
- You can add additional lines in the `check fields` array to provide special column processing

Adding a Detail Table

To specify a detail window in an Entry Library program, you must add the following to the program's `def.c` file:

- Add the detail table to the `filename` array, if the table is not already present
- Add the detail table to a line to the `scrollfiles` array

Displaying Specific Detail Table Rows on a Form

While you can only process one detail table row at a time on a form, you can display multiple rows through the use of table lookups. You can control the display of rows by using the `match` attribute in the screen definition file. By default, the first detail row loaded from the database displays in the form.

For example, for a form concerned with processing undergraduates, specify a `match` value of `UNDG` for the program column. This specification causes one of the following in the form screen:

- Display of an undergraduate Program Enrollment record
- Creation and display of a new defaulted undergraduate Program Enrollment record

Adding a Lookup Table

You specify table lookups in the screen file for the entry program form. The CX screen package (SRC) can perform table lookups directly against the database. For more information on specifying table lookups, see the following section of this document: *Creating Screen and Form Definition Files*.

Limiting the Number of Detail Tables in a Form

To limit the number of detail windows accessed in an entry program form, do the following in the `attributes` section of the form:

- Add a new field: **scroll_files_limited_to:**
- Set the text attribute to the comma or space separated list of detail windows that should be accessible.

The following is an example:

```
scroll_files_limited_to: optional,  
text = "aa, ctc, rel, relsec";
```

Entry Library Def.c Macros

Introduction

The def.c file for an Entry Library program contains macros for specifying default values for the associated entry program. You specify the macro values in the *filename* array of the def.c file.

Def.c Macro Definitions

The following macros are specified in the def.c file.

Note: Each of the macros are used in the *prog_param* array. You can override the macros with a command line parameter.

SCREEN_PATH

Specifies the first default path to be searched under \$CARSPATH/install/scr for screens referenced by the program. If screens are not found in the SCREEN_PATH then the program will search in \$CARSPATH/install/scr/Lib/libentry for the screen.

DEFAULT_MENU

Specifies the name of the default menu screen.

IDTYPE_SCREEN

Specifies the location of the default Idtype window, invoked with the ID-Type command from any ID field.

MENU_TITLE

Specifies the default title on the main ring-menu.

Example of Macros

The following is an example specification of macros in a def.c file.

```
#include "mac.h"
#define SCREEN_PATH "admit/admentry/"
#define DEFAULT_MENU "admmenu"
#define IDTYPE_SCREEN "Lib/libids/type"
#define MENU_TITLE "ADMISSIONS ENTRY"
```

Entry Library Def.c Variables

Introduction

The def.c file for an Entry Library contains variables for specifying program parameters for the associated entry program. You specify the variable values in the *Local variables and program parameters* array of the def.c file. You must define and initialize variables in the variables section of the array before the variables can be used in the program parameters (*prog_param*) section, the section of the file where program parameters are defined. The program parameters section of the array makes variables available for use by the entry library.

Note: The actual variable names used are not important; however, the labels associated with these variables in the *Local variables and program parameters* array are important to the processing of the program.

Typically, parameters can be passed to override the initial value of these variables.

Variables That You Can Specify

The following are definitions and initializations of variables used by the program associated with the def.c file.

Auto_mode

Specifies to the entry library to automatically select the appropriate mode: Query, Insert, or Update. To set this mode, you must define this variable as *True*. If you define this variable as *False*, the user must manually specify a query and then an update.

Force_query

Specifies to the entry library that an unsuccessful query must occur before initiating Insert mode. To set this mode, you must define this variable as *True*. If you define this variable as *False*, the user can initiate Insert mode at any time.

Sel_2nd_query

Specifies that, when the Find ID command is selected by the user, the entry library allows the user to specify an additional table in which to restrict the ID search. For example, the program could restrict the ID lookup to those who are alumni. The list of possible tables in which to restrict the search depends on those tables that are bound on the associated form. To set this mode, you must define this variable as *True*.

Debug_level

Specifies the level of progress messages to display. Normal messages appear from level 1 to 7. You can increase the level of debugging messages from levels 9 through 60. The system displays messages set at or below the debug level that you specify.

Pause_level

Specifies the message level, and higher, for which the program should be paused. If a message appears, and the pause level is less than or equal to the level of the message, the program pauses for the user's response.

System_uid

Tracks the user ID that last added or updated a row. The library initializes the variable, which is used during an add or update to a row.

System_gid

Tracks the group ID that last added or updated a row. The library initializes the variable, which is used during an add or update to a row.

Today

Tracks the last add and update of a row. The library initializes the variable, which is used during an add or update to a row.

Example of Variables

The following is an example specification of variables in a def.c file.

```
/* Local variables and program parameters */
int      auto_mode = FALSE;
int      force_query = FALSE;
int      sel_2nd_query = FALSE;
int      debug_level = 0;
int      pause_level = 1000;
int      system_uid;          /* buffer for system user id */
int      system_gid;         /* buffer for group id */
long     common_id;          /* buffer for common id */
long     today;              /* today's INFORMIX date */
char     ofc_addby[] = "ADM "; /* Default office add by code */
char     prog_code[] = "UNDG"; /* Default program code to use */
char     site_code[] = "CARS"; /* Default site code to use */
char     tick_code[] = "ADM "; /* Default tickler code to use */
```

Entry Library Def.c Local Functions

Introduction

Jenzabar designed Entry Library to allow for easy addition of local functions to perform necessary business logic in the application. There are two types of functions:

- Check functions, which the user triggers when modifying a value of a column. You specify check functions in the *check field* array.
- Special functions, which operate at the table/row level. You specify special functions in the *filename* array, specifically in the *special flags* and *special function* sections of the *filename* array.

You should declare any local functions that you create in the *Local functions* array of the library entry def.c file.

Note: Typically, locally created functions are located in the *select.c* or *chklocal.c* source files in the program directory.

Check Functions

The following are check functions that exist in the library.

ent_setdefault

Sets column value based upon a lookup-joining clause, only if the current value is zero or blank.

ent_setfield

Sets column value based upon a lookup-joining clause.

ent_chk_id

Verifies that an ID column value exists in some other table to ensure that the ID is a faculty member, business, or any other ID-related entity.

ent_chk_ss

Verifies that the entered social security number does not already exist in the database.

Special Functions

The following are special functions that exist in the entry library.

idperm

Specifies to the program to perform a table lookup to verify that a user's UID or GID exists in the ID Permissions table (*idperm_table*) for the office code on one of the following associated records being updated:

- ID record (*id_rec*)
- Profile record (*profile_rec*)

Note: This function requires the use of the ENT_FLGET and ENT_FLUPD special flags. This function does not stop you from changing values on the ID and profile columns on the screen, but disallows any changes from being written to the database.

ent_spcfunct

Implements the sorting and filtering logic available on scroll tables viewed in detail windows.

Note: This function uses the setup information from the following tables:

- Entry Selection table (*entssel_table*)
- Entry Selection Criteria table (*entselcrit_table*)

You must set the special function flags, ENT_SCSTART and ENT_SCGET, in the filename array for the table that you want to sort and filter.

When you enter the scroll region set for a scroll table (set with ENT_SCSTART), the system sorts the rows based upon the current sort criteria. Upon retrieving rows for display (set with ENT_SCGET), the system checks the row for meeting display criteria and either displays or skips the row.

Local Functions Example

The following is an example specification of local functions.

```
/* Local functions */  
int      chk_id();  
int      idperm();
```

Entry Library Def.c Program Parameters

Introduction

The Program Parameters array (prog_params) of an Entry Library program's def.c file does the following:

- Identifies which options are expected and recognized by the program
- Provides the label mapping to the parameters that is used by the entry library to gain access to the locally defined program variables

Parameter Types

The prog_params array identifies which options are expected and recognized by the program. The param_type structure consists of the following seven elements.

Note: The seven elements are true for all programs using the CX parameter routines.

1. option
Defines a recognized option to the program with a single character. If you use a **\0**, the parameter being defined is not accessible on the calling line of the program.
2. buffer address
Points to the current value of the option. The option should be a character pointer and therefore there is a (char *) in front of non-character type variables. If a string is used here, such as ADMMENU, the length variable must be zero.
3. type
Identifies the type of variable. Valid values are:
 - PRM_CHAR
 - PRM_LOGICAL
 - PRM_INT
 - PRM_LONG
 - PRM_DATE
 - PRM_DOUBLE
 - PRM_FLOAT
4. length
Identifies the length of PRM_CHAR type variables.

Note: The value should be zero for all non-char type variables and also on those char type variables where a string was given for the buffer address.
5. mask
Identifies special characteristics of the option.

Note: Remember the following about the mask element:

 - If the option is required, PRM_REQUIRED should appear in the field.
 - If the type is PRM_LOGICAL, PRM_TRUE or PRM_FALSE should be specified. PRM_TRUE/PRM_FALSE means that if the option is specified the variable should be set to TRUE/FALSE.
 - To place two attributes in the mask field, place a single vertical bar between the attributes, for example: PRM_REQUIRED|PRM_TRUE.
6. label
A short identifier for the option, usually one word. This name, along with the PROG_BUFFER macro, is used in the arrays below to specify relationships between this option and columns from the database.
7. description

A longer multi-word string used for producing usage messages.

Parameter Labels

The single character option letter and the variable name are of no importance to entry library. Jenzabar recommends that the single letter options are associated with the same labels across all entry programs for the sake of consistency. The entry library requires the following parameter labels:

- auto_mode
- debug_level
- display_only
- form_selected
- idtype_screen
- menu_title
- menuname
- pause_level
- sel_2nd_query
- scr_path

Program Parameters Example

The following is an example specification of program parameters.

```
struct param_type prog_params[] = /* array of common db fields */
{
  {'d', (char *)&display_only, PRM_LOGICAL, 0, PRM_TRUE, "display_only",
   "pass parameter to limit to display only"},
  {'p', prog_code, PRM_CHAR, 4, NULL, "prog",
   "program code to be used, default is 'UNDG'"},
  {'T', tick_code, PRM_CHAR, 4, NULL, "tick",
   "tickler code to be used, default is 'ADM'"},
  {'o', ofc_addby, PRM_CHAR, 0, NULL, "ofc_added_by",
   "specify office running program, default is 'ADM'"},
  {'m', DEFAULT_MENU, PRM_CHAR, 0, NULL, "menuname",
   "name of menu screen, default is 'admmenu'"},
  {'f', (char *)NULL, PRM_CHAR, 0, NULL, "form_selected",
   "name of desired form instead of using a menu"},
  {'t', (char *)&today, PRM_DATE, 0, NULL, "today",
   "effective date for changes"},
  {'L', site_code, PRM_CHAR, 4, NULL, "site",
   "site code to be used"},
  {'P', SCREEN_PATH, PRM_CHAR, 0, NULL, "scr_path",
   "path for screens, default is 'admit/admentry/'"},
  {'a', (char *)&auto_mode, PRM_LOGICAL, 0, PRM_TRUE, "auto_mode",
   "pass parameter to automatically enter query mode"},
  {'M', MENU_TITLE, PRM_CHAR, 0, NULL, "menu_title",
   "pass parameter to change the ring menu title"},
  {'D', (char *)&debug_level, PRM_INT, 0, NULL, "debug_level",
   "specify higher level for more messages (1,3,5,7,9)"},
  {'S', (char *)&pause_level, PRM_INT, 0, NULL, "pause_level",
   "specify lower number for more pauses (1-9)"},
  {'\0', (char *)&system_uid, PRM_INT, 0, NULL, "system_uid",
   "system's user id of person running program"},
  {'\0', (char *)&system_gid, PRM_INT, 0, NULL, "system_gid",
   "system's group id of person running program"},
  {'w', IDTYPE_SCREEN, PRM_CHAR, 0, NULL, "idtype_screen",
   "path for type window, default is 'Lib/libids/type'"},
  {'\0', (char *)&common_id, PRM_LONG, 0, NULL, "id",
   "buffer to hold id number, will soon be obsolete"},
  {'q', (char *)&sel_2nd_query, PRM_LOGICAL, 0, PRM_TRUE, "sel_2nd_query",
   "allow additional selection restrictions on name query"},
};
int max_params = (sizeof(prog_params)/sizeof(struct param_type));
```

Detail Tables

Introduction

The def.c file for an Entry Library program contains the *scrollfiles* array for specifying detail tables for the associated entry program. The *scrollfiles* array identifies files from the def.c's *filename* array that have multiple records and a related scrolling form. The detail tables are displayed in detail windows (scroll screens).

Def.c Scroll Tables Array

The scrollfiles array structure (scfile_type) consists of five fields, which are as follows:

filename

The name of the database file

screen name

The name of the progsr for this detail window

option

The character used to select this detail window from the pop-up menu

line 1 description

The user-friendly description of the detail window. The description appears in the detail window pop-up menu, displayed when you select **Scroll**.

line 2 description

The user-friendly description of the detail window, if the description of line 1 is an empty string, this description line appears in the pop-up menu.

Note: This field should typically be set to the empty string, "".

Example Scroll File Array

The following is an example specification of a scroll file array.

```
struct scfile_type      scrollfiles[] =      /* array of scroll files */
{
    { "accomp_rec", "accomp", 'A', "Accomplishments ", ""},
    { "ctc_rec", "ctc", 'C', "Contacts ", ""},
#ifdef ENABLE_FIN_AID
    { "aid_rec", "aid", 'D', "fin aiD awards ", ""},
#endif
    { "ed_rec", "ed", 'E', "Education ", ""},
    { "enr_stat_rec", "enrstat", 'T', "", "enrollmenT status "},
    { "relation_rec", "rel", 'F', "First relation ", ""},
    { "relsec_rec", "relsec", 'R', "", "second Relation "},
    { "hold_rec", "hold", 'H', "Holds ", ""},
    { "int_rec", "int", 'I', "Interests ", ""},
    { "involve_rec", "invl", 'V', "inVolveMents ", ""},
    { "aa_rec", "aa", 'O', "", "Other address "},
    { "addree_rec", "addree", 'N', "", "other Name "},
#ifdef ENABLE_MULTISITE
    { "site_rec", "site", 'S', "Sites ", ""},
#endif
#ifdef ENABLE_IMMUNE
    { "immune_rec", "immune", 'U', "immUnizations ", ""},
#endif
    { "exam_rec", "exam", 'X', "", "tests/eXams "},
    { "emp_rec", "emp", 'W', "", "Work"},
};
int      max_scfiles = (sizeof(scrollfiles)/sizeof(struct scfile_type));
```

Tables for Entry Library Screens

Introduction

The filename array of the Entry Library def.c file identifies the tables that are bound to the screens used by the entry program. The file_type structure consists of two required fields and three optional ones. The fields are as follows:

- filename
- getkey
- putkey
- special flags
- special function

Filename Array Fields

The following fields are in the filename array.

filename

Specifies the name of the database table.

getkey

Specifies the index to be used for retrieving rows from this table

Note: The columns that compose the getkey must appear in the common fields array below so that the program will know what values to fill into the index columns before loading rows from the database table. In addition, the order in which the tables are listed in the filename array are important when the loading of one table's rows is dependent upon the values of another table.

putkey

Specifies the index to be used for updating rows from this table. This information is determined by the library if the value is set to NULL

special flags

Specifies any special processing on the table

special function

Specifies a locally written function or a standard entry library function prepared to handle specified conditions. You must enter the above if any special flags were specified other than: ENT_LOCK, ENT_AUTOINS, ENT_FORCELOAD, and ENT_ADDID.

Special Flags You Can Specify

The following are the special flags that you can specify:

ENT_LOCK

Locks the row and disallows selection or update by other users

CAUTION: Locks can seriously affect the ability of others to perform queries against the database. The application update logic verifies that no one else has modified the row.

ENT_AUTOINS

Allows new rows to be added due to defaulted values without requiring modification by the user.

Note: If you do not use this flag, the program only adds rows if the user changes a column value within the row.

ENT_FORCELOAD

Causes the row to be loaded even when none of the fields from the table are bound on the screen currently in use.

Note: By default, the program retrieves rows only if the table name is bound to the currently active form.

ENT_SCSTART

Provides an alternate sorting of the rows. This is a special function flag that is checked at the start of entry in a scrolling region (or detail window).

ENT_SCGET

Filters out rows that are not of interest, or you do not want to see. This is a special function flag that is checked during the retrieval of each row within a scrolling region (or detail window).

ENT_ADDID

Allows the ability to add or update an ID from within the related scroll region via a PTP (process-to-process) connection to identity.

Other Special Function Flags

The following special flags work in conjunction with special functions:

- ENT_FLADD
- ENT_FLUPD
- ENT_FLDEL
- ENT_FLGET
- ENT_FLWRITE
- ENT_FLALL
- ENT_SCADD
- ENT_SCUPD
- ENT_SCDEL
- ENT_SCCHK
- ENT_SCALL
- ENT_ALL
- ENT_READONLY
- ENT_NOREAD
- ENT_UPDATE
- ENT_NOINSERT
- ENT_NODELETE
- ENT_NOPERMS

Table Level Functions

Two table level functions come with Entry Library. They are as follows:

ent_spec_func

This special function manages the appropriate calls to sort and filter routines based on criteria as defined in the Entry Selection table (entsel_table) and Entry Selection Criteria table (entselcrit_table).

Note: You must use the ENT_SCSTART and ENT_SCGET special flags with this function.

idperm

This special function performs a table lookup to verify that a user's UID or GID exists in the ID Permissions table (idperm_table) for the office code on the ID record (id_rec) or associated Profile record (profile_rec) being updated.

Note: You must use the ENT_FLGET and ENT_FLUPD special flags with this function.

Special Flag Example

The following is an example specification of special function flags.

```

struct file_type      filename[] =          /* array of filenames */
{
#ifdef ENABLE_IDPERMS
  { "id_rec",         "id",    NULL,  ENT_LOCK|ENT_FLGET|ENT_FLUPD,idperm},
#else
  { "id_rec",         "id",    NULL,  ENT_LOCK},
#endif
  { "adm_rec",        "adm_prim","adm_prim",  ENT_LOCK|ENT_AUTOINS},
  { "tick_rec",       "id",     "tick_prim",
                                ENT_LOCK|ENT_FORCELOAD},
#ifdef ENABLE_IDPERMS
  { "profile_rec",   "id",     NULL,
                                ENT_LOCK|ENT_AUTOINS|ENT_FLGET|ENT_FLUPD,idperm},
#else
  { "profile_rec",   "id",     NULL,  ENT_LOCK|ENT_AUTOINS},
#endif
  { "church_rec",    "id",     NULL,    ENT_LOCK},
  { "bus_rec",       "id",     NULL,    ENT_LOCK},
  { "milit_rec",     "id",     NULL,    ENT_LOCK},
  { "disab_rec",     "id",     NULL,    ENT_LOCK},
  { "sch_rec",       "id",     NULL,    ENT_LOCK},
  { "prog_enr_rec",  "prog_prim",  NULL,
                                ENT_LOCK|ENT_FORCELOAD},
  { "ctc_rec", "id", "ctc_no", ENT_AUTOINS|ENT_SCSTART|ENT_SCGET,ent_spec_func},
  { "site_rec", "id", "site_no", ENT_AUTOINS},
  { "enr_stat_rec", "id", "enrstat_key1"},
  { "exam_rec", "id", "exam_no", ENT_SCSTART|ENT_SCGET, ent_spec_func},
  { "hold_rec", "id", "hld_no"},
  { "immune_rec", "id", "immune_no"},
  { "int_rec", "id", "int_prim", ENT_SCSTART|ENT_SCGET, ent_spec_func},
  { "involve_rec", "id", "invl_no"},
  { "aa_rec", "id", "aa_prim"},
  { "aid_rec", "id", "aid_prim", ENT_SCSTART|ENT_SCGET, ent_spec_func},
  { "addree_rec", "prim_id", "addree_prim"},
  { "accomp_rec", "id", "accomp_no"},
  { "relation_rec", "prim_id", "rel_no", ENT_ADDID},
  { "relsec_rec", "sec_id","relsec_no", ENT_ADDID },
  { "ed_rec", "id", "ed_no", ENT_SCSTART|ENT_SCGET, ent_spec_func},
  { "emp_rec", "id", "emp_no", ENT_ADDID},
};
int      max_files = (sizeof(filename)/sizeof(struct file_type));

```

Table Update Order

Introduction

The *updateorder* array of the Entry Library *def.c* file specifies the order in which the program performs table updates. The update order can be an important processing concern.

For example, it is absolutely necessary that the Contact record (*ctc_rec*) be processed before the Tickler record (*tick_rec*). This is because if a Contact record is updated, the program is supposed to reset the Tickler record's *nextreview* date value. If the Tickler record is written before the new or updated Contact record, then the update of the Tickler record's *last_upd_date* will not get written to the database.

Update Order Array Fields

The *updateorder* array *ffile_type* structure consists of the following two fields.

filename

This field is the name of the database file.

index

This field is set by the program. Leave as 0.

Matching Entries in the Filename Array

The number of entries in the *updateorder* array must match the number of entries in the filename array. The entry library checks that both arrays have the same number of entries and prints an error message if they do not match. The entry library only verifies the number of entries. It is important that all tables listed in the *filename* array are also listed in the *updateorder* array.

Update Order Array Example

The following is an example specification of an update order.

```
/* file update order */
struct ffile_type  updateorder[] =
{
    { "id_rec", 0 },
    { "adm_rec", 0 },
    { "church_rec", 0 },
    { "bus_rec", 0 },
    { "sch_rec", 0 },
    { "ctc_rec", 0 },
    { "tick_rec", 0 },
    { "prog_enr_rec", 0 },
    { "profile_rec", 0 },
    { "milit_rec", 0 },
    { "disab_rec", 0 },
    { "enr_stat_rec", 0 },
    { "exam_rec", 0 },
    { "hold_rec", 0 },
    { "immune_rec", 0 },
    { "int_rec", 0 },
    { "involve_rec", 0 },
    { "aa_rec", 0 },
    { "aid_rec", 0 },
    { "addree_rec", 0 },
    { "accomp_rec", 0 },
    { "relation_rec", 0 },
    { "relsec_rec", 0 },
    { "ed_rec", 0 },
    { "emp_rec", 0 },
    { "site_rec", 0 },
};
int  max_upfiles = (sizeof(updateorder)/sizeof(struct ffile_type));
```

TABlename Array in an Entry Library Program

The following two lines satisfy the linking requirements of the entry library for a table.

```
struct table_type  tablename[1];
int  max_tables = 0;
```

Note: The *tablename* array is now obsolete due to enhancements in the screen package.

Table and Field Links

Introduction

The Entry Library `def.c` file contains arrays for specifying links to tables and common fields between tables to be used by an entry program. The arrays are as follows:

- Common Fields array
- Update Field array
- Add Field array

Common Field Array Structure Definition

The three linking arrays all use the *tfield_type* structure for specifying relationships between fields of different files. The *tfield_type* structure consists of five fields. The fields are as follows.

tablename

Name of the database table used by the program

destination column

Name of the column in the destination table that is to be set to the value of the common field

destination tablename

Name of the tablename that contains the destination column. A NULL in this field indicates that the destination tablename is the same as the tablename specified in the first field of the structure.

Note: This field is always set to NULL in this array.

common column

Name of column whose value will be copied into the destination column buffer.

common table

Name of the table that the common column is located in. If you specify `PROG_BUFFER`, the common column name corresponds to the label in the `prog_params` array.

Common Field Array

Entry Library programs use the *commonfld* (common field) array to determine the common fields between tables. The entry program uses this information when:

- Binding fields to forms
- Loading rows
- Updating row buffers before writing the row to the database.

Note: The common table specified must appear in the *filename* array before the destination file appears in the *common field* array.

Information for Loading Rows

When loading rows from a table, an Entry Library program uses the following in the `def.c` file:

- The *getkey* field of the *filename* array to identify which key to use to retrieve rows.
- The *common field* array for locating the correct row using the specified getkey.

Before loading the row(s), the program sets the key fields with the desired values from common fields of other rows (or parameters passed to the program) as specified in the *common field* array.

Note: All fields specified in the *getkey* field (in the *filename* array) must also appear in the *common fields* array.

Buffers for Binding Columns and Updating Records

A Entry Library program uses one buffer for multiple fields for both of the following:

- Binding columns to forms
- Updating records before writing them to the database

The use of one buffer has the following advantages:

- Immediate update of screens when two common fields are bound to the same form. If the value changes in one column, the value in another common column also changes.
- Change of all files with the fields in common with the changed field when the program writes records to the database

Common Fields Array Example

The following is an example specification of a *common fields* array.

```
struct  tfield_type      commonfld[] =          /* array of common db fields */
{
  {"id_rec",      "id",      NULL, "id",      PROG_BUFFER},
  {"adm_rec",     "id",      NULL, "id",      "id_rec"},
  {"adm_rec",     "prog",    NULL, "prog",    PROG_BUFFER},
  {"church_rec",  "id",      NULL, "id",      "id_rec"},
  {"bus_rec",     "id",      NULL, "id",      "id_rec"},
  {"milit_rec",  "id",      NULL, "id",      "id_rec"},
  {"disab_rec",  "id",      NULL, "id",      "id_rec"},
  {"sch_rec",     "id",      NULL, "id",      "id_rec"},
  {"tick_rec",   "id",      NULL, "id",      "id_rec"},
  {"tick_rec",   "tick",    NULL, "tick",   PROG_BUFFER},
  {"profile_rec", "id",      NULL, "id",      "id_rec"},
  {"prog_enr_rec", "id",      NULL, "id",      "id_rec"},
  {"prog_enr_rec", "site",   NULL, "site",   PROG_BUFFER},
  {"prog_enr_rec", "prog",    NULL, "prog",    "adm_rec"},
  {"relation_rec", "prim_id", NULL, "id",      "id_rec"},
  {"relsec_rec",  "sec_id",   NULL, "id",      "id_rec"},
  {"aa_rec",      "id",      NULL, "id",      "id_rec"},
  {"aid_rec",     "id",      NULL, "id",      "id_rec"},
  {"addree_rec",  "prim_id", NULL, "id",      "id_rec"},
  {"accomp_rec",  "id",      NULL, "id",      "id_rec"},
  {"accomp_rec",  "site",   NULL, "site",   PROG_BUFFER},
  {"immune_rec",  "id",      NULL, "id",      "id_rec"},
  {"involve_rec", "id",      NULL, "id",      "id_rec"},
  {"int_rec",     "id",      NULL, "id",      "id_rec"},
  {"hold_rec",   "id",      NULL, "id",      "id_rec"},
  {"enr_stat_rec", "id",      NULL, "id",      "id_rec"},
  {"enr_stat_rec", "prog",    NULL, "prog",    PROG_BUFFER},
  {"exam_rec",   "id",      NULL, "id",      "id_rec"},
  {"exam_rec",   "site",   NULL, "site",   PROG_BUFFER},
  {"emp_rec",    "id",      NULL, "id",      "id_rec"},
  {"ed_rec",     "id",      NULL, "id",      "id_rec"},
  {"ed_rec",     "site",   NULL, "site",   PROG_BUFFER},
  {"ctc_rec",    "id",      NULL, "id",      "id_rec"},
  {"ctc_rec",    "tick",   NULL, "tick",   PROG_BUFFER},
  {"site_rec",   "id",      NULL, "id",      "id_rec"},
};
int  max_cmnflds = (sizeof(commonfld)/sizeof(struct tfield_type));
```

Update Field Array

Entry Library programs use the *updatefld* (update field) array to determine the columns to be set if the user updates a row.

Note: Columns set in this array are typically the following column types:

- *last update date*, which you set to the *today* field from the program parameters array.
- *last updated by user ID*, which use the *system_uid* field from the program parameters array.
- *next review date* in the Tickler record (*tick_rec*), which is set to the current date if the user updates a Contact or Tickler record.

Update Field Array Example

The following is an example specification of the *update field* array.

```
struct tfield_type    updatefld[] = /* array of common db fields */
{
    {"id_rec",      "upd_date",      NULL, "today",  PROG_BUFFER},
    {"sch_rec",    "upd_date",      NULL, "today",  PROG_BUFFER},
    {"profile_rec", "prof_last_upd_date", NULL, "today",  PROG_BUFFER},
    {"church_rec", "upd_date",      NULL, "today",  PROG_BUFFER},
    {"bus_rec",    "upd_date",      NULL, "today",  PROG_BUFFER},
    {"ctc_rec",    "next_rvw_date", "tick_rec", "today",  PROG_BUFFER},
    {"tick_rec",   "next_rvw_date",  NULL, "today",  PROG_BUFFER},
};
int    max_updflds = (sizeof(updatefld)/sizeof(struct tfield_type));
```

Add Field Array

Entry Library programs use the *addfld* (add field) array to determine the columns to set if the user adds a record.

Note: Columns set in this array are typically those columns which have common fields from the PROG_BUFFER.

An example entry in this array is when you do not want to link two similar fields as the same field with a common field entry. The following example specifies that when adding a Program Enrollment record (*prog_enr_rec*), the *major1* field will be set to the current value of the *adm_major* field from the Admission record (*adm_rec*).

Add Field Array Example

The following is an example specification of the *add field* array.

```
struct tfield_type    addfld[] = /* array of common db fields */
{
    {"id_rec",      "ofc_add_by",  NULL, "ofc_added_by", PROG_BUFFER},
    {"enr_stat_rec", "ofc_add_by",  NULL, "ofc_added_by", PROG_BUFFER},
    {"hold_rec",    "add_date",    NULL, "today",      PROG_BUFFER},
    {"prog_enr_rec", "major1",     NULL, "major",       "adm_rec"},
    {"church_rec", "upd_date",    NULL, "today",      PROG_BUFFER},
    {"bus_rec",    "upd_date",    NULL, "today",      PROG_BUFFER},
    {"ctc_rec",    "next_rvw_date", "tick_rec", "today",  PROG_BUFFER},
    {"sch_rec",    "upd_date",    NULL, "today",      PROG_BUFFER},
    {"site_rec",   "site",        NULL, "site",       PROG_BUFFER},
};
int    max_addflds = (sizeof(addfld)/sizeof(struct tfield_type));
```

Special Check Functions

Introduction

The Entry Library `def.c` file contains the `chkfld` (check field) array to identify columns that have special check functions.

Check Field Array Fields

The *check field array* `chkfld_type` structure consists of the following four fields.

column name

The name of the column in which to perform special checking

tablename

The name of the table that contains the column

function

The check function to execute

string

The parameter passed to the check function.

Check Functions You Can Specify

The following are check functions that you can specify.

Note: The column level check functions that exist within the library include:

- `ent_chk_id`
- `ent_setdefault`
- `ent_setfield`
- `ent_chk_ss`

`ent_chk_id`

Uses the string parameter to specify an additional table in which the ID must exist, in order for the ID to be valid.

`ent_setdefault`

Fills in looked up values for columns that are blank, zero or NULL. The function uses the string parameter to specify what column should be filled with which value. The format of the string parameter is:

Example: `[dest_table.]dest_column = [src_table.]src_column joining src_table.join_column`

`ent_setfield`

Fills in looked up values for columns that are blank, zero or NULL. The function is similar to `ent_setdefault` except that the function sets the column regardless of the column's current value.

`ent_chk_ss`

Verifies that the `ss_no` entered is not a duplicate of one existing in the database.

Check Function Array Example

The following is an example specification of the *check function* array.

```
struct  chkfld_type      chkfld[] =  /* array of fields for special checking */
{
  {"church_id", "profile_rec", ent_chk_id, "church_rec"},
  {"adv_id", "adm_rec", ent_chk_id, "fac_rec"},
  {"sch_id", "ed_rec", ent_chk_id, "sch_rec"},
  {"sch_id", "ed_rec", ent_setfield,
   "ed_rec.ceeb = sch_rec.ceeb joining sch_rec.id"},
  {"ceeb", "ed_rec", ent_setfield,
   "ed_rec.sch_id = sch_rec.id joining sch_rec.ceeb"},
  {"zip", "id_rec", ent_setfield,
   "city = zip_table.city joining zip_table.zip"},
  {"zip", "id_rec", ent_setfield,
   "st = zip_table.st joining zip_table.zip"},
  {"zip", "aa_rec", ent_setfield,
   "st = zip_table.st joining zip_table.zip"},
  {"ref_id", "adm_rec", ent_chk_id},
  {"cnslr_id", "adm_rec", ent_chk_id},
  {"ctc_id", "bus_rec", ent_chk_id},
  {"news1_id", "profile_rec", ent_chk_id},
  {"news2_id", "profile_rec", ent_chk_id},
  {"adv_id", "prog_enr_rec", ent_chk_id},
  {"grd_rpt_id", "prog_enr_rec", ent_chk_id},
  {"prim_id", "relation_rec", ent_chk_id},
  {"sec_id", "addree_rec", ent_chk_id},
  {"id_used_by", "addree_rec", ent_chk_id},
  {"bus_id", "emp_rec", ent_chk_id},
  {"sec_id", "relation_rec", ent_chk_id},
  {"prim_id", "relsec_rec", ent_chk_id},
  {"sec_id", "relsec_rec", ent_chk_id},
  {"id", "involve_rec", ent_chk_id},
  {"corr_id", "ctc_rec", ent_chk_id},
  {"prev_name_id", "id_rec", ent_chk_id},
};
int      max_chkflds = (sizeof(chkfld)/sizeof(struct chkfld_type));
```

Process to Process (PTP) Functionality

Introduction

The Entry Library `def.c` file contains an area for specifying programs and parameters for Process to Process (ptp) functionality in the entry program. You can indicate the source and destination files and fields to send via ptp.

Process to Process Field Structure

The process to process section `ptpfld_type` structure consists of four fields. The fields are as follows:

source file

The name of file on server side containing source field.

source field

The name of field on server side to be copied to client.

dest file

The name of file on client side containing dest field.

dest field

The name of field on client side to receive copied value.

Specifying PTP Functionality

The following two items appear in this section of the `def.c` file. The entry program uses both items to implement the `ENT_ADDID` functionality, which allows adds and updates to referenced IDs from within detail windows.

ent_ptpprog

Represents the invocation string used to start a child (client) process. The value must:

- Be a BINPATH-relative string
- Reference another entry library program
- Contain appropriate and relevant parameters

The entry library appends a `-o` parameter followed by an *office_add_by* code to the string provided. The targeted entry program needs to recognize each of these parameters, and the current entry program should have a program parameter with an *ofc_add_by* label.

ptpfld (ptp field) array

Used to identify fields that are copied across the PTP pipe when the program invokes the `ENT_ADDID` functionality within a detail window (or detail window). The system copies columns specified in this array across the PTP pipe and populates the fields on the client side to eliminate key strokes. For example, when both the parent and child in Parent-Child relationship reside at the same address, a user can copy over relevant address information when creating the related ID.

Note: The original intent of this feature was to provide the ability to add a new ID number on-the-fly from a scroll region such as the Relationship detail window, associating the ID being added with the currently queried ID.

Process To Process Example

The following is an example specification of process to process functionality.

```
/* specify program and parameters for ptp, relative to BINPATH */
char    *ent_ptpprog = "identry -f ptpid -p";
/* indicate the source and destination files/fields to send via ptp */
struct  ptpfld_type  ptpfld[] =
{
    "id_rec", "addr_line1", "id_rec", "addr_line1" },
    "id_rec", "addr_line2", "id_rec", "addr_line2" },
    "id_rec", "city", "id_rec", "city" },
    "id_rec", "st", "id_rec", "st" },
    "id_rec", "zip", "id_rec", "zip" },
    "id_rec", "ctry", "id_rec", "ctry" },
    "id_rec", "phone", "id_rec", "phone" },
    "id_rec", "phone_ext", "id_rec", "phone_ext" },
};
int      max_ptpflds = (sizeof(ptpfld)/sizeof(struct ptpfld_type));
```

Address Maintenance

Introduction

The Entry Library `def.c` contains the *relationship field* (`relfld`) so that it can automatically maintain address information for related IDs as well as for the currently selected ID. This ability is partially table-driven and partially `def.c` driven. The *relfld* array exists to allow custom specification of columns in the ID record (`id_rec`) which, when modified, cause the invocation of the relationship maintenance functionality.

Note: Currently, only modifications to columns of the ID record (`id_rec`) can invoke this logic.

Relationship Field Structure

The Relationship Field `relfld` structure consists of one field:

column name

The name of the `id_rec` column to consider for maintenance

Address Maintenance Example

The following is an example specification of address maintenance functionality.

```
/* indicate the id_rec fields on which to maintain relationship info */
struct relfld_type relfld[] =
{
    {"addr_line1" },
    {"addr_line2" },
    {"city" },
    {"st" },
    {"zip" },
    {"ctry" },
    {"phone" },
    {"phone_ext" },
};
int max_relflds = (sizeof(relfld)/sizeof(struct relfld_type));
```

GET_PRIMARY_REC Functions

Introduction

The entry library calls the GET_PRIMARY_REC function to retrieve:

- A row from the first table in the filename array
- Rows for the rest of the tables based upon column values from the retrieved row and parameters in the program

Note: The entry library can work against other records besides the ID record (id_rec) and related tables.

The GET_PRIMARY_REC function is typically defined in the getrec.c source file of each entry program. The function calls, in turn, the ent_getprim_rec function from the entry library designed to retrieve an ID record.

The following is the code from a typical getrec.c source file:

```
#include "dec.h"
/* -----
=====
    get_primary_rec(pscr, pfile) - get primary record to be processed
    returns: SCR_DONE - record found, enter display/update mode
             SCR_OK   - record not found, enable insert mode
             SCR_ABORT - record not selected, exit query mode
             SCR_ERR  - unexpected error occurred, exit
=====
----- */
get_primary_rec(pscr, pfile)
SCREEN *pscr; /* current screen pointer */
struct file_type *pfile;
{
    return(ent_getprim_rec(pscr, pfile));
}
```

GET_PRIMARY_REC Processing

The purpose of this function is to find the desired row and load it into the current record buffer (rec_c) of the table and return an SCR_DONE status. Otherwise, an error return status is expected.

The possible return values for the function are: SCR_DONE, SCR_OK, SCR_ABORT, and SCR_ERR. See the above example for descriptions of these values.

The parameters passed to the get_primary_rec function provides a pointer, pscr, to the currently displayed screen and a pointer, pfile, to a file type structure which holds valuable information about the targeted table from which a row is to be selected.

Suggestions for Writing a GET_PRIMARY_REC Function

Following are suggestions for writing your own get_primary_rec function:

- If you want to keep the function simple, you can hard-code an scr_get against a single column and select the desired row in this way.
- If you want to make the function more complex, such as allowing the user to enter selection criteria in certain columns and then build a select statement that retrieves the matching rows, refer to the following:
 - If multiple rows are found, the user can use a pop-up window to select the row desired. You can implement this feature by adding a new screen group field to the screen definition and doing a scr_getset on the group field.
 - In addition, you can specify a check function for the scr_getset that would track the criteria that was entered for the different columns.

In either case, you must create an SQL select statement that:

- Returns all of the expected columns in the order specified by the fields dbview structure array
- Copies the returned buffer into the buffer pointed to by rec_c

File Type Structure Example

The file_type structure is defined in the entry.h include file. Its definition follows:

```

struct file_type
{
    char      *name;          /* pointer to informix filename */
    char      *getkey;       /* pointer to get key */
    char      *putkey;       /* pointer to put key */
    long      spcflag;       /* perms and special funct calls */
    int       (*spcfunc)();  /* pointer to special functions */
                                /* spcfunc(type,pfile,pscr,errbuff) */
                                /* where type - type of call */
                                /* pfile - pointer to file */
                                /* pscr - pointer to screen */
                                /* buff - pnt to error buff */
    struct dbview *fields;   /* pointer to fieldlist dbview array */
    long      *fieldperms;   /* pointer to fields permission array*/
    int       numfields;     /* number of fields in file */
    int       totfields;    /* number of fields in file */
    char      *where;        /* pointer to where clause (SQL) */
    char      *wheresqlda;   /* pointer to where sqlda */
    int       rec_len;       /* record length */
    char      entsel[9];     /* current entry sort/selection code */
    struct dmm *pdmm;        /* pointer to scroll file dmm */
    char      *rec_status;   /* record status (ONE character only)*/
    char      *rec_c;        /* pointer to current record buffer */
    char      *rec_p;        /* pointer to previous record buffer */
    char      *rec_a;        /* pointer to add compare rec buffer */
    char      *rec_d;        /* pointer to defaulted rec buffer */
    char      *rec_i;        /* pointer to initialized rec buffer */
    char      *rec_m;        /* pointer to deflt match rec buffer */
};

```

File Type Structure Members

The following are some of the more important structure members.

name

A pointer to the database table name.

fields

A pointer to an array of dbview structures, which holds column names (vwname), column offsets (vwstart) into a buffer (rec_c), column types (vwtype), and column lengths (vwlen).

numfields

The number of selectable columns in this table.

totfields

The number of columns to be bound from this table.

rec_len

The record length in bytes (very useful for copying record structures from one place to another).

pdmm

A pointer to a dmm that holds multiple rec_status, rec_c, rec_p, and rec_a groups.

rec_c

A pointer to the current buffer. This is the buffer that the get_primary_rec function is to fill in.

rec_p

A pointer to the previous buffer. The previous buffer is initialized with a copy of the current buffer just before entering update mode.

rec_a

A pointer to the copy of the table buffer

rec_d

A table buffer pointer that holds the default values from the current form.

rec_i

A table buffer in which all columns are blank or zero.

rec_m

A table buffer that holds the match values from the current form.

IS_DISPLAY_ONLY Functions

Introduction

The IS_DISPLAY_ONLY function allows the programmer to indicate to the entry library that this user should not be allowed to update any of the displayed information. You can specify an IS_DISPLAY_ONLY function in an Entry Library program's def.c file.

Normally you define the function in the display.c source file as follows to not restrict update:

```
#include "mac.h"
/* -----
=====
    is_display_only() - determine if selected row may be modified
                      return - non-zero to enter display only mode.
                          - zero   to allow update of related information.
=====
----- */
is_display_only()
{
    return(0);
}
```

Determining a Column's Value

The IS_DISPLAY_ONLY function does not provide any parameters. Since there are no compile time structures of the data rows, all column values must be referenced as offsets from a buffer or through other pointers.

Following is an example of how to determine the value of a column.

```
struct file_type *pfile;
struct dbview   *pfld;
long            idno;
pfile = ent_get_pfile("id_rec");
pfld = dbe_getpview("id", pfile->fields, pfile->numfields);
idno = *(long *) (pfile->rec_c + pfld->vwstart);
```

Note: In this example, the following is determined:

- The value of the ID record (id_rec), id column
- The current record buffer

Note: The type of the column is assumed to be long. These variables should be declared as static and then set just the first time into the function since the value of pfile and pfld in this example would not change over the course of the application execution.

Check Functions

Introduction

A check function can be triggered/executed when a user directly modifies a specific database column. When a change of value in one column immediately causes other things to happen, a check function is needed. You can specify a check function in an Entry Library program's def.c file.

An example check function declaration follows:

```
ent_chk_ss(pchkfldp, doscr)
struct chkfldp_type *pchkfldp; /* Field name to be checked */
SCREEN *doscr; /* do any scr functions */
```

Check Function Return Statuses

The return statuses from a check function include the following.

ENT_OK

Indicates that the check function was successful and that the cursor may move on to the next column.

ENT_INVALID

Indicates that the check function was not successful and that the cursor should remain on the same column.

Check Function Parameters

If a non-NULL value is passed for doscr, the check function should handle any necessary screen refreshing. If the value of doscr is NULL, no screen is currently active and the screen should not be refreshed by the check function.

The parameters passed to the function include the following.

doscr

A pointer to currently displayed screen

pchkfldp

A pointer to the column that has been modified.

Check Function Pointers

The pchkfldp parameter is a pointer to a chkfldp_type structure. An example of the chkfldp_type structure is as follows:

```
/* ----- This structure is used in both the file_dmlt and form_dmlt ----- */
struct chkfldp_type
{
    char                *scrname; /* screen name of code */
    struct chkfld_type *pchkfld; /* ptr to chkfld array structure */
};
```

The chkfldp_type structure provides the unique screen name of the modified code value and a pointer to the **chkfld_type** structure.

Note: This structure provides additional characteristics that are useful within the check function.

An example of the chkfld_type structure is as follows:

```
struct chkfld_type
{
char      *dbcode;      /* db name of field to check */
char      *filename;    /* db file containing dbcode */
int       (*function)(); /* function to perform checking */
char      *tablename;   /* string passed to the function to
                        specify additional information */
/* The following field are filled in by ent_formload */
char      *codebuff;    /* buffer that dbcode is bound to */
};
```

Special Functions

Introduction

A special function is triggered/executed during different table and row level processing operations and allows the programmer to affect the normal sequence of events in a program's processing. You can specify a special function in an Entry Library program's def.c file.

Return Statuses

The expected return statuses from a special function can differ depending upon the program event being checked. A special function's return status of:

- ENT_FATAL causes the program to display an error and terminate processing
- ENT_WARN causes the program to display a warning message, yet continue processing
- ENT_OK causes all normal processing to continue

Note: Additional return statuses of ENT_SKIP, ENT_INVALID, and ENT_QUIT are also possible.

Events

The currently recognized events are the following

ENT_FLGET

Triggered whenever a row is read from the database and after the rec_c buffer is copied into the rec_p buffer.

Note: This event ignores return statuses.

ENT_FLADD

Triggered after a row has been inserted. Because it appears that a partial update will be committed to the database, a non-ENT_OK return status stops the update process for all subsequent.

ENT_FLUPD

Triggered after a row has been updated. Because it appears that a partial update will be committed to the database, a non-ENT_OK return status stops the update process for all subsequent..

ENT_FLDEL

Triggered after a row has been deleted. Because it appears that a partial update will be committed to the database, a non-ENT_OK return status stops the update process for all subsequent.

ENT_FLWRITE

Triggered after the program determines that the row must go to the database, but before the row is written to the database. This event provides one last opportunity to update the column information in the rec_c buffer before updating the database. Because it appears that a partial update will be committed to the database, a non-ENT_OK return status stops the update process for all subsequent.

ENT_FLALL

Specifies all FL (File Level) events.

Note: This event is shorthand for specifying all file level events.

ENT_SCADD

Triggered just before a new scroll row is inserted into the dmm. A non-ENT_OK return status cancels the insert.

ENT_SCUPD

Triggered when the user moves off of a modified scrolling row. A ENT_OK return status causes the dmm to be updated and normal processing to continue.

ENT_SDEL

Triggered when a user attempts to delete a scrolling row. Any return status other than a ENT_OK aborts the delete operation.

ENT_SCCHK

Triggered when a user closes a detail window. Anything return status other than a ENT_OK re-displays the same detail window.

ENT_SCALL

Specifying all of the SC (SCrolling level) events.

Note: This event is shorthand for specifying all scrolling level events.

Special Function Parameters

The parameters passed to the special function are as follows.

type

The event that has triggered the function.

Pfile

A pointer to the file structure for the table that is currently being processed.

Pscr

A pointer to the currently active screen. The special function is responsible for refreshing the screen if needed.

Errbuf

A pointer to a entry library error buffer. If the return status from the special function is not ENT_OK or ENT_SKIP, this error buffer is displayed.

Special Function Example

An example special function specification is declared as follows:

```
spc_func_name(type, pfile, pscr, errbuf)
int
struct file_type
SCREEN
char
{
    switch (type)
    {
        case ENT_FLGET:
            break;
        case ENT_FLUPD:
            break;
        default:
            break;
    }
    return(ENT_OK);
}
```

Transaction Procedures

Introduction

The transaction procedure function provides hooks around the database transaction related to the updating of rows. There are three possible flags that an Entry Library program can pass to this function, including:

- ENT_START
- ENT_END
- ENT_AFTER

The entry program calls the transaction procedure with ENT_START after starting the transaction, but before processing any tables. The entry program passes ENT_END to the function after completing all table processing but before committing the database transaction. And lastly, the entry program passes the ENT_AFTER flag to the function after committing the transaction.

Transaction Procedure Example

The following is a skeletal version of the transaction procedure function that appears in the entry library.

Note: Return statuses for the following function are ENT_OK or ENT_FATAL. If ENT_FATAL is returned on the ENT_END event, the database transaction is rolled back.

```
spc_trans_proc(type)
int type;
{
    int    status = ENT_OK;
    switch(type)
    {
        case ENT_START:
            break;
        case ENT_END:
            break;
        case ENT_AFTER:
            break;
        default:
            break;
    }
    return(status);
}
```


SECTION 10 – SCREENS AND FORMS

Overview

Introduction

This section describes CX screens and forms. CX uses the following screens:

- CARS-designed screens, defined in screen definition files and presented using the CX Screen Package (SCR)
- PERFORM screens provided by INFORMIX, Inc.

This section describes the features that you can specify for a screen in a screen definition file.

Typical Entry Screens

An entry screen can display profile information, such as name, identification number, and social security number, if that information has been previously entered. Entry windows are designed for the following purposes:

- To be models of typical forms that come through an office
- To hold specific information about students, such as which church a student belongs to and which school a student attends
- To hold information about individuals and institutions, such as vendors and foundations

Typical Detail Windows

A detail window is a data entry window that enables you to view and access information that is not currently displayed on the window. A detail window holds multiple-item information, such as interests and involvements. When you access a detail window, it appears in a small window over an entry screen. Each detail window may contain one or more records.

Using the PERFORM Screen Commands

PERFORM screen commands

Using PERFORM screens, you can add and update tables in the CX database. The following lists the commands that appear on the PERFORM screens, as well as the key entries and the purpose corresponding to each command listed.

For further details on how to use each command, see The *PERFORM Screen Transaction Processor* in *Informix SQL Reference Manual*.

Add

Creates a new row in the active table.

Cancel

Cancels the last command you selected.

Current

Recalls the most up-to-date version of the screen you were viewing before you moved to another table.

Detail

Automatically selects, causes to display, and queries the active table's detail level, but only if a master-detail relationship exists for two tables.

Exit

Leaves the PERFORM screen, and returns you to the last menu you were on when you selected a table.

Finish

Proceeds to the next sequential screen.

Master

Moves to a master table from a detail table, but only if a master-detail relationship exists for two tables.

Next

Shows the next sequential row in the current list.

Output

Produces an output file in which rows appear just as they do on the screen, including data, field titles, and lines.

Previous

Shows prior row in the current list.

Query

Searches the database based on values you enter into the fields on a PERFORM screen.

Note: You can query on any field, or combination of fields, in a PERFORM screen.

Remove

Deletes the row on the screen from the active table.

Screen

Cycles through the screen pages of the form.

Table

Shows a new active table when there is more than one table displayed on the screen.

Update

Places the cursor in the first field of the current table for you to modify as many fields as necessary.

View

Shows the contents of a field of data.

Creating Screen and Form Definition Files

Introduction

CX programs use screen definition files to display data on your screen and receive input from you. Screen definition files are a part of CX Screen Package (SCR), developed by Jenzabar to assist the application programmer with the presentation of data to screens and the input of data from the user. Jenzabar designed the package to work with the INFORMIX relational database management system.

In addition to the SCR program's screen interaction with the user, SCR also provides interaction with printer devices through the Form Production System (FPS). FPS allows the program to spool the printing of data and allows you greater control over the timing and error correction of each spooled job.

Screen Section

The definition file provides the layout of a screen in a text file which can be changed without affecting the program that uses it. Screens are changed by editing the definition file.

Note: The screen definition is very similar to those used by PERFORM. If the file is a form definition instead of a screen definition, the word screen (beginning the screen section) is replaced by the word form.

Types of Fields

A screen definition can have two types of fields.

Text fields

Used to label the data fields displayed to the user

Data fields

Used for data that is displayed or to be modified by the user. Data fields are delimited by brackets ([and]) and are called by the screen field name, defined by an ASCII label found between the brackets. The field can also have a database field name defined in the attributes section, similar to a PERFORM screen (e.g., screen-field-name = database-field-name).

Screen Section Features

The features of the Screen section in the screen definition file are described below.

Multiple Use of Field Names

You can use the same screen field name in more than one location on the screen section by using the *alias* instruction.

Note: For more information on the *alias* instruction, see *Instructions You Can Specify* in this section.

Single Spaces Between Adjacent Data Fields

In the screen section, a caret (^) is a special character substituting the combination of adjacent closing and opening brackets (]), thus allowing a single space between two adjacent data fields instead of two spaces.

Display of Brackets in Text Fields

You can add brackets ([and]) into the text fields of the screen using the backslash (\) character. The backslash prevents SCR from interpreting the text field as a data field definition because of the presence brackets. For example, the string [name] would be encoded as a data field called *name*, but the string "\name\" would be part of a text field displayed as [name] to the user.

Note: The results of the backslash character is toggled between opening ([) and closing (]) brackets, so the brackets to be displayed to the user must be balanced.

Multiple Screen Sections

The feature of PERFORM that allows multiple screen sections in one screen definition file is not available in the SCR package.

Opening Brackets as a Field's Beginning of Line

When defining display-only data fields in the screen section, you can make the beginning of the line to represent the opening bracket ([) of the field. When the system finds a closing bracket (]) or caret (^) on a line of the screen definition with no opening bracket ([), the system assumes that the beginning of the line is the beginning of the data field. This feature allows the data from a field to appear in the leftmost column of the CRT screen.

For example, you can use this feature to display 80 columns of data on an 80 column screen by beginning the data field name in the first column of the screen and placing the closing bracket (]) in column 81.

Use of Most ASCII Characters

You can use most printable, non-blank ASCII characters that are not data field names. This feature allows you to define the maximum number of data fields, especially when you want to define many one- or two-character fields on one screen. The following are the characters you cannot use:

- / (slash)
- " (double quote)
- { } (braces)
- = (equal sign)
- [] (brackets)
- ^ (caret)
- > (greater than)
- < (less than)
- : (colon)
- () (parentheses)
- , (comma)
- . (period)
- \$ (dollar sign)
- ; (semi-colon)

The following field names cannot begin with the following reserved words:

- _scr
- alias
- attributes
- autonext
- blank
- center
- comments

- default
- downshift
- dwshift
- end
- errdefault
- form
- group
- gui_noproportional
- gui_nostrip
- gui_title
- include
- instructions
- joining
- length
- lookup
- noentry
- noj
- noupdate
- optional
- qclear
- qualifier
- queryclear
- required
- reverse
- right
- screen
- scroll
- type
- upshift
- verify
- void
- zerof
- zerofill

Attributes Section

The following are attributes that you can give to fields in the attributes section of a screen definition file.

Note: Since you can set and unset some of these attributes from within the application program, adding or taking away attributes may not always change the behavior of the program.

There are some differences between PERFORM screens and SCR screens in the attributes section:

- Most of the attributes that have the same names are identical, but some have different meaning, such as autonext, and some have different syntax, such as lookup and joining.
- The PERFORM feature of specifying joined fields does not apply to SCR (i.e. f1 = name = *dbname).

Attributes Section Format

The format of lines in the attributes section is as follows:

```
screenname = dbname, attribute1, attribute2, ... ;  
screenname: attribute1, attribute2, ... ;
```

Remember the following:

- When you omit the database field name (dbname), you should not use the equal sign (=)
- The comma may be replaced with a colon (:)
- All attribute lines must end with a semicolon (;)
- The white space (space, tab, newline) is always optional around the punctuation characters (equal sign, comma, colon, semicolon)

The following is an example of the Attributes section:

```
screenname = dbname, optional,  
    attribute1,  
    attribute2,  
    ... ;  
screenname: optional,  
    attribute1,  
    attribute2,  
    ... ;
```

Guidelines for the Attributes Section

To enhance the readability and maintainability of the screen definition files, use the following guidelines:

- Attributes should appear in alphabetical order one per line and indented 4 spaces
 - Note:** The one exception is the “optional” attribute which, if used, should appear directly after the “dbname” or colon of the first line.
- A blank line should separate fields in the attributes section

Attributes You Can Specify

The following are the attributes you can specify for a screen.

Autonext

Specifies that the fields in a group may be input in a circular manner. For example, when you press the return key after the last field in the group, the cursor moves to the first field in the group.

Note: Unlike PERFORM screens the autonext attribute only applies to group fields

Blank

Causes the display of a numeric field to be blank when the value is zero (0). This attribute may be used in addition to the format attribute. In the case where a format has also been given and the value is zero, the field will be blank.

Button

Sets the button attribute flag causing the field to appear as a button in QuickMate.

ButtonText=“string”

Performs the functions of the button and text attributes (binds a buffer to the field and sets it as a QuickMate button).

Center or Centerj

Centers a field in the field display. Remember the following:

- The system performs all justification on character fields when the field is input rather than when it is output. This causes the data received by the program to be centered.

- The system assumes that character data is already justified properly for output, so no centering is done when the field is displayed. This allows program screens to specify the format of character data as it will exist in the database.
- The system performs all justification of numeric fields when the field is output rather than when it is input, since numeric data is always stored the same way in the database.

Comments

Defines a string to be displayed to the user when any input is done on the field.

Example: comments = "Enter the due date"

All characters between the double quotes (") appear on the next to last line of the screen. The comments attribute has special meaning on lookup fields (fields that are part of a lookup clause). In this case, the comment value allows you to assign a value to the field in the event that the lookup value is not found.

Default

Defines the initial value of a field. SCR assigns the default value to the data associated with the screen field every time the program causes SCR to clear the field. Since programs may initialize the data outside the control of SCR, changing this attribute may or may not affect the screen display.

Example: default = "USA"

default = 123.45

default = "01/31/88"

default = today

Note: The value of the default is not quoted with double quotes (") if the field contains numeric data or is the word today. The word today may be used to default a date field to the current date.

The default attribute should NOT be used on lookup fields (fields that are part of a lookup clause). Refer to the qualifier attribute for more information.

Dwshift or downshift

Converts all upper-case letters to lower-case when the field is input. Data containing upper-case letters may appear in the field, but it will be down shifted if the field is modified during data input.

Errdefault

Defines a string to be displayed to the user when a table lookup fails. You can assign an errdefault value to fields that are part of a lookup clause. The value is used in:

- The screen or form
- Lookups based on this field (until data is changed)

Example: errdefault = "USA"

errdefault = "Not found."

errdefault = "0"

Format

Controls field displays. This attribute is useful for numeric fields and character fields.

Note: When formatting numeric fields the string defining the format of the data closely resembles the using clause of an INFORMIX ACE report.

Example: format = "\$\$, \$\$&.&&"

The following characters in a numeric format string have special meaning:

- * Asterisk fill (affects position)
- & Zero fill (affects position)
- # Blank fill (affects position)
- < Shift left (affects position)
- ' Comma separator (affects position)
- . Decimal point (affects position)
- Sign if negative (floats left)
- + Sign, negative or positive (floats left)
- (Parenthesis if negative (floats left)
-) Parenthesis if negative (affects position)
- \$ Dollar sign (floats left)

You can also format character fields, but the format string is set up differently. The character data from the field is represented by the field name surrounded by braces ('{' and '}') or parentheses ('(' and ')'), preceded by a dollar sign ('\$'). The following screen example uses a character format string:

```
screen
{
Character field...[cfld                ]
}
end
attributes
cfld: format="** ${cfld} **";
end
```

The results after entering the data, My Data, into the field would be:

Example: Character field...[** My Data **]

The field name feature may also apply to any other field on the screen. Using this method we can print the data from any other fields on the screen into the character field, as in the following example:

```
screen
{
Character field...[cfld
]
}
end
attributes
codefld: optional;
datefld: optional;
cfld: format="Code and date: ${codefld} ${datefld}";
end
```

The results if the codefld data contained ABCD and the datefld data contained 01/01/89 would be:

Example: Character field...[Code and date: ABCD 01/01/89]

Note: The character field formatting is the most useful when applied to forms and display-only fields. Also, with the character field formatting the justification attributes may be applied to the resulting format to allow centering and right justified displays.

Group

Allows a program to deal with more than one screen field as one entity. The group attribute field list is one or more field names inside parentheses separated by commas, where the field name is either a screen field or database field name defined in the screen section or earlier in the attributes section (i.e. name1, name2, ... nameN). This attribute is very closely connected to the inner workings of the program and should not be added or removed from a screen field without checking how it is used in the program. Group fields are normally used for defining the list of fields to be used for data input, or to define one line of a detail window.

Example: group = (amt,desc,type,flag)

Gui_noproportional

Communicates to CARS' GUI front-end that the text fields on the screen should be displayed using the fixed width data font instead of the default proportional font, usually used for displaying text fields.

Note: The gui_noproportional attribute is only appropriate for use on the special screen field, SCREEN_INFO.

This attribute currently maps to the *verify* attribute. The *verify* attribute should not be used on the SCREEN_INFO field.

Gui_nostrip

Communicates to CARS' GUI front-end that the text fields on the screen should be displayed as defined without the customary stripping of trailing periods.

Note: The gui_nostrip attribute is only appropriate for use on the special screen field, SCREEN_INFO.

Note: This attribute currently maps to the *void* attribute. The *void* attribute should not be used on the SCREEN_INFO field.

Gui_title

Communicates to CARS' GUI front-end what the title of the window displaying this screen definition should be.

Note: The `gui_title` attribute is only appropriate for use on the special screen field, `SCREEN_INFO`.

This attribute currently maps to the 'default' attribute. The 'default' attribute should not be used on the `SCREEN_INFO` field.

You can also use the `gui_title` attribute to specify the title of the screen in character mode. You specify a location in the screen layout for the display of the title. Following is an example:

```
screen
{
    [ SCREEN_INFO ]
}
end
attributes
SCREEN_INFO:
    center,
    [gui_noproportional,]
    [gui_nostrip,]
    gui_title = "Screen Title Text";
end
```

Help="string"

Sets the help struct element to "string" for linking into online help files.

Include

Restrict the values allowed on a screen. The include list is one or more values or ranges inside parentheses separated by commas. A *value* is any data value appropriate for the type of field. A *range* is two values separated by either a colon (:) or the word *to*. All ranges are inclusive (i.e., the lower and upper range values are valid along with any value(s) between the two).

Example: `include = (0,100 to 5000,7000)`

`include = ("A":"D", F, P, " ")`

Note: When users perform a table lookup, the window that opens displays all the data from within the parentheses. If you want the data to display with an accompanying definition or explanation, use the following syntax:

Example: `include = (Y = Yes, N = No)`

In the above example, the values Y and N will appear (associated with the text "Yes" and "No" respectively) when the user performs a lookup.

When the definition contains more than one word, use quotes around the string as in this example:

Example: `include = (H = "Head of Household", S = Single, J = "Married Filing Joint", M = "Married Filing Separately")`

White space is ignored outside of the quoted information. The equal sign (=), quotes ("), and commas (,) serve as delimiters.

Joining

Joins a screen field to a database field name. You can use this attribute for looking up the value of a field in the database. Remember the following:

- The database name contains the database file name and the database field name separated by a period (.).
- If the database name is preceded by an asterisk (*), the value of the screen field must be found in the database field being joined.

- Unlike PERFORM, if the screen field is blank and the *required* attribute is not set, the value does not need to exist in the database, even if the asterisk is included on the database name. This allows four modes of database validation:
 - *Optional blank* and *optional* in the database
 - *Optional blank* and *required* in the database if a value is entered
 - *Value required* and *optional* in the database
 - *Value required* and *required* in the database

Example: joining *ctry_table.tctry_tcode
 joining vnd_rec.vnd_ID

The joining attribute causes the database file name to be opened for use as a table and enables the table lookup command (CTRL-T) to be entered by the operator when input is being done on the field. The join field will be displayed in the window along with any fields (without constant default values) given in the lookup attribute. The database values displayed for the field may be restricted by using the default attribute on the lookup fields.

Lookup

Specifies one or more field names separated by commas that will be filled in based on the joining attribute for the field. In addition to reading values from the database, you can fill in other screen fields from data read out of the database. The field names may be:

- Screen field names or database field names that appear in the screen
- Screen field names that have been specified previously in the attributes section

Example: lookup text, amount

Note: When the you select the Table Lookup command, the fields in the lookup, which do not have a constant qualifier value specified, appear in the table window (in addition to the join field). This allows the screen to determine how much data to display in the Table Lookup command.

If a database field appears in the table lookup command but not on the screen, you can specify the field in the attributes section as optional and not displayed in the screen section.

Match="string"

Sets the default attribute to "string", and sets the noentry and nouupdate attributes. These three attributes are needed to define a field as a "match field" for library entry program screens.

Noentry

The noentry attribute is not handled internally by SCR; however, some application programs use this attribute.

Nojustify or Noj

The SCR Package uses these attributes for character fields. The default for this type of field is left justification, with other attributes allowing right and center justification. if you want the character field to remain exactly as it is entered by the user, use one of these attributes.

Nouupdate

Disallows any changes to the screen field. This feature prevents the cursor from entering into the screen field.

Note: Unlike PERFORM, which allows noentry to apply to record additions and nouupdate to apply to record updates, SCR is unaware of the status of the current record. For this reason, SCR only deals with the nouupdate attribute internally.

Optional

Allows a field to be taken off the screen display without affecting the application program. If the program is expecting a particular screen field to exist on the screen and attempts to manipulate the data, it will receive an error if the field is not defined for the screen. This attribute allows the program to deal with the screen field without actually displaying it on the screen.

Note: Some fields in the screen section are not required to appear in the attributes section. If one of these fields is removed from the screen section, you must add the field to the attributes section with the optional attribute.

Qclear or queryclear

Specifies that the field should be cleared on queries. SCR does not handle this feature internally.

Qualifier

Specifies additional criteria for a lookup field. Use this attribute on lookup fields (fields that are part of a lookup clause) only. The qualifier value allows you to make additional specifications that the table record must meet in order to be considered valid. The attribute value should always be in double quotes and may have any of the following formats:

```
qualifier = "FA"           : field must be equal to "FA"
qualifier = "=FA"         : field must be equal to "FA"
qualifier = "<>FA"         : field must not be equal to "FA"
qualifier = ">FA"         : field must be greater than "FA"
qualifier = "<FA"         : field must be less than "FA"
qualifier = ">=FA"        : field must be greater than or equal to "FA"
qualifier = "<=FA"        : field must be less than or equal to "FA"
qualifier = "FA:FZ"       : field must be greater than or equal to "FA" and
                           less than or equal to "FZ"
qualifier = "field:SCR_FIELDNAME"
                           : field must be equal to the current value of the
                           screen field 'SCR_FIELDNAME'
qualifier = "#value1,value2,..."
                           : field must be equal to one of the listed values
```

Required

Determines that the field cannot be left blank (or zero if the field is numeric) when you perform data input. If you move the cursor into a required field, the cursor cannot leave that field until you make a valid non-blank entry.

If the program performs data input on a group of fields within the control of SCR, the Finish command causes SCR to verify that all required fields in that group have been entered. If any required field contains blank (or zero) data, the cursor moves to that screen field where data entry is required.

Note: The program has ultimate control over the order of field input and determines which groups of fields can be entered; therefore, adding this attribute might not always accomplish the desired result.

Reverse

Causes the field to appear with highlighting (e.g., reverse video), depending on the terminal capabilities.

Right or Rightj

Makes a field right justified in the field display. The system does justification on character fields when the field is input rather than when it is output. This causes the data received by the program to be right justified. SCR assumes that character data is already justified properly for output; therefore, SCR performs no right justification when the field is displayed. This feature allows program screens to specify the format of character data as it exists in the database.

Note: The system performs all justification of numeric fields when the field is output rather than when it is input since numeric data is always stored the same way in the database.

Scroll

Allows a group of fields on the screen to scroll data values. The scroll attribute field list contains one or more field names separated by commas inside parentheses. Field names in the list are screen field names defined in the screen section or earlier in the attributes section (i.e. name1, name2, ... nameN). Because it is very closely connected to the inner workings of the program, you should not add or remove the attribute from a screen field without checking how it is used in the program.

Example: scroll = (id1,id2,id3,id4)

Note: The scrolling capability of screen fields allows more than one record of data to be displayed on the screen at a time. All attributes assigned to the scroll group apply to all fields within the scroll list. This allows the screen section to contain each screen field while the attributes section only contains the scroll field with all appropriate attributes assigned to it. For example, if the scroll field is numeric a format attribute could be specified, causing all fields in the list to be displayed in the given format.

Text="string"

Sets a field buffer to "string"; thus, binding a buffer to that field.

Upshift

Converts all lower-case letters to upper-case when the field is input.

Note: Data containing lower-case letters may be displayed in the field, but data will be up shifted if you modify the data.

Verify

Specifies that the field should be entered twice before accepting the data. SCR does not handle this feature internally.

Void

Allows multiple pages of the same form to void specific fields until the last page is printed. For example, if you print payroll checks and an employee's check stub contains more deductions than will fit on one form, the system prints multiple forms with the check amount included only on the last form. If you specify the void attribute for the check amount field, all forms before the last will contain the value of the default attribute.

Example: ckamt: format="\$,\$\$,,\$\$,,\$\$&.&&", default="*****VOID****", void;

Note: The void attribute is only used for forms.

The void attribute is only used if the voiding instruction is given in the instructions section (See Voiding).

In this special use of default, you can use a character string in a numeric field. For example, the default value could be *****VOID****, if the check amount field is large enough to hold the message.

Zerof or Zerofill

Displays a numeric field with leading zeros. The result is a right justified number with all unused screen field positions to the left of the number containing zeros.

Instruction Section Format

The format of lines in the instructions section is as follows:

Example: instruction = value;
instruction;

Note: All instruction lines must end with a semicolon (;). The white space (space, tab, newline) is always optional around the punctuation characters (equal sign, semicolon).

Instructions You Can Specify

The following are the instructions you can specify in the instructions section of the screen definition file.

CAUTION: SCR screens differ greatly from PERFORM screens in the instructions section. Do not attempt to use any instructions other than those described below.

Alias

Allows the data from one field to be displayed in more than one location on the screen. The instruction's primary purpose is to allow forms the ability to duplicate data for forms, such as check stubs and return forms.

- The alias name (on the left-hand side of the equal sign) must be a screen field name either defined in the screen section or listed in the attributes section as optional.
- The original field name (on the right-hand side of the equal sign) may be either a screen field name defined in the screen or attributes section or a database field name assigned in the attributes section.
- Neither the alias name nor the original field name may be a group or scroll field, i.e. they **MUST NOT** contain the group or scroll attribute. The alias name will receive all attributes assigned to the original field name unless that attribute is already specified for the alias in the attributes section. Following is an example:

```
Form
{
ID Number...[id1   ]   [id2   ]   [id3   ]
}
attributes
id1 = id_no, format="#####";
id3: format="&&&&&&&";
instructions
alias id2=id1;
alias id3=id1;
end
```

Note: In the example, note the following:

- The alias field id2 is identical to id1. The only difference between id3 and the other two fields is the display format.
- If the ID number is zero, both id1 and id2 will display blank, while id3 will display seven zeros.
- In both alias instructions, the field id1 could be replaced with id_no, the database field name.
- In the alias of id3, the field id1 could be replaced with id2. The database name, as in the example, should only be given for the original field and not for the aliases. Repeating the database name will cause an error during translation.

Alignment

Specifies that FPS allows the user to verify the placement of the printed data on the form. This instruction gives you opportunity to ensure that the data will fit into the predetermined locations on printed forms.

Note: This instruction is only used for forms.

Formtype

Defines the name for the form as it will be used by FPS. The name assigned as the form type should not be quoted.

Example: formtype = grade reports;
formtype = prcheck;

Note: This instruction does not apply to screens.

Number

Defines a screen field name that contains the physical form number printed by FPS. As each form prints, FPS increments the physical form number and prints it in the specified field.

Example: number = ckno;
number = receipt;

Note: This instruction is only used for forms.

Tracking

Causes FPS to create tracking information when printing the spooled job. The tracking information records:

- The voided forms
- The printed forms
- The logical data record printed on each physical form.

Other programs can record this information in the database.

Note: This instruction is only used for forms.

Skip

Causes FPS to skip one blank line after every other form as it is printed.

Note: Some printed form stock has perforations that are not evenly spaced, or two forms can be combined on one page where the space between them is one line less than the space between the pages.

This instruction is only used for forms.

Voiding

Causes the void attribute to take effect.

Note: If you omit the void attribute, the system ignores the voiding instruction

This instruction is only used for forms.

SECTION 11 – REPORTS AND OUTPUT CONTROL

Overview

Introduction

This section describes the design of CX reports and output control. CX uses the INFORMIX report writer to create reports. This section provides information on ACE reports:

- Using ACE report commands
- ACE report formatting
- CX enhancements to ACE Reports (*acearray* functions and the *runreport* script)
- Tips for improving the design of ACE reports

This section also describes CX print spooling software for printing reports or other output. The print spooling software provides:

- Output device control
- Spool queue management

ACE Reports Sorting Program

Jenzabar created the Sortpage program (*sortpage*) that you can use to re-sort ACE report output when records get out of their original order. This change in ordering primarily happens because of the use of *adr* on the ACE report output. *Sortpage* uses the UNIX sort utility and values supplied by ACE to do the sorting. See the *Sortpage Program* in the *Common Programs* section of this manual for more information.

ACE Report Writer Commands

Introduction

These pages describe the commands that you can specify in an ACE report.

Running an ACE Report

Use the following commands to run an ACE report.

saceprep {reportname}

This command creates an operating system file *reportname.arc* containing the ACE commands. It does this by compiling the *reportname* file to create a platform independent file that the sacego interpreter can run.

Note: Any errors that occur when running aceprep are syntactical errors, not logic or read errors.

sacego {reportname}

This command runs the ACE report with runtime messages. These messages inform the user which select statement is being processed.

Note: Use this command to test the report for select statements. If any errors occur during the read or formatting process, you will find them at this time.

sacego -q {reportname}

This command suppresses the runtime messages that precede the standard output of the ACE report.

ACE Commands

The ACE report source has three mandatory sections:

- DATABASE
- SELECT
- FORMAT

All other sections are optional.

The following are the commands and the syntax used by the ACE report writer.

DATABASE

```
database CARS_DB end
```

Note: The DATABASE command defines the name of the database that is being accessed. Using CARS_DB utilizes the database name in your environment variable CARSDB. This is particularly useful when your system has more than one database. Once compiled, the value of CARSDB is fixed in the .arc file. However, the run reports script uses the CARSDB variable to override the value compiled into the program. Reports run from the shell or from a script not using this feature will use the compiled value for the database name.

DEFINE

```
define={function name}  
    ={variable name type}  
    ={param[number] name type}  
end
```

Note: Some functions and their definitions are the following:

- `_getcars` - Use to call environmental variables.
- `_midstring` - Use to center a piece of text.
- `_full_name` - Use to extract the first name from the `id_rec` name field.
- `_last_name` - Use with array of days of the week.
- `_first_name` - Justifies three parts of the line (left, center, right)
- `_dashdays` - Formats the RCS header and source lines.
- `_toupper` - Changes lowercase to uppercase.

INPUT

Prompt for variable-name using "string > "

OUTPUT

page length N {default=66}
right margin N {default=132, but not enforced; only used in conjunction with word wrap}
left margin N {default=5}
top margin N {default=3}
bottom margin N {default=3}
report to "filename"
report to pipe "prntername" {default to CRT}

SELECT

```
select [*] from filename [whereclause] [orderbyclause] end  
select field-name [whereclause] [orderbyclause] into temp tmpfile end  
read "filename" delimiter ":" order by ordfield end
```

Note: The read command causes the ACE report to get its data from an ASCII file, but act as if the data came from the database.

ORDER BY {maximum of 8 sort fields}

order by fieldname [ascending,descending] [fieldname]

FORMAT

The Format section is used to print the accessed data and may contain the following subsections:

first page header

Note: Executed only once to place the header on the first page. One-time variable initializations should go here.

page header

Note: This subsection is executed to place a header on every page after the first if there is a first page header section. If there is no first page header section, then it is executed for the first page also.

before group of {sortfield}

on every record

after group of {sortfield}

page trailer

on last record

The following commands may be used in any of the subsections of the Format section.

print [using "###.##"]

let {variable} = {value}

if-then-[else] (may be nested)

while

WHILE EXPRESSION DO STATEMENT

FOR VAR = EXP TO EXP [STEP EXP] DO STATEMENT

Aggregates

percent of
[group] count
[group] total of
[group] average of
[group] min of
[group] max of

Other

pageno (prints current page number)
date (prints current date in format Day, Month, Year)
time (prints time report is run in format HH:MM:SS with HH = 1-24)

Defining Variables and Functions

You provide definitions to variables and functions in the DEFINE section of the ACE report. This section comes after the DATABASE section and before the SELECT section. The following provides the syntax for defining variables and functions.

Defining Variables

```
define
    variable counter type integer
    variable special total type double
    variable selection value type char(10)
end
```

Defining Command Line Variables

```
define
    param[1] salary type double
    param[2] daysworked type integer
    param[3] empname type character length 20
end
```

Defining Functions

```
define
    function _getcars
    function _midstring
end
```

Note: The variable associated with param[1] will contain the first value on the command line that occurs after the name of the ACE report that is being run.

Defining For Input Variables:

```
define
    variable input item type character length 20
    variable input quantity type integer
end

    input prompt for inputitem using "Type item for selection " prompt for inputquantity
    using "Type quantity for selection "
end
```

Information Macros in the Define Section

You can specify macros in the define section of the ACE report, including:

- The `REP_DEFLOC' macro, which tells the location of the report.
- The `REP_DEFREV' macro, which tells the revision information and status (including the last date and time revised, who revised it.)

The following is an example define section:

```
define
    param[1] inputno                type long
    REP_DEFINE
    REP_DEFLOC( $$ )
    REP_DEFREV( $$ )
end
```

An example revision number for an ACE report is 6.1.4.1, where:

- 6 = release number
- 1 = CX revision number
- 4 = CX client number
- 1 = local client revision number

Output Commands

You specify the page specifications and margins for the ACE report in the Output section. The section comes after the DATABASE and DEFINE sections and before the READ section.

Default values

```
top margin = 3
bottom margin = 3
left margin = 5
right margin = 132
page length = 66
output to = CRT
```

Example for Changing Default Values:

```
output
  page length 24 {typical CRT size}
  right margin 80 {typical CRT size}
  left margin 0
  top margin 2
  bottom margin 1
  report to "ace_output" {write report to a file}
end
```

Example to send report directly to a line printer:

```
output
  report to printer
  OR
  report to pipe "lpr"
end
```

Print Commands

You can use the print statement to print quoted strings such as characters, character fields, arithmetic fields, and expressions.

Examples of print statements are:

- print name
- print name, id_no
- print name, " ",id_no," ",state
- print "The name is", name, "."
- print x
- print temp

Examples of print statements without trailing blanks are:

- print name without trailing blanks
- print city clipped

Aggregate Commands

You specify an aggregate, a total or gross amount, using the following commands:

- Count
- Percent
- Total
- Average
- Group Count (used within AFTER GROUP OF clauses)

Examples of aggregate commands are:

```
print "There were ", count using "#####", "attendees at the"
print "meeting this year."
print
print "We expect ",2 * count using "#####", "attendees at the"
print "meeting next year."

after group of lead_city
print group count using "#####"
```

Pause Command

You use the pause command to stop the report processing for a user's response. The format for the command is as follows: pause "textstring"

Example: pause "Type a carriage return to continue."

Skip Commands

SKIP n LINES

Used to create blank lines, rather than using print.

SKIP TO TOP OF PAGE

Used to return control of the report to the page header upon specified conditions

Example ACE Reports

Introduction

The following reports provide simple examples for specifying database reads and the sorting of information.

Report to Print All Names in the Database

To print all names in the database, you specify the following simple statements.

Example: database cars end

```
select fullname from id_rec end
```

```
format every record end
```

1. Specify the database: database {cars} end
2. Specify what to read: select {fieldnames} from {table names} end
3. Specify to print the data (format): do default format for every record end

Select and Sort Report

To specify a select and order by clause in an ACE report, do the following.

1. Specify the Select statement:
select fieldlist from [tables][where clause] [order by clause][into temp x] endsymbol
2. The parameters that you specify are:

x

Any temporary file that is used to hold data during the reading of the database before formatting begins. It is required if any testing or calculations of the data read is to occur.

fieldlist

One or more fieldnames separated by commas.

tables

One or more table names separated by commas.

where clause

The keyword 'WHERE' followed by a Boolean-expression.

Note: You can express the Boolean-expression as a *field relop constant* (relop is a relational operator: =, <>, <, >, <=, >=), or as *logop Boolean-expression* (logop is one of the logical operators: AND, OR)

endsymbol

Either the word *end* or a semi-colon.

3. Specify the ORDER BY clause:

```
order by fieldnames [ascending|descending] end
```

Note: You can combine the use of ascending and descending within the same sort clause. Any field listed in the sort clause must be an indexed field within the database.

Example: sort by name ascending state descending end

Example SELECT and ORDER BY Reports

The following are example ACE reports that contain SELECT and ORDER BY clauses.

```
database cars end
select id_number,
       fullname,
       state
from id_rec
order by fullname end
format every record end
```

```
database cars end
select id_number,
       name,
       state,
       sex
where ((state = "OH" or
        state = "MA") and
        (id_number = profile_id))
order by fullname, state descending end
format every record end
```

Formatting ACE Reports

Introduction

The following specifies the command clauses you can specify in the FORMAT section of an ACE report.

FORMAT Command Clauses

The format command is composed of combinations of up to 7 clauses. All of the clauses are optional. However, you must specify at least one clause.

FIRST PAGE HEADER

Statements contained in this clause will be executed immediately after the top margin of the first page is printed.

PAGE HEADER

Statements contained in this clause will be executed immediately after the top margin is printed on every page. If a first header page is present, this clause is not active until the second page.

BEFORE GROUP OF

Can only be used when using a sort clause. This format clause is followed by one of the fieldnames in the sort clause. The statement in this clause will be executed immediately before any new value occurs in the fieldnames specified.

ON EVERY RECORD {fieldname}

Used for processing, printing, and calculating information on every record that was selected.

AFTER GROUP OF {fieldname}

Can only be used when using a sort clause. Used in a similar fashion to BEFORE GROUP OF, except it occurs after every new value in the fieldname specified.

ON LAST RECORD

Similar to AFTER GROUP OF, except statements are executed after the last record has been processed. For the last record, the order of action is: BEFORE GROUP OF, ON EVERY RECORD, AFTER GROUP OF, and ON LAST RECORD.

PAGE TRAILER

Statements contained in this clause will be executed at the bottom of each page, just above the bottom margin.

Page Headers

In page headers, print the parameters that are passed to the ACE report, not the actual field values. If you use the actual field values, and the system does not choose any records, the field values print on the report as blank or zeros.

For example, in a report that prints the session and year, use the following in the page header (where sess and yr are parameters defined in the Define section.):

```
'print sess clipped, 1 space, yr using "####",
```

Do not use the following:

```
'print adm_plan_enr_sess clipped, 1 space,  
adm_plan_enr_yr using "####"
```

Note: In page headers, use the 'REP_JUSTIFY' macro or the '_midstring(text)' function for centering text.

Page Trailers

ACE reports in CX have footers generated by code matching this format:

```
on last record
    REP_LAST_REC
page trailer
    REP_TRAILER
```

The REP_TRAILER macro expands to code which will produce the path of the ACE report at the bottom of the last page of the output.

Note: If you use an IF-THEN-ELSE statement to put a different footer on all of the pages except the last, then the number of lines printed by the IF part must be the same as that in the ELSE part. This is an ACE requirement.

ON LAST RECORD Statements

The following are the statements that you can specify that are executed when the last record is processed.

LET STATEMENT

Assignment to variables that were defined in the DEFINE command.

Example: let runningcount = runningcount + 1

IF-THEN-ELSE STATEMENTS:

FORMAT = If condition then action end

Note: You only need to use "begin - end" if there are two or more executable formatting statements within the "if-then" statement.

Note: Jenzabar recommends that *if* and *else* line up, and that *begin* and *end* line up for readability.

Example: if (salesprice > average of salesprice) then

```
    begin
        print "Above average."
    end
if (salesprice > average of salesprice) then
    print "Above average."
else
    print "Below average."
end
```

Troubleshooting ACE Reports

Introduction

The following is a set of suggestions which may help in debugging Ace reports.

Apparent problem with data

In this case the report output appears to be wrong with no indication of the cause. This is a common problem with summary type reports since the data leading to totals is not visible on the output. If the Ace report is an old one, the problem is probably the data. On the other hand if the Ace report is just being written, then the logic or select statements may be incorrect. Two standard techniques are useful in this case.

a) Cut out the select portion of the report using the editor and put it in a file. Replace the parameter variables with suitable literal constants. Then use this file as input to isql (or dbaccess). This step should give the actual data rows which were available to the Ace report.

b) Put a section in the “on every row” section to produce a formatted output for each data row. It is a good idea to leave this section in the Ace report when done so that if a change is desired, the debugging code is already there. This may be done in two different ways. The first is to just comment out the debugging code. Of course this method precludes there being any comments in the debugging code since comments may not be nested in Ace. The second method makes use of the fact that the Ace report is passed through the m4 macro processor when it is translated. To use this method, place the following line of code at the beginning of the Ace report:

```
m4_define(`DEBUG', `Y')
```

Then for each group of lines of debugging code use the following syntax:

```
m4_keepif(DEBUG, `Y')  
  
    some lines of debugging code  
  
m4_keepend
```

In use, the Y in the define is changed to N prior to checkin so the debugging code is not in the production report. However, if a change is to be made in the report, then after check out, the N is changed back to a Y for debugging.

Core dump when translating the report

This problem is usually due to one of two problems. The first is that the Ace has become too large. The second is that the Ace report is a new one and the problem is caused by the REP_DEFLOC macros. There are techniques to handle both of these.

a) Report too large. Ace has not been updated very much since Informix version 3.3. In fact when they went from isql to dbaccess the main change was that the direct support for Ace reports and Perform screens that existed in isql was dropped. Many severe hard-coded limitations exist in Ace. For example, there is a limitation of 100 variables in the Ace report. Other limitations relate to other areas; for example, there appears to be a limitation on the number of temp tables that may appear in the select statements. Unfortunately the preparation program, saceprep, does not effectively handle reports that become too large.

Many times it just gives up with a core dump. The error message may be segmentation fault or bus error from the operating system.

If this error occurs then either one of the built-in limitations has been reached, or the report code has just become too large. In either case, an approach is to selectively remove code until the Ace report will compile. One method to do this is to comment out regions. However, the technique shown above using the m4 macros is actually better especially if you have been placing comments in the code. Selected regions may be surrounded by the m4_keepif and m4_keepeof lines, or a “binary search” technique may be used to locate the region with the problem.

b) The REP_DEFLOC macros are used to create the line in the trailer on the last page which identifies the Ace report source path so that a user may readily identify a report for someone who wishes to make a change or fix a problem. Unfortunately, there are conditions under which this macro will cause a core dump. A simple fix is to comment out these macros, or to check in the report as soon as it will compile cleanly so that the macro has a defined path.

Core dump when the Ace report is run

Unfortunately the saceprep program is not a good “compiler” so many actual program errors will not be found until runtime. This error may also be due to a report becoming too large. Another cause is improper string manipulation. If the report steps through a string looking for a substring and in doing so steps off the end of the string, then this movement usually causes a segmentation fault.

If the problem is found to be that the Ace report has become too large, there are two techniques for resolution. Both techniques involve breaking the report into multiple reports. The first Ace report only collects the data and stores it in large arrays using the acearray functions. Then in the “on last row” clause, the arrays are dumped out to be used as input for another Ace report. The difference between the two methods lies in the location where the data is stored.

a) Now that the Ace report may execute an SQL statement directly to store data, the intermediate data may be stored in a table (see later section on SQL functions). The follow-on Ace report(s) may then select data from this intermediate table. This table may be a standard database table with a schema, or it could be a “temporary” table created on the fly by the _exec_sql() function. Unlike the normal temp table created in the select statement, this table will not disappear when the Ace report terminates.

b) Ace reports may read data from an ascii file as if a select statement had been used. In this case, pseudo fields must be defined so the report knows how to deal with them since the report cannot get this type data from the database as it normally would. Then a read statement is executed which brings the data in from the ascii file. The data needs to be stored in the file in a delimited format similar to unload format. The delimiting character may be specified so that one interesting approach is to use a “,” (comma) to delimit the fields in the ascii file. This way the data could be loaded directly into a spreadsheet or other utility that can read CDF files in addition to being used by the follow-on report.

Acearray Functions in ACE Reports

Introduction

CX has expanded the capability of the ACE report writer by adding functions to it. These functions include:

- Functions that interact with the RCS system.
- A set of functions allowing you to greatly exceed the normal limit on the number of variables.
- Functions that add arrays to ace

The following pages describe how to use the latter two types of functions, called *acearray* functions. These functions all have a name starting with `_var`, which distinguishes them from the other CARS-added functions.

Access to Acearray Functions

The *acearray* functions are C language functions and are supplied to all clients in the directory: `$CARSPATH/src/common/sacego`

Since ACE is supplied as an object library, the actual `sacego` executable is created by the command `make install` in the above directory.

Summary List of Acearray Functions

The following briefly describes the functions and their usage.

`_vardef`

Defines a variable or an array of variables

`_varget`

Gets data from a previously defined variable

`_variget`

Gets data from a variable using an index variable (a loop variable)

`_varstore`

Stores data into a variable

`_varistore`

Stores data into a variable using an index variable (element defined by a variable)

`_varaccum`

Accumulates data into a variable using several methods

`_variaccum`

Accumulates data into a variable specified by an index using several methods

`_varpct`

Calculates a percentage from two parameters passed and correctly handles the case of divide by zero

`_varpctold`

Calculates a percentage from two parameters passed (old version of `_varpct`)

Use of the Acearray Functions

CX added the *acearray* functions to gain additional variables that can be used in the ACE report. By also adding arrays of variables, CX made it possible to have several thousand variables in the ACE report. You must define each variable and then data can be stored or received in the variable. When defining each variable, you must give a name to each variable (or array of variables). For example, you could define a variable called *somevar*. Alternatively, you could define an array of 100 such variables which you could call *somevar*[100].

Note: The string *somevar* is referred to as the *tagname*. In the following descriptions, N refers to the number of elements in an array, while n, n1, and n2 refer to some particular element in the array.

Because of the manner in which ACE deals with variables internally, most of these functions work with most of the datatypes that are allowed in ACE. They may or may not work with the newest data types such as datetime, interval, and varchar. They were initially written before these types were available and this update did not address that issue.

Note: Variable tagnames are case sensitive.

Acearray Function Variables

The following lists the *acearray* functions and their arguments.

_vardef("name", "type", value)

name: the variable tagname optionally followed by [N]
type: the type of value ("char", "integer", "money", ...) - desired
value: a type-similar initializer for all elements of name

_varstore("name", value)

name: the variable tagname optionally followed by [n]
value: a type-similar value to store in the appropriate element(s) of name

Note: If "name" was defined as an array and this call does not include a specific index, the value will be stored in *all* locations.

_varistore("name", index, value)

name: the variable tagname without any [n] specifier
index: the 1 relative index into the name array (1 <= index <= N)
value: the type-similar value to store in name[index]

_varget("name")

name: the variable tagname, followed by [n] if name is an array

_variget("name", index)

name: the variable tagname without any [n] specifier
index: the 1 relative index into the name array (1 <= index <= N)

_varaccum("name", {factor | "+" | "-" | "*" | "/"}, {"array" | value})

name: the variable tagname optionally followed by [n] or [n1-n2]
factor: a numeric multiplier or an operator {"+", "-", "*", or "/"}
array: a variable tagname optionally followed by [n] or [n1-n2]
value: the type-similar value to accumulate into each element of name

_variaccum("name",{factor | "+" | "-" | "*" | "/"}, {"array" | value}, index)
name: the destination variable tagname optionally followed by [n] or [nl-n2]
factor: a numeric multiplier or an operator {"+", "-", "*", or "/"}
array: a source variable tagname optionally followed by [n] or [nl-n2]
value: a type-similar value to accumulate into each element of name
index: an integer value which specifies the element of "name" desired.

_varpct(subtotal, total) or _varpct("array1[n]", "array2[m]")
subtotal: a numeric value dividend
total: a numeric value divisor

Note: This version of `_varpct` handles overflow.

_varpctold(subtotal, total)
subtotal: a numeric value dividend
total: a numeric value divisor

Note: This is the old version of `_varpct` that did not handle overflow well.

The `_vardef` Function

This function defines a variable or an array of variables for an ACE report and requires three parameters. The first two must be character strings.

- The first parameter identifies the name of the variable and, if it is to be an array, its array dimension is a positive integer surrounded by brackets.
- The second parameter designates the type of the variable. Allowable variable types are "smallint", "integer", "float", "smallfloat", "char", "decimal", "money" and "date". Type specifiers must be in lower case as shown here since they will have a string compare operation performed by the functions to determine the variable's type.
- The third parameter is the initial value to be stored in each element of the variable. Its type must be similar to the defined type of the variable. The call to this function normally will go into the first page header section of the ACE report so that it will only be called once. These variables are not known by ACE which is why you can circumvent the ACE limit.

Note: Defining a variable which has already been defined during this run of the ACE report results in an error.

`_vardef` Examples

Some examples of `_vardef` usage are as follows:

let ret_value = _vardef("student", "char", " ")
Defines a variable called "student" which can hold ANY size character string. It is initialized to contain a single blank.

let ret_value = _vardef("student[10]", "char", " ")
Defines a character array of ten elements (from 1 through 10), each of which may store a different size string.

let ret_value = _vardef("num_students[10]", "integer", 0)
Defines an integer array which has a tagname of `num_students`. This array can hold up to ten integers all of which are initialized to zero. The individual elements are `num_students[1]...num_students[10]`.

The `_varstore` Function

This function places data into a variable previously defined by `_vardef` and requires two parameters.

- The first parameter must be a character string identifying the variable. If the variable was defined as an array of size N, the name usually will include an index into the array as an integer surrounded by brackets. The index (of type integer) must range inclusively from 1 to N.
- The second parameter is the data to be stored into the variable and must be of a type similar to that which was used in the `_vardef` statement.

`_varstore` Examples

Some examples of `_varstore` usage are as follows:

```
let ret_value = _varstore("student", "Smith, John")
```

Stores the string "Smith, John" into the variable "student"

```
let ret_value = _varstore("num_students[3]", 25)
```

Stores the integer 25 into the array "num_students" in element 3, which is the third element.

```
let ret_val = _varstore("num_students", 15)
```

Stores the integer 15 into the array "num_students" in ALL 10 locations.

The `_varstore` Function

This function places data into an array previously defined by `_vardef` and requires three parameters. Its purpose is to allow one to store values into an array while in a loop by using the value of an integer variable to determine the element of the array to use.

- The first parameter must be a character string identifying the name of the variable (tagname).
- The second parameter must be a numeric value. If the variable was defined as an array of size N, the value must range inclusively from 1 to N. If the variable was not defined as an array then a value of 1 is valid; however, you should use the `varstore` function in that case. This value will usually be supplied by a variable rather than a literal constant.
- The third parameter is the data to be stored into the variable and must be of a type similar to that which was used in the `_vardef` call for this array.

`_varstore` Examples

Some examples of `_varstore` usage are as follows:

```
let ret_value = _varstore("student", 1, "Smith, John")
```

Stores the string "Smith, John" into the variable "student." (`_varstore` should probably have been used.)

```
let ret_value = _varstore("num_students", 3, 25)
```

Stores the integer 25 into the array "num_students" in element 3, as does the following two lines:

```
let index = 3
```

```
let ret_value = _varstore("num_students", index, 25)
```

The `_varget` Function

This function retrieves data from a variable previously defined by `_vardef`. It requires one parameter, a character string that identifies the name of the variable. If the variable was defined as an array of size N, the name must include an index into the array as an integer surrounded by brackets. The integer index must range inclusively from 1 to N.

_variget Examples

Some examples of `_variget` usage are as follows:

```
let char_var = _variget("student")
```

Retrieves character data from the variable "student"

```
let int_var = _variget ("num_students[3]")
```

Retrieves integer data from the array "num_students" from element 3.

The _variget Function

This function retrieves data from a variable previously defined by `vardef`. It requires two parameters.

- The first parameter must be a character string that identifies the desired variable which must have been previously defined with `_vardef`.
- The second parameter must be a numeric value. If the variable was defined as an array of size N, the value must range inclusively from 1 to N. If the variable was not defined as an array then a value of 1 is valid; however, the `_variget` function should be used instead in that case.

_variget Examples

Some examples of `_variget` usage are as follows:

```
let char_var = _variget("student", 1)
```

Retrieves character data from the variable "student" (Since "student" is not an array the `_variget` function should be used. It is more efficient since `variget` is just a wrapper around `_variget`.)

```
let int_var = _variget ("num_students", 3)
```

Retrieves integer data from the array "num_students" from element 3. This action will more often be accomplished by the following statements:

```
let index = 3
```

```
let int_var = _variget("num_students", index)
```

The _varaccum Function

This function does either:

- Combines (add, subtract, multiply, or divide) two arrays element by element
- Combines (add, subtract, multiply, or divide) a constant with all elements of an array

This function requires three parameters.

- The first parameter is always a character string and must contain the name of a non-character type array previously defined by `_vardef`. If all elements of the array are to be affected, then no element qualifier is necessary. Otherwise, a qualifier must directly follow the array name in the form. This qualifier can be of the form `[n]` or `[n-m]`, where `n` and `m` are both within the range of 1 to N inclusively.
- The second parameter is either a string representing the operation ("`+`", "`-`", "`*`", or "`/`") to be performed or a factor that will be multiplied to the third parameter value before being combined via the desired operator with each value specified by the first parameter.
- The third parameter may be either a character string name with the same characteristics of the first parameter (including number of elements specified) or it may be a constant numeric (non-character) type value.

_varaccum Examples

Some examples of `_varaccum` usage are as follows:

Note: In the following examples, assume that `Arr1` and `Arr2` are arrays that have both been defined by a `_vardef` and that both have 10 elements. Likewise, `Arr3` is an array defined by `_vardef` which has 15 elements. All three contain the same type of data.

let int_var = _varaccum("Arr1", "+", "Arr2")

The above statement accomplishes the following actions:

```
Arr1[1] = Arr1[1] + Arr2[1]
Arr1[2] = Arr1[2] + Arr2[2]
Arr1[3] = Arr1[3] + Arr2[3]
.
.
.
Arr1[10] = Arr1[10] + Arr2[10]
```

let int_var = _varaccum("Arr1", "*", "Arr2")

The above statement accomplishes the following actions:

```
Arr1[1] = Arr1[1] * Arr2[1]
Arr1[2] = Arr1[2] * Arr2[2]
Arr1[3] = Arr1[3] * Arr2[3]
.
.
.
Arr1[10] = Arr1[10] * Arr2[10]
```

let int_var = _varaccum("Arr1", "/", "Arr2")

The above statement accomplishes the following actions:

```
Arr1[1] = Arr1[1] / Arr2[1]
Arr1[2] = Arr1[2] / Arr2[2]
Arr1[3] = Arr1[3] / Arr2[3]
.
.
.
Arr1[10] = Arr1[10] / Arr2[10]
```

let int_var = _varaccum("Arr1", "-1", "Arr2")

The above statement accomplishes the following actions:

```
Arr1[1] = (-1 * Arr2[1])
Arr1[2] = (-1 * Arr2[2])
Arr1[3] = (-1 * Arr2[3])
.
.
.
Arr1[10] = (-1 * Arr2[10])
```

let int_var = _varaccum("Arr1[5-7]", "+", "Arr3[10-12]")

The above statement accomplishes the following actions:

```
Arr1[5] = Arr1[5] + Arr[10]
Arr1[6] = Arr1[6] + Arr[11]
Arr1[7] = Arr1[7] + Arr[12]
```

The `_variaccum` Function

This function is a version of `_varaccum` suitable for placing into a loop. Its relationship to `_varaccum` is similar to the one which `_variget` has to `_variget`. In addition to the same three parameters which `_varaccum` takes, `_variaccum` takes a fourth parameter, which is the value which specifies the element of each of the arrays which you should use.

Note: Because `_variaccum` only looks at the tag name of the arrays which may be present in parameters one and three, if these parameters contain an index value such as `[n]` it will be ignored

`_variaccum` Examples

Some examples of `_variaccum` usage are as follows:

let index = 3

let int_var = _variaccum("Arr1", "+", "Arr2", index)

The above statements have the following effect:

`Arr1[3] = Arr1[3] + Arr2[3]`

let index = 5

let int_var = _variaccum("Arr1", "1.05", "Arr2", index)

The above statements have the following effect:

`Arr1[5] = +(1.05 * Arr2[5])`

Note: These statements make the selected element of `Arr1` 5% larger than the corresponding element of `Arr2`.

The `_varpct` Function

This function is used to calculate percentages. It requires two parameters, both of which must be non-character type values. The result of $(value1 * 100)/value2$ is returned. If `value2` is zero, an extremely large number is returned, in fact: 999999.99. This value is returned if the value of the first parameter is $>1000*$ value of the second parameter. The reason is that this number will overflow the format specified for the return value so that the report will have asterisks on it at that location. This will alert the user that the number in that location is invalid. In the same situation `_varpctold`, returns a zero which the user may mistake for a valid result.

`_varpct` Examples

Some examples of `_varpct` usage are as follows:

let float_val = _varpct(30, 100)

Returns 30.

_varpct(subtotal, total)

Returns a percentage calculated by multiplying subtotal by 100 and then dividing by total

let float_val = varpct(30, 0)

Returns 999999.99.

The `_varpctold` Function

This function is used to calculate percentages. It requires two parameters, both of which must be non-character type values. The result of $(value1 * 100)/value2$ is returned. If `value2` is zero, then 0 is returned. It is only included in case anyone had built ACE reports which depended upon the previous behavior. It should not be used for new reports.

Note: This function is an older version of the `_varpct` function.

_varpctold Examples

Some examples of `_varpctold` usage are as follows:

let float_var = _varpct(30, 100)

Returns 30.

_varpct(subtotal, total)

Returns percentage calculated by multiplying subtotal by 100 and then dividing by total.

let float_var = _varpct(30,0)

Returns 0.

CAUTION: This result could be overlooked by the user. The `_varpct` works better in this case.

Troubleshooting Array Functions

Numerous error messages exist regarding syntax errors in the parameters passed to these routines, as well as error messages regarding their execution. The routines verify the definition of the variables by name and ensure that no variable is redefined, and that meaningless data assignments are not made. All messages are sent to the ACE report's standard out. Their format is similar to those produced by ACE itself.

Spurious error messages about undefined variables can appear when an actual case of an undefined variable occurs. This can be triggered by a typographical error in a call to `_varget`, for example. Fix the first problem and the others will disappear.

It is recommended that you add debug code on every row section of the report. This may produce a formatted output of the entire set of input data during debug. As an alternative, the section might be activated by a parameter passed by the user. In this way, the end-user may be able to search for data errors that would otherwise be hidden in a summary type of report. For an example of this type of debugging, see the sample report `setwaitrnk` at the end of the SQL Array Functions section.

Sample Report

The following is a sample ACE report to illustrate the usage of most of the array functions.

```
Revision Information (Automatically maintained by 'make' - DON'T CHANGE)
-----
$Header$
-----
}
database CARS_DB end

define
  variable tstvar1 integer
  variable tstvar2 integer
  variable i integer
  function _vardef
  function _varstore
  function _varistore
  function _varget
  function _variget
  function _variaccum

  REP_DEFINE
  REP_DEFLOC($Source$)
  REP_DEFREV($Header$)
end

output
  REP_OUTPUT
end

  {Simple select just to illustrate the operation of array functions.}
select
  id
from
  id_rec
where
  id = 1
end

format
  page header
  REP_FORMAT
  REP_HEADER("DEMO of CARS' ACE ARRAYS")

{ Need to define the "array variables" to be used. NOTE! Cannot do this in
define section because Ace does not know about these variables. }

  call _vardef("tstary[5]", "integer", 0)
  call _vardef("test[5]", "char", " ")

{ In general if the user function does not return a value the call keyword
MUST be used. If the function does return a value then it
can be used anywhere that a constant could be used in an expression }

  on every row
  call _varstore("tstary[1]", 2)
  call _varstore("tstary[2]", 4)
  call _varstore("tstary[3]", 6)
  call _varistore("tstary", 4, 8)
  call _varistore("tstary", 5, 10)
  call _varstore("test[1]", "Test of length")

  let tstvar1 = _varget("tstary[1]")
  let tstvar2 = _variget("tstary", 2)

  print "tstvar1 = ", tstvar1 using "###"
  print "tstvar2 = ", tstvar2 using "###"
  {Note usage of temporary variable!}
  print "varget(tstary[1] = ", _varget("tstary[1]")
  print "varget(tstary[2] = ", _varget("tstary[2]")
  print "variget(tstary[3] = ", _variget("tstary", 3)
  print "variget(tstary[4] = ", _variget("tstary", 4)
  print "variget(tstary[5] = ", _variget("tstary", 5)
  print "variget(test[1] = ", _variget("test", 1)
  print "varget(test[1] = ", _varget("test[1]")

  for i = 1 to 5 do
  begin
  call _variaccum("tstary", "+", 1, i)
```

```

end

print
print "Results after using the _variaccum function:"
print "tstvar1 = ", tstvar1 using "###"
print "tstvar2 = ", tstvar2 using "###"
print "varget(tstary[1] = ", _varget("tstary[1]")
print "varget(tstary[2] = ", _varget("tstary[2]")
print "variget(tstary[3] = ", _variget("tstary", 3)
print "variget(tstary[4] = ", _variget("tstary", 4)
print "variget(tstary[5] = ", _variget("tstary", 5)
print "variget(test[1] = ", _variget("test", 1)
print "varget(test[1] = ", _varget("test[1]")

on last row
REP_LAST_REC

page trailer
REP_TRAILER

end
{

```

Sample Output

The following is a sample output of the ACE report shown above.

```

----- SAMPLE OUTPUT -----
Tue Jun 18 1996                CARS College                Page 1
20:49                        DEMO of CARS' ACE ARRAYS
tstvar1 = 2
tstvar2 = 4
varget(tstary[1] =          2
varget(tstary[2] =          4
variget(tstary[3] =          6
variget(tstary[4] =          8
variget(tstary[5] =         10
variget(test[1] = Test of length
varget(test[1] = Test of length

Results after using the _variaccum function:
tstvar1 = 2
tstvar2 = 4
varget(tstary[1] =          3
varget(tstary[2] =          5
variget(tstary[3] =          7
variget(tstary[4] =          9
variget(tstary[5] =         11
variget(test[1] = Test of length
varget(test[1] = Test of length

}

```


SQL Functions

Introduction

Jenzabar has added four functions that allow you to execute SQL statements within the ACE report itself. These functions provide the following benefits:

- Reports can have both a report and an update function.
- Reports do *not* need to produce SQL statements that are then piped to isql under the control of a C shell script.

The following are the four functions in this category.

The `_exec_sql` Function

This function allows you to execute an add or update type of SQL statement within the format section of the ACE report. It is not useful for executing a select statement.

Example: `_exec_sql("sqlstring", "sparm")`

The following are the parameters for this function.

sqlstring

A character variable containing the SQL statement to execute.

sparm

Two settings are possible:

- "S" if sqlstring involves the insertion of a serial value which you would like returned.
- " " if sqlstring does not involve the insertion of a serial value which you would like returned.

The `_ctrl_trans` Function

This function controls transactions.

Example: `_ctrl_trans("parm")`

Note: ACE does not open its cursor using the *with hold* clause, so you cannot use the `_ctrl_trans` function in the *on every row* section. The entire operation of the ACE report must be a single transaction. The value of the Informix error code is returned by these functions, so you can decide during execution whether to commit or rollback the work.

parm

Three values are possible:

- "B" for begin work
- "C" for commit work
- "R" for rollback work

Note: The call with a parameter of "B" is not needed since ACE executes this itself (for the temporary files).

The `_ctc_add` Function

This function allows you to add `ctc_recs` directly without having to create an SQL statement in the ACE code, and then pipe output to isql.

Example: `_ctc_add(id, tick, corr, due_date, time, compl_date, resrc, status, cgc, enrstat)`

The `_ctcdetl_add` Function

This function allows you to add `ctcdetl_recs` directly without having to create an SQL statement in the ACE code, and then pipe the output to `isql`.

Example: `_ctcdetl_add(ctc_no, sp_hndl, statnry, envl, sign_id, hnd_sgn, money1, money2)`

Sample Report

The following is a sample ACE report to illustrate the use of the array and SQL functions.

```

{
    setwaitrnk

    This report will be run from cron every night to update the admissions
    waiting list. This waiting list consists of those applicants which are
    on a waiting list for acceptance. Their current enrollment status will be
    WAITLIST (or some suitable macro value). When a student is taken off of the
    waiting list their enrollment status will become something else, for example
    ACCEPTED. This report will use the new features of sacego which allow one
    to update the database to update the ranks of the persons remaining on the
    waiting list so that there will be no gaps in the list. It will do so by
    reading the necessary data for all waiting list members into an array.
    It will then search this array for holes and move other members appropriately
    to fill in the holes.
}
{
Revision Information (Automatically maintained by 'make' - DON'T CHANGE)
-----
$Header$
-----
}
{
    If changes are made in this report and the build in debugging lines are
    desired then simply change the N to a Y in the define line below.
}
m4_define(`DEBUG', `N')

{ If it is desired to run the program without updating the database so that
test cases may be reused then the N should be changed to Y in the define line}
below}
m4_define(`TEST', `N')

database CARS_DB end

define
    param[1] param_prog      char(4)
    param[2] param_sess      char(4)
    param[3] param_year      smallint
    REP_DEFINE
    REP_DEFLOC($Source: /usr/carsdevi/modules/admit/reports/RCS/setwaitrnk,v $)
    REP_DEFREV($Header: setwaitrnk,v 8.3 97/04/04 13:28:54 rreno Developmental $)
    variable prev_rank      integer
    variable cur_rank integer
    variable rank_diff      integer
    variable cur_id         integer
    variable adm_prog char(4)
    variable i              integer
    variable j              integer
    variable debug          integer
    variable cur_name char(32)
    variable list_count      integer
    variable function_ret    integer
    variable sql_string      char(256)      {for sql function arg }
    variable quote          char(1)
    variable head_sess      char(4)

    function _vardef
    function _varistore
    function _variget
    function _varget
    function _variaccum
    function _ctrl_trans
    function _exec_sql

end

output
    REP_OUTPUT
end

select
    adm_rec.id,
    adm_rec.rank,
    adm_rec.prog,
    id_rec.fullname
from
    adm_rec,
    id_rec
where
    ((id_rec.id = adm_rec.id) and
    (adm_rec.enrstat = "ADM_STAT_WAIT") and
    (adm_rec.prog = $param_prog and

```

```

(adm_rec.plan_enr_sess = $param_sess or " " = $param_sess) and
adm_rec.plan_enr_yr = $param_year))

order by rank
end

format
{*****
  This section generates the header for the first page of the report
  and is the normal place to put one-time variable initializations. }

  first page header
  REP_FORMAT
  REP_HEADER("Make Waiting List Current")

  let rep_text = "Updates the Waiting List and Reports on It"
  print REP_JUSTIFY("", rep_text clipped, "")

  if (param_sess = " ") then let head_sess = "ALL "
  else let head_sess = param_sess
  let rep_text = "For program - ", param_prog, " Sess - ", head_sess,
    " and Year - ", param_year using "####"
  print REP_JUSTIFY("", rep_text clipped, "")

  {initialize variables}
  let list_count = 0
  let quote = ''

  {create arrays and initialize them}
  call _vardef("id_array[ADM_MAX_WAITLIST]", "integer", 0)
  call _vardef("rank_array[ADM_MAX_WAITLIST]", "integer", 0)
  call _vardef("org_rank_array[ADM_MAX_WAITLIST]", "integer", 0)
  call _vardef("name_array[ADM_MAX_WAITLIST]", "char", " ")
  call _vardef("prog_array[ADM_MAX_WAITLIST]", "char", " ")

  print
  print
  column 3, "cur",
  column 8, "prev",
  column 14, "student",
  column 29, "student",
  column 60, "result"

  print
  column 3, "rank",
  column 8, "rank",
  column 16, "id",
  column 31, "name",
  column 62, "code"

{*****}
  page header
  REP_FORMAT
  REP_HEADER("Make Waiting List Current")

  let rep_text = "Updates the Waiting List and Reports on It"
  print REP_JUSTIFY("", rep_text clipped, "")

  if (param_sess = " ") then let head_sess = "ALL "
  else let head_sess = param_sess
  let rep_text = "For program - ", param_prog, " Sess - ", head_sess,
    " and Year - ", param_year using "####"
  print REP_JUSTIFY("", rep_text clipped, "")

  print
  print
  column 3, "cur",
  column 8, "prev",
  column 15, "student",
  column 30, "student",
  column 60, "result"

  print
  column 3, "rank",
  column 8, "rank",
  column 16, "id",
  column 31, "name",
  column 62, "code"

```

```

{*****}
  on every row
    { In this section we simply gather the data and store it
      in the arrays.}

    let list_count = list_count + 1

    call _varistore("id_array", list_count, id)
    call _varistore("rank_array", list_count, rank)
    call _varistore("org_rank_array", list_count, rank)
    call _varistore("name_array", list_count, fullname)
    call _varistore("prog_array", list_count, prog)

    { For debugging purposes }
m4_keepif(DEBUG, `Y')
    print "index = ", list_count using "###&",
          " id = ", id using "#####&",
          " fullname = ", fullname,
          " rank = ", rank using "##&",
          "prog = ", prog
m4_keepend

{*****}
  on last row
    { In this case as happens frequently when using the array
      functions, all of the interesting work is done in the
      "on last row" section of the Ace report. The arrays are
      already sorted in rank order, but if there has been activity
      which has taken people off of the list then there will be
      holes in the list. We now scan the list and change the ranks
      to provide a continuous list. When this process is finished
      then we step through the list and update the ranks. As we
      step through the list if we find a hole in it ie. a difference
      greater than 1 then we move down the list increasing each rank
      (that is decreasing the rank value) by one less than this
      difference.

      As an example of this process the following shows the stages
      that would occur for the assumed starting rank_array values.

          begin      2nd step 3rd step 4th step  final      1      1
          3          1          1          1          1          1
          4          2          2          2          2          2
          7          5          3          3          3          3
          8          6          4          4          4          4
          10         8          6          5          5          5
          11         9          7          6          6          6
          17         15         13         12         7          7
          18         16         14         13         8          8

    }

    let prev_rank = _variget("rank_array[1]")

    { If the rank of the first one is not one then need to adjust
      all of them appropriately }
    if (prev_rank != 1)
    then
      begin
        let rank_diff = prev_rank - 1

        for i = 1 to list_count do
          begin
            call _variaccum("rank_array", "-", rank_diff, i)

          { for debugging purposes }
m4_keepif(DEBUG, `Y')
            let debug = _variget("rank_array", i)
            print ">>>> In first pass rank for ", i using "##&", " is ", debug
              using "##&"
m4_keepend

          end
        end

        { At this point the first rank is 1 and we need to check others}
        let prev_rank = 1
        for i = 2 to list_count do
          begin
            let cur_rank = _variget("rank_array", i)
            let rank_diff = cur_rank - prev_rank

```

```

{ for debugging purposes }
m4_keepif(DEBUG, `Y')
    print ">> For I= ", i using "##&", " cur = ", cur_rank using "##&",
        " prev = ", prev_rank using "##&," diff = ", rank_diff using "##&"
m4_keepend

    if (rank_diff > 1)
    then
        begin
            for j = i to list_count do
            begin
                call _variaccum("rank_array", "-", rank_diff - 1, j)

{ for debugging purposes }
m4_keepif(DEBUG, `Y')
                let debug = _variget("rank_array", j)
                print ">>>> For J= ", j using "##&", " cur = ", debug using
                    "##&"
m4_keepend

            end

            end

            let prev_rank = cur_rank - rank_diff + 1 { prepare for next pass }
            end

            { At this point the data for the new list is complete so
              we now proceed to update the database and print a report
              of our actions }
            for i = 1 to list_count do
            begin
                {First get the data for this student}
                let prev_rank = _variget("org_rank_array", i)
                let cur_rank = _variget("rank_array", i)
                let cur_id = _variget("id_array", i)
                let cur_name = _variget("name_array", i)
                let adm_prog = _variget("prog_array", i)

                { Create the sql string to perform the desired action }
                let sql_string = "update adm_rec set rank = ",
                    cur_rank using "##&",
                    " where prog = ", quote, adm_prog, quote,
                    " and id = ", cur_id using "#####&", ";"

{ For debugging purposes }
m4_keepif(DEBUG, `Y')
                print
                print "sql_string = ", sql_string
m4_keepend

m4_keepif(TEST, `N')
                if (cur_rank != prev_rank)
                then
                    begin
                        let function_ret = _exec_sql(sql_string, "")
                    end
m4_keepend

                print
                column 3, cur_rank using "##&",
                column 8, prev_rank using "##&",
                column 13, cur_id using "#####&",
                column 22, cur_name,
                column 60, function_ret using "----&"

            end

            {Need to commit the transactions in order to make our changes permanent}
            let function_ret = _ctrl_trans("C")
            print
            print
            print
            need 4 lines
            print "After committing the transactions the result was ", function_ret
            using "----&"

            print
            print "If this result is not zero then there was some problem!"
            print "In addition, the result code for each line should have been zero"

```

```
REP_LAST_REC
{*****}
  page trailer
  REP_TRAILER
end
```

Troubleshooting Use of SQL Functions

Introduction

Reports using the SQL functions are frequently complex and difficult to debug. Most such reports should contain debugging sections. For simple reports such as those that add contact records, such debugging code may only be needed when originally testing the report and later when making changes in it. In this case, the m4 macro technique illustrated in `setwaitrnk` shown above should be used. If the report is going to update important tables where accuracy should be verified first, then it is good practice to include on the menuopt a question such as “Final Pass (Y/N)” and pass the answer to the Ace report. If the answer is “N”, the Ace only prints a report of what it is going to do. If the answer is “Y”, the report may or may not print but will do the `_exec_sql()` statement. In this way you may do one or more verification runs before deciding to do a final pass where the update statements are done.

Security Setup with SQL functions

Prior to releasing the SQL functions, there was no real security concern with menu options that allowed an end-user to run an Ace report written by that user or someone else. However, now that the Ace report may cause changes in the database, the same concerns that motivate sites to block access to `isql` now also apply to the user Ace capability. The default setup for SQL functions blocks end-users (menu users) from access to the SQL functions. The RCS type of the Makefile in the `src/common/sacego` directory has been changed to “mult” from the old value of “prog”. Changes have been made in Makefile and a new file has been added to the makelist named `ro_ace_fn.c` which together cause two executables to be created by a `make translate` command, and both to be installed by one of the `install` commands. These two executables are named `sacego` (which has access to the SQL functions) and `rosacego` (which does not have access to the SQL functions). When a user runs an Ace report from a normal menuopt which executes an installed Ace report through the use of the script `runreports`, then `sacego` is used. However, if one of the two CX supplied menuopts which allow a user Ace to be run is used, `runreports` will utilize `rosacego`.

Note: If the institution personnel have created other methods to allow end-users to run their own Ace reports then these have to be examined for security implications.

Runreport Script: A Report Sorting Enhancement

Introduction

Jenzabar developed the *runreport* script to enhance the capabilities of the ACE Report Writer. The script provides the capability of specifying in a single ACE report various sorts on different field names or field values. The script makes use of variables in WHERE and/or SORT clauses, which are substituted by your entered values, to produce output sorted in different ways. Without the enhancement of *runreport* script, you would have to create multiple versions of the same report for each desired sorting scenario.

File Locations

All reports should be created in \$CARSPATH/<module>/others with the letters 'rpt' as the final three letters of the uninstalled source file name. The installed version will be located in OTH_PATH'<module> with the .oth extension.

Note: Unlike the situation of an installed module from a *reports* directory, one from an *others* directory is not compiled.

WHERE and SORT Clauses

The following are variables you use when specifying WHERE and SORT clauses.

SELECTFIELD

The standard variable name to be replaced by a database field name within a where clause. The # is a single digit to be used if more than one database field name is to be supplied by an operator through the menu system.

SELECTVALUE#

The standard variable name for a specific value of the field name entered for a field name. Use SELECTVALUE without a SELECTFIELD variable.

SORTFIELD#

Names for variables within the sort clause follow the standard, SORTFIELD#. As with SELECTFIELD, you can use more than one SORTFIELD where '#' is a single digit making each variable unique. You must include the SORTFIELD in the read statement and the SORTFIELD must be an indexed field within the database. Because ACE has a limit of eight sort fields, you can use only eight SORTFIELD variables within a sort clause..

Note: When specifying WHERE and SORT clauses, make sure that the values for SELECTFIELD, SELECTVALUE, and SORTFIELD are included in the report header so that the user knows the criteria used to produce the output.

Example WHERE and SORT Clauses

The following ACE report example incorporates variables within both the WHERE and SORT clauses. The example, which prints ACT or SAT scores of recruits, uses SORTFIELD in several areas throughout the report.

```
database cars end
define
    variable previd      type long
    variable prevdate    type date
    variable totalengl   type integer
    variable totalmath   type integer
    variable totalsoc    type integer
    variable totalnat    type integer
    variable totalcomp   type integer
    variable gcount      type integer
    variable runningcount type integer
    variable minvar1     type integer
    variable maxvar1     type integer
    variable minvar2     type integer
    variable maxvar2     type integer
    variable text        type character length 80
    param[1] inputsess   type character length 4   {enter academic session}
    param[2] inputyr     type integer             {year}
    param[3] status      type character length 8   {enrollment status}
    param[4] inputtype    type character length 8   {Type of exam}
    variable text        type character length 80
    REP_DEFINE
end
output
    REP_OUTPUT
end
read  into temp
      id_no
      name
      exam_date
      exam_score1 exam_score2 exam_score3 exam_score4 exam_score5
      SORTFIELD
joining adm_id = id_no
      and id_no = profile_id
      and adm_id = exam_id
where adm_cur_enrstat = status
      and adm_plan_enr_yr = inputyr
      and adm_plan_enr_sess = inputsess
      and SELECTFIELD = "SELECTVALUE"
      and (exam_type =inputtype and exam_score5 > 0)
end
sort by  SORTFIELD exam_score5 descending name id_no end
format
page header
    REP_HEADER
    REP_DATE
    REP_TIME
let text = status, " ", inputtype clipped, " SCORES"
print _midstring(text)
let text = "As of ",inputsess clipped,1 space,inputyr using "####"
print _midstring(text)
let text = "Where SELECTFIELD = SELECTVALUE and Sorted by SORTFIELD"
print _midstring(text)
skip 1 line
print "   Recruit                Date      Engl   Math   Natl   Socl   Comp"
print "-----"
      let totalengl = 0
      let totalmath = 0
      let totalsoc = 0
      let totalnat = 0
      let totalcomp = 0
      let runningcount = 0
before group of SORTFIELD
print SORTFIELD
  on every record
    if( id_no <> previd) or (id_no = previd and exam_date <> prevdate)
      then begin
print name,1 space, exam_date,4 spaces,exam_score1 using "###",
5 spaces, exam_score2 using "###", 6 spaces;
      if (inputtype = "ACT") then
        print exam_score3 using "##",5 space,exam_score4 using "##",4 spaces;
      else print 13 spaces;
print exam_score5 using "####"
      let totalengl = totalengl + exam_score1
      let totalmath = totalmath + exam_score2
      let totalsoc = totalsoc + exam_score4
      let totalnat = totalnat + exam_score3
```

```

let totalcomp = totalcomp + exam_score5
let runningcount = runningcount + 1
let gcount = gcount + 1
let previd = id_no
let prevdate = exam_date
end
on last record
print 44 spaces,"-----"
print" Min/Maximum Scores of each ",11 spaces;
if (inputtype = "ACT") then begin
  let minvar1 = min of exam_score1
  let maxvar1 = max of exam_score1
  print 5 spaces, minvar1 using "##","/",maxvar1 using "##",3 spaces;
  let minvar1 = min of exam_score2
  let maxvar1 = max of exam_score2
  print minvar1 using "##","/",maxvar1 using "##",3 spaces;
  let minvar1 = min of exam_score3
  let maxvar1 = max of exam_score3
  print minvar1 using "##","/",maxvar1 using "##",2 spaces;
  let minvar1 = min of exam_score4
  let maxvar1 = max of exam_score4
  print minvar1 using "##","/",maxvar1 using "##",3 spaces;
  let minvar1 = min of exam_score5
  let maxvar1 = max of exam_score5
  print minvar1 using "##","/",maxvar1 using "##"
end
else begin
  let minvar1 = min of exam_score1
  let maxvar1 = max of exam_score1
  print minvar1 using "###","/",maxvar1 using "###",2 spaces;
  let minvar1 = min of exam_score2
  let maxvar1 = max of exam_score2
  print minvar1 using "###","/",maxvar1 using "###",2 spaces;
  print 12 spaces; {skip scores 3 and 4 if not ACT}
  let minvar1 = min of exam_score5
  let maxvar1 = max of exam_score5
  print minvar1 using "###","/",maxvar1 using "###"
end
skip 1 line
print 7 spaces,"TOTAL RECRUITS: ", runningcount using "####"
  skip 5 lines
  REP_SOURCE($Source: /usr/carsdevi/modules/util/documents/ace/RCS/selectsort,v $)
  REP_REVISION($Header: selectsort,v 8.0 95/04/22 10:21:40 root Developmental $)
end

```

Runreport Script

The *runreport* script enhances ACE reports' use of WHERE and SORT clauses. The script does the following:

- Calls the report source and interprets the values for SELECTFIELD, SELECTVALUE, and SORTFIELD as entered by the user.
- Substitutes the variables with the user's entered values (or values within the where clause) using the m4 macro processor.
- Compiles the report.

Note: The system does not compile the report until the variable names have been substituted.

- Runs the report at the time specified by the user.

Note: Both the editing of the report source and the compiled version will be located in '/tmp' until the report execution has completed.

The *runreport* script is located in the following directory path: \$SCPPATH/common/runreports.scp. The following is the *runreport* script source.

```
#
# runreports: Script to execute an ACE report (csh)
#
# Parameters:
#   First are the optional '-' options:
#   -MSGS = Indicates the next parameter is a filename to be used
#           with the 'msg' routines for queuing status and error
#           messages.
#   -f Forrtype
#
#   The next parameter is the name of the ACE report.
#
#   The next set of parameters (the rest excluding the last) are
#   passed as arguments for the ACE report. Special parameters for
#   defining macros can be given as pairs of parameters of the form:
#   -Dmacro_name macro_value
#   If any such macro definitions are given, the file will translated
#   with the specified macro assignments and then executed.
#
#   The last parameter is the process to pipe output to (or 'file').
#   If 'file' is specified as the last parameter, output will
#   go to a file in the user's home directory. The file is named
#   with the name of the ACE report followed by '.out'. Also
#   a message is mailed to the user when the report is finished.
#
set Exitstat=1
#
#   Process -MSGS option
#
if ("$1" == "-MSGS") then
  set Msgs_file="$2"
  shift
  shift
else
  set Msgs_file=""
endif
if ($#argv < 2) then
  /bin/csh -f SCP_PATH/util/msg_queue.scp \line -s "ACE report could not run" -f
"$Msgs_file" \line -m "$0 Error: Too few parameters: $"
  exit 1
endif
#
#   Get Forrtype if specified
#
if ($1 == "-f") then
  set Forrtype="-f $2"
  shift
  shift
else
  set Forrtype=""
endif
#
#   Get name of report file
#
set Output=$argv[$#argv]
set Pathname=$1
shift
set Filename=$Pathname:t
set Root=$Filename:r
#
#   Prepare list of arguments for translation and execution
#
set Trans_args=()
set Exec_args=()
while ($#argv > 1)
  if ($1 =~ -D*) then
    set Trans_args=($Trans_args:q "$1=$2")
    shift
    shift
  else
    set Exec_args=($Exec_args:q $1:q)
    shift
  endif
end
```

```

#
# Translate the file if any macro definitions were specified
#
if (#Trans_args > 0) then
  set Exec_file=/tmp/$Root
  if (-e /tmp/$Root) then
    /bin/csh -f SCP_PATH/util/msg_queue.scp \line -s "ACE: Cannot run
'$Root' " -f "$Msgs_file" \line -m "The '$Root' report is already being run.
Try later."
    exit 1
  else
    onintr cleanup
    cp $Pathname /tmp/$Root
    cd /tmp # Work from /tmp
    csh -f MAK_PATH/user/arc/translate $Root $Trans_args >& $Root.out
    set Exitstat=$status
    if ($Exitstat != 0) then
      /bin/csh -f SCP_PATH/util/msg_queue.scp \line -s "ACE: Error in
'$Root' " -f "$Msgs_file" <<EOM >& /dev/null
An error occurred in the translation of
'$Pathname'.
The error message is as follows:
~r $Root.out
EOM
      mv -f /tmp/$Root.err $HOME
      goto cleanup
    else
      rm -f $HOME/$Root.err
    endif
  endif
#
# No runtime translation necessary.
#
else
  set Exec_file=$Pathname
endif
#
# Execute the report.
# Output to a file in the user's home directory
#
if ("Output" == "file") then
  set Fileout=$Root.out # Name of ACE output file
  set Pathout=$HOME/$Fileout # Pathname for output file
  UTL_PATH/acego -q $Exec_file $Exec_args:q >& $Pathout
  set Exitstat=$status
  if ($Exitstat != 0) then
    /bin/csh -f SCP_PATH/util/msg_queue.scp \line -s "ACE: Error in '$Root' " -
f "$Msgs_file" <<EOM >& /dev/null
An error occurred in the execution of
'$Pathname'.
The error message is saved in '$Root.err' in your home directory.
EOM
    mv $Pathout $HOME/$Root.err
  else
    /bin/csh -f SCP_PATH/util/msg_queue.scp \line -s "ACE output for
'$Root' " -f "$Msgs_file" <<EOM
The output for your '$Root' ACE report has been saved in the file
'$Fileout' in your home directory. It will be over-written
the next time you run the '$Root' ACE report.
EOM
    rm -f $HOME/$Root.err
  endif
#
# Pipe output to a printer
#
else
  UTL_PATH/acego -q $Exec_file $Exec_args:q |& $Output $Formtype
  set Exitstat=$status
  if ($Exitstat != 0) then
    /bin/csh -f SCP_PATH/util/msg_queue.scp \line -s "ACE: Error in '$Root' " -
f "$Msgs_file" <<EOM >& /dev/null
An error occurred in the execution of
'$Pathname'.
The error message was sent to '$Output'.
EOM
  endif
endif
#
# Cleanup
#
cleanup:
set nonomatch
rm -f /tmp/{$Root,$Root.*}
exit $Exitstat

```

Runreport Processing

The edited source and compiled version for execution are stored in '/tmp' only for the duration of the script's execution. Users should be cautioned not to run a report if the same report is being run by another user during the current run (i.e., you cannot run two versions of the same report concurrently). The compiled version of the second run will overwrite the initial compiled version in '/tmp'.

Script Menu Option File

From the menu processor, you run the *runreports* script using a menu option. The menu option file uses the macro, RUN_REPORTS, to pass your values and the installed command file to the *runreports* script. Following is an example of the *runreports* script menu file. Note the lines where SELECTFIELD, SELECTVALUE, and SORTFIELD are passed to macros and then to the script.

```
SD=ACT/SAT Scores
LD=ACT/SAT Scores of Recruits To College
RD=
                                ACT/SAT Scores of Recruits To College
                                -----
DC=DC_PRINT(DC_PATH/selectsort.doc)
PR=RUN_REPORTS
PP=
PA=-f
PP=
PA=FT_STANDARD
PP=
PA=`OTH_PATH'/admit/recrpt.oth
PP=Enter Planned Enrollment Session (e.g., FA, SP)
PA=upshift,default="FA",length 4
PP=Enter Planned Enrollment Year (e.g., 1984, 1985)
PA=include=(1983:2000),default="1985",type integer,length 4
PP=Enter Enrollment Status
PA=upshift,default="APPLIED",length 8
PP=Enter Exam Type (ACT or SAT)
PA=include=(ACT,SAT),upshift,default="ACT",length 4
PP=
PA=PA_DEFINE(SELECTFIELD)
PP=Enter Selection Field (e.g., country, state, city)
PA=dwshift,default="country"
PP=
PA=PA_DEFINE(SELECTVALUE)
PP=Enter Value For selection Field (e.g., USA, OH, Fairfield)
PA=default="USA"
PP=
PA=PA_DEFINE(SORTFIELD)
PP=Enter Sort Field (e.g., state, city)
PA=dwshift
PP=PP_OUTPUT
PA=PA_OUTPUT
SP
```

Using Print Spooler Software

Introduction

CX print spooling software provides output device control and spool queue management. This allows you to print reports or other jobs without tying up your terminal. In addition, you can manage jobs and queues with formtypes and owners.

The Spooling Process

The spooling process can be broken down into two phases, including:

- The queuing of the print job
- The printing of the job

CX has three programs involved in the phases of the spooling process:

1. The `lpr` program, which does the following:
 - Queues the print job into a spool queue directory
 - Calls the `lpd` program
2. The `lpd` program, which does the following when it locates a print job:
 - Opens the job file and the output device
 - Calls the print filter (`lpf`)
 - After the job is finished printing, `lpd` removes the job file from the queue
3. The `lpf` program, which does the following:
 - Copies the queued job file to the output devices

Creating a Spool Queue

The task of establishing a printer as a spooled device basically involves two steps. The first is to create the spool queue using the `MKSPOOLER` command and the second is to test the initialization of the printer.

1. Determine the following:
 - The `tty` line (or `lp`) to be used to connect the printer
 - The type of spooler from `$CARSPATH/system/etc/prtypes.s`.

Note: If you are connecting a network printer at a specific IP address, provide the IP address to `mkspooler` instead of the `tty` device name

The type entry is copied into `$CARSPATH/install/sys/lib/prtab` for use by the spooling software.

2. Use the `mkspooler` command to create the spool queue. The command specification is as follows: **`mkspooler ttyname spoolername [spoolertype]`**
3. For example, do the following to add a NEC spinwriter on terminal line `tty07` as `nec2`:
 - At the shell prompt, enter: **`su`**
 - Enter the password.
 - Enter the following: **`mkspooler tty07 nec2 nec`**
4. After the system creates the spooler, you can test the spooler with the `lpinit` utility.

Note: A simple test is to send the output from the `lptest` command to the new spooler (e.g., `lptest | buslpr`).

Spooler File Locations

The following lists the directory locations for spooler-related files on CX.

Spooler Files

The following files are located in \$CARSPATH/install/utl

- lpr
- lpc
- lpq (linked to lpc)
- lprm

Spooler Daemon and Filter

The following files are located in \$CARSPATH/install/sys/util

- lpd (Spooler Daemon)
- lpf (Spooler Filter)

Printer configuration file

The printer configuration file is located in \$CARSPATH/install/sys/lib/prtab

Printer options file

The printer options file is located in \$CARSPATH/install/sys/lib/proptions

Spool directory for FPS intermediate files

The spool directory for FPS intermediate files is located in \$CARSPATH/spool/forms

Spool directory for queue pr

The spool directory for queue pr file is located in \$CARSPATH/spool/{printer name}

SECTION 12 – THE MENU SYSTEM

Overview

Introduction

This section describes the menu system of CX. In order for a CX menu user to access the various screens and reports that are available on CX, each screen and report must have a corresponding option on the CX menu. Also, if there are particular parameters that the system requires when calling up a screen or report, the system must prompt the user for the required information. This section explains how you can create or modify the options that appear on the CX menu to best suit your institution's needs.

The process of identifying parameters that the system requires in order to display a screen or report is comprised of several phases. All of these phases must be complete in order for a CX user to be able to access the screen or report from the CX menu. This section identifies the phases in the process for placing a screen or report on the CX menu, and guides you through the necessary procedures.

The Process for Placing a Screen or Report on the Jenzabar CX Menu

The following explains the process involved in placing a screen or report on the CX menu structure.

1. Decide whether or not a screen or report needs to have a value passed to it, and define what that value is.
2. Create the screen or report with the specified variables needed to accept the values you have previously defined.

Note: See the Informix documentation for further information on creating screens or reports.

3. Create or modify a menu option (menuopt) file.
4. Update the appropriate menu description (menudesc) file to place the new or updated menu option file on the CX menu structure.
5. Use the CX menu structure to run the screen or report using the updated menu and menu option you created or modified.

Menu Option (Menuopt) Files

Introduction

A menu option (menuopt) file is a file that defines an option on the CX menu structure. A menu option file contains valuable information relating to that specific menu option.

Example: Dean's List Menuopt File

Following is an example menu option (menuopt) file. This example contains all the components of the menuopt file that are necessary for the Dean's List window to function properly.

```
}
screen
{
    DEAN'S LIST

    PP_SESS[PA4]
    PP_ACAD_YR[PA5]
    PP_PROG[PA7]
    PP_STATUS[PA9]
    PP_GPA[PA10]
    PP_FT_HRS[PA11]
}
end
attributes
SD: optional,
    default = "Dean's List";
SP: optional,
    default = "schedtime,,N";
OUTPUT: optional,
    default = "${CARSPRINTER},";
RD1: optional,
    default = "All students with a session gpa greater than or equal to"
RD2: optional,
    default = "the specified gpa value and taking the full time"
RD3: optional,
    default = "hours of credit specified will appear on the list."
RD4: optional,
    default = "";
RD5: optional,
    default = "NOTE: Be sure the student statistics have been"
RD6: optional,
    default = "updated by using the following option,"
RD7: optional,
    default = "'m4_getoptdesc(regist/programs/trns.Cu)'. "
PR: optional,
    default = "RUN_REPORTS";
```

```

PA1: optional,
    default = "-f";

PA2: optional,
    default = "FT_STANDARD";

PA3: optional,
    default = "ARC_PATH/regist/deanlist";

LU4 = sess_table.txt, optional;

PA4: optional,
    comments = "COMMENT_SESS_TBCODE COMMENT_TBL",
    default = "SESS_DEF",
    length = 4,
    lookup LU4 joining *sess_table.sess,
    upshift;

PA5: optional,
    comments = "COMMENT_CALENDAR COMMENT_TBL",
    default = "ACAD_YR_DEF",
    ACAD_YR_INCL,
    length = 4,
    type integer;

PA6: optional,
    default = "ALL_SITES";

LU7 = prog_table.txt, optional;

PA7: optional,
    comments = "COMMENT_PROG_TBCODE OrCOMMENT_BLANK_ALL. COMMENT_TBL"
    default = "PROG_DEF",
    length = 4,
    lookup LU7 joining *prog_table.prog,
    upshift;

LU8 = subprog_table.txt, optional;

PA8: optional,
    comments = "COMMENT_TSUBPROG_TBCODE COMMENT_BLANK_ALL. COMMENT_TBL",
    default = "SUBPROG_DEF",
    length = 4,
    lookup LU8 joining subprog_table.subprog,
    upshift;

PA9: optional,
    comments = "Enter student academic status, blank for STUAC_STAT_REG. COMMENT_TBL"
    include = ( STUAC_STAT_REG, " " ),
    length = 1,
    upshift;

PA10: optional,
    comments = "Enter the lowest session GPA value for selection.",
    default = "3.5",
    length = 5,
    type double;

PA11: optional,
    comments = "COMMENT_FULL_TIME",
    default = 12.0,
    length = 5,
    type double;

end

```

Menu Option Prompt

Text that begins with "PP_" in a menu option file signifies a macro. A macro is a short way of representing longer strings of text. The *make* processor expands the macros at a later time so that end users see the expanded statements in standard English, rather than macros, on the screen. For more information on macros in *CX*, see *Setting Up Macros* in this guide.

Menu Option Attributes

Menu option (menuopt) attributes are abbreviated notations that define parts of the menu option file. The following lists valid menu option attributes, their descriptions, and examples. Some of the attributes are optional, and might not be used in each menu option file. The attributes sections of the menuopts should be in standard format. Standard format dictates that attributes appear in alphabetical order, one per line with blank lines between the tags, and the optional attribute should appear on the tag line. Run the *stdscr* script to automatically standardize the attribute section of the screen.

Note: The number symbol (#) after an attribute in the following list denotes a number.

ADR

The ADR attribute passes appropriate default values to the *CX* addressing process. When you specify the ADR attribute, the Addressing Parameters window appears for the end user.

Example: ADR: optional, default = "N,N,N,0";
(This specifies four parameters, separated by commas, to the ADR screen.)

comments

You must define the comments attribute for any PA# attribute that appears in the menu option file. You should equate the comments attribute to a COMMENT macro, if possible. If a table lookup, blank (for all), or wildcard value is valid, then you must specify the COMMENT_TBL, COMMENT_BLANK_ALL, or COMMENT_WILDCARD macro, respectively.

Note: The COMMENT_TBL macro should appear in any comment for which the PA# attribute contains a table lookup or an include attribute.

Example: comments = "COMMENT_ACAD_PROG,
COMMENT_BLANK_ALL. COMMENT_TBL",

default

You should specify a default value for all appropriate attributes. Equate the default to a macro whenever possible.

Example: default = "DEF_MAJ",

dwshift

When you specify the dwshift attribute, anything a user types for the field to which this attribute corresponds appears in lowercase letters.

Example: dwshift;

GET_ORDER (Get Order)

The GET_ORDER attribute specifies the order the cursor is to follow through the fields in the menu option file. If you do not specify the GET_ORDER attribute, the cursor flows in a logical order. The GET_ORDER attribute is optional and is seldom used.

Example: GET_ORDER: group=(PA3,PA4,PA5,PA7), autonext;

include

You must specify the include attribute for attributes in which you do not specify a lookup attribute. Use an include macro whenever possible. Integer ranges are valid values with include attributes. Table lookup capability is provided for include attributes.

Example: ACAD_YR_INCL,
include = (1:10000),
include = (MGRD,FGRD),

LABELS

The LABELS attribute passes appropriate default values to the CX letter and label screen. The default value of the LABELS attribute contains four values (separated by commas) that act as defaults for the Labels Parameter window. When you specify the LABELS attribute, the Labels Parameters window appears for the user.

Example: LABELS: optional; default = "N,lup35x5,R,N".

length

You must specify a length for all PA# attributes. If the top is related to a database column, the length must equal the column length defined in the schema. You can equate the length value to a macro for the financial module columns, where the defined length in the schema is a macro.

Example: length = GL_CNTR_LEN,
length = 40,

lookup

You should use table lookups when a valid table exists for a particular field. You can force the value entered to be in the table name with an asterisk (*), though this still allows a blank value to be entered as input if the required attribute is not present.

Example: lookup LU# joining *major_table.major,

LU# (Look Up)

Use the LU# attribute to specify a table lookup. The LU# attribute defines the table and column being looked up. The columns are displayed within the standard table lookup window. Each LU# attribute must have an optional attribute associated with it, as well as an associated PA# attribute that contains a lookup joining clause.

Example: LU5 = table.lookup_column, optional;
PA5: optional,
...
lookup LU5 joining *table.column;

optional

You must include the optional attribute in all attributes for the menu option file.

Example: optional,

Note: With both optional and required attributes, the required takes precedence. Optional indicates if the field must be in the screen definition section.

OUTPUT (Output)

The default value of the OUTPUT attribute contains two values (separated by a comma) that act as defaults for the Output Parameters window. The first value denotes the default output mode, which is either "file," "more," or a valid printer. The second value denotes the default file, which is only valid if the default output mode is "file."

Example: OUTPUT: optional,
default = "\${CARSPRINTER},"

OUTPUT: optional,
default - "file,outputfile",

PA# (Parameter attribute(s))

The default for the PA# attribute defines a possible value that is passed to the process being executed. Comments are displayed on line 24 of the terminal. The arguments are passed to the process in the order defined by the integer appended to the PA attribute. You should use macros for defaults, comments and includes whenever feasible. The length, comments and optional attributes are mandatory for each PA# attribute.

Example: PA7: optional,
upshift,
length 4,
lookup LU7 joining *table.column,
comments ="COMMENT_TABLE_COLUMN.
COMMENT_TBL",
default = "COLUMN_DEF";

PR# (Process Run)

The default value for the PR# attribute defines which process to execute. The default is typically either an application program or a script. The PR# attribute is mandatory, and you can define only one PR# attribute for each menuopt.

Example: PR: optional, default = "RUN_PROG_INFORMER";

PW (Password)

A password is required before the menu option screen is presented to the data entry operator.

Example: PW: optional,
default = "@REGIST";

RD# (Run Description)

The default values for the RD# attribute define the run description for the menu option. The default values appear in numerical order, defined by the integer appended to the RD attribute. The appended integer must be unique within the set of RD attributes. The RD attributes are optional. This text gets displayed in a window when the Help command is used in the menuopt.

Example: RD1:, optional,
default = "This option updates all the student records.";

To reference the text for another menuoption use the m4_getoptdesc function. This allows the short description to be dynamically updated as it will automatically retrieve the description from the referenced menuoption. The txt2rd and rd2txt assist in the development of the RD# tags, and can be run from inside the VI editor. They are described below:

- txt2rd - Transforms a block of text into the format required for a run description
- rd2txt - Transforms a block of RD# tags into a block of text.

required

The required attribute will force the user to enter a value other than a blank value in the field to which the required attribute corresponds.

Example: required,

SD (Short Description)

The default value for the SD attribute defines the short description for the menu option that appears on the CX menu. The SD attribute is mandatory, and you can define only one SD attribute for each menu option. Capitalize the first letter of all significant words in the short description to adhere to CX standards. The menuopt title will be the same as the short description. Use the m4_center macro command to center the title, and start it at column 20 to align with the menuopt prompts.

Example: SD: optional,
 default = "Move Graduates to Alumni";

Note: The SD default has a maximum of 26 characters.

SP (Schedule Process)

The default value of the SP attribute contains three values (separated by commas) that are defaults for the Output Parameters window. The first value denotes the default time, the second value denotes the default day, and the third value denotes whether or not to run the process in the background.

Note: You can use the keyword "schedtime" as the first value, if it is defined in the \$CARSPATH/system/etc/menuparam.s directory path. If "schedtime" is not defined in this file, the default value is "1100P."

Example: SP: optional,
 default = "schedtime,,Y";
SP: optional,
 default = "7:00P,sunday,Y";
SP: optional,
 default = "NOW,,N";

type

Use the type attribute only for numerical columns.

Example: type double;

upshift

Specify the upshift attribute to cause anything that a user types for the field to which this attribute corresponds to appear in uppercase letters.

Example: upshift,

WARN (warning)

The default text for the WARN attribute is displayed in a dialog box prior to the Output Parameters window appearing. The WARN default cannot exceed 74 characters. You can use the WARN attribute to notify the user that output requires wide paper by setting the default to the WARN_WIDE_OUTPUT macro.

Example: WARN: optional,
 default = "Execute the 'Update Stats' option first.";
WARN: optional,
 default = "WARN_WIDE_OUTPUT";

How to Create a Menuopt File

The following lists the steps to follow for creating a menu option file for a screen or report.

1. Enter **cd \$CARSPATH/menuopt/module/reports** to access the subdirectory containing ACE report menu option files for a particular module, or enter **cd \$CARSPATH/menuopt/module/screens** to access the menu option files for the screen subdirectory for a particular module.
2. Enter **make add F=filename** to create a skeleton menu option file that will reference ACE reports
3. Enter **vi filename** to place the contents of the file you selected on the screen.
4. Modify the menu option (menuopt) file according to your requirements.
5. Press **<Esc>**, then enter **:wq** to exit the file.
6. Enter **make F=filename** to let the *make* processor expand the macros and translate and create an object file.
7. Does the filename "*filename.opt*" exist?
 - If yes, the *make* processor was successful; go to step 8.
 - If no, the *make* processor was not successful. Look at the "*filename.err*" file to see what problems *make* encountered. Resolve the problems and then repeat step 6 until the *make* processor is successful.
8. Enter **menu -o .OBJ/filename.opt** to execute the menu option in a local test mode.

Note: The screen information should exist in a box that appears near the top of your terminal. The lay out of the screen determines the exact position in the source file.
9. Press **Ctrl-w** for the information from the run description (RD) lines to appear in a box near the bottom of the screen.
10. Press **Finish** to execute the menu option.

If the system asks you where you want the output placed and when you want the report to be processed, the PR and PA lines in the menu option file are correct. Go to step 12.

If you receive a message indicating that the CX menu processor is having a problem, most likely, the PR line or the PA line specifying the location of the installed screen or report is incorrect, or you have not temporarily installed the screen or report that the PR or PA line is referencing. Resolve the problem, then go to step 12.
11. Enter **make tinstall F=filename** to temporarily install the menu option file. Create a menu description enter to make the option available to data entry people.
12. Do you want to create another menu option file for a screen or report?
 - If yes, go to step 1.
 - If no, you have completed this procedure.
13. After you have completed testing and the menu option is performing according to the specifications, check the menu option back into RCS and it will record the changes.

Example: %make cii F=filename L="*<a comment about the change>*"

How to Modify a Menuopt File

The following lists the steps to follow for modifying a menu option that contains a screen or report.

1. Enter **cd \$CARSPATH/menuopt/module/reports** to access the subdirectory containing ACE report menu option for a particular module, or enter **cd \$CARSPATH/menuopt/module/screens** to access the menu option files for a screen subdirectory for a particular module.
2. Enter **cp filename newfilename** to copy the menu option file to your working directory.
3. Enter **vi newfilename** to place the contents of the file you selected on the screen.
4. Modify the menu option (menuopt) file according to your requirements.
5. Press **<Esc>**, then enter **:wq** to exit the file.
6. Enter **make add F=newfilename** to add the *make* header information to the file and put the file under Revision Control System (RCS) control.
7. Enter **make F=newfilename** to let the *make* processor expand the macros and translate and create an object file.
8. Does the filename "*newfilename.opt*" exist?
 - If **yes**, the *make* processor was successful; go to step 9.
 - If **no**, the *make* processor was not successful. Look at the "*filename.err*" file to see what problems *make* encountered, resolve the problems, then repeat step 7 until the *make* processor is successful.
9. Enter **menu -o .OBJ/filename.opt** to execute the menu option in a local test mode.

Note: The screen information should exist in a box near the top of your terminal. The exact position is determined by the layout of the screen in the source file.
10. Press **Ctrl-w** for the information from the run description (RD) lines to appear in a box at the bottom of the screen.
11. Press **Finish** to execute the menu option.

If the system asks you where you want to place the output and when you want the report to be processed, the PR and PA lines in the menu option file are correct. Go to step 12.

If you receive a message indicating that the CX menu processor is having a problem, most likely, the PR line or the PA line specifying the location of the installed screen or report is incorrect, or you have not temporarily installed the screen or report that the PR or PA line is referencing. Resolve the problem, then go to step 12.
12. Enter **make tinstall F=filename** to prepare the file for inclusion on the CX menu.
13. After you have completed testing and the menu option is performing according to the specifications, check the menu option back into RCS and the changes will be recorded.

Example: %make cii F=filename L="<a comment about the change>"

Menu Description (Menudesc) Files

Introduction

A menu description (menudesc) file is a file that defines which options are to appear on the CIS menus and submenus. A menu description file provides valuable information relating to a specific menu or submenu.

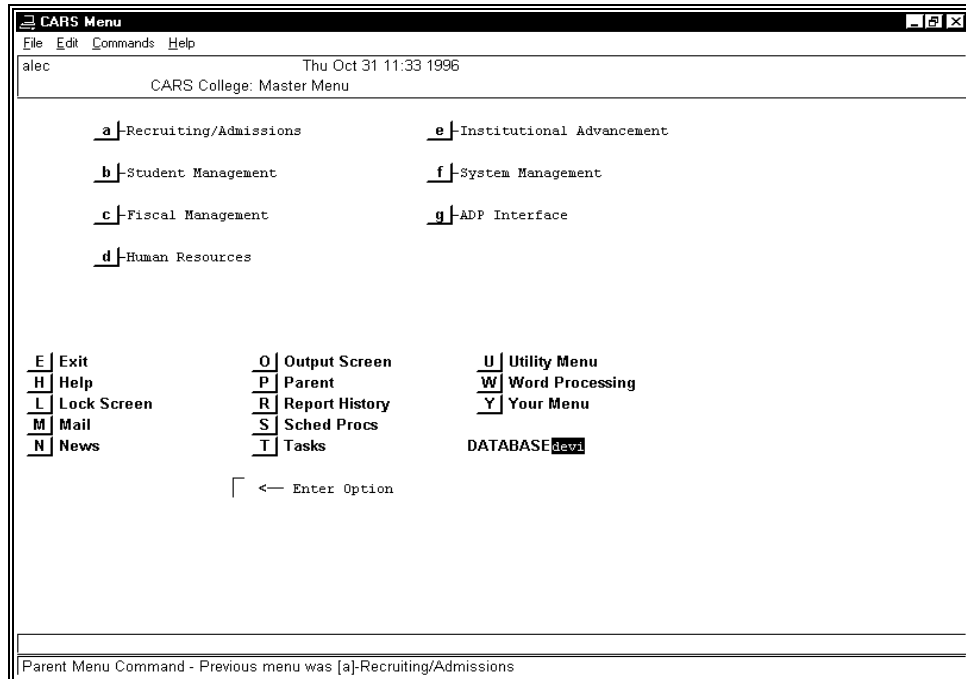
Example: Jenzabar CX master menudesc file

Following is an example of a menu description (menudesc) file for the Jenzabar CX Master Menu.

```
TI=INST_NAME: Master Menu
AO=MNU_AO_TIME
SD=CARS Solution Master Menu
LD=Master Menu For INST_NAME
MNU_SUB(student)
MNU_SUB(fiscal)
MNU_SUB(instdev)
MNU_SUB(system)
```

Example: Jenzabar CX Master Menu

Following is an example of how the Jenzabar CX Master Menu appears when you use the CX menu structure. The attributes in the menu description file corresponding to the CX Master Menu define what options appear on this menu.



Menu Description Attributes

Menu description (menudesc) attributes are abbreviated notations that define parts of the menu description file. The following lists valid menu description attributes, their descriptions, and examples. Some of the attributes are optional, and might not be used in each menu description file.

AC (Access)

Use the AC attribute to specify whether or not a login name or group is permitted to access this option. This references the keyword in the file \$CARSPATH/system/etc/menuac.s which defines user access.

Example: AC=financial (This means only those in the financial group are allowed to execute this menu option.)

Example: AC=@300 (This means that you are using the value of line 300 in the associated list in the file.)

You can also permit access to users in multiple groups.

Example: AC: optional,

default = "financial carstctrl" (You can separate the items in the list with a comma or with whitespace.)

You can exclude particular users from a group by using an exclamation point (!).

Example: AC=common!suzie (This includes everyone in the group "common" except suzie.)

AO (Auto Logout)

After the specified time, in minutes, the menu will automatically log out.

Example: AO=MNU_AO_TIME_10

LD (Long Description)

Up to 78 character description of what this menu item does.

Example: LD=Maintain course and schedule information

MNU_OPT

Points to the location of a menuopt file relative to \$CARSPATH/menuopt.

Example: MNU_OPT(regist/reports/deanlist)

MNU_SUB

Points to the location of a submenu relative to \$CARSPATH/menusrc.

Example: MNU_SUB(student/regist/regist)

PW

You must have a password to enter this menu description. (This references the keyword in the file \$CARSPATH/system/etc/menupw.s which defines the password.)

Example: PW=@REGIST

SD (Short Description)

A short description (up to 25 characters) of what this menu item does.

Example: SD=Fiscal Management

TI

The title appearing on the screen when this menu description is selected.

Example: TI=Student Management: Main Menu

How to Create a Menudesc File

The following lists the steps to follow for creating a menu description (menudesc) file that contains the screen and report menu option files that you created previously.

1. Create a subdirectory into which you will be creating the menu description file.
2. Enter **cd menusr/module/subdirectory** to access the subdirectory in which you are creating the menu description (menudesc) file.
3. Enter **makeinit mnu** to tell the Revision Control System (RCS) the type of files that are to reside in this subdirectory.
4. Enter **make add** to create the skeleton menu description file.

Note: You do not need to specify a filename with this command; the *make* processor will automatically create only menu description files in this directory.
5. Enter **vi menudesc** to add the screen and report information to the skeleton file
6. Enter **make tinstall** to temporarily install the file.

Note: You do not need to specify a filename with this command; the *make* processor will create only menu description files in this directory.
7. Modify the menudesc file in the parent directory and add a MNU_SUB(.....) line to pull in the subdirectory created in the previous step.
8. Did you add a new menu option or did you change the short description (SD) line in your menu description file?
 - If yes, cd to the previous directory and enter **make reinstall** to install the master menu description to include your latest changes and options.
 - If no, go to step 9.
9. Enter **menu** at the prompt to use the CX menu structure.
10. Select the appropriate menu options to locate the menu options you added in the menu structure.
11. Are your menu options on the CX menu structure?
 - If yes, you added the menu option and menu description files correctly.
 - If no, or an error message appears, check the information in the MNU_SUB or MNU_OPT attributes to ensure the locations are correct.

Menu Parameter (Menuparam) File

Introduction

You can modify the menu processor operation by updating the menu parameter file ('\$CARSPATH/system/etc/menuparam.s') which is installed in \$SYSPATH/etc/menuparam. You modify the variables in this file to affect the way the menu system interacts with users.

Example

The following is an example of the menu parameter (menuparam) file.

```
interval      15
background    5
scheduled     9
byeok
schedtime    1100P
nodocprint
notimedisplay
```

Menu Parameter Options

You set the menu parameter file options to control menu system interaction. The following lists the options, their descriptions.

interval

Controls the number of seconds between screen time updating and checking. You can enter a value between 10 and 300.

background

Specifies the maximum number of background jobs. You can enter a value between 1 and 9.

scheduled

Specifies the maximum number of scheduled tasks. You can enter a value between 0 and 9.

Note: The maximum values in Background and Scheduled apply to each user, individually, system wide.

byeok

Specifies whether the system will Bye out with background jobs running. The default is nobye which you specify by leaving the line blank.

schedtime

Specifies when to run the process. You can enter a time (using A or P to denote a.m. or p.m.; or military time) or NOW to run the process immediately. The default is 1100P which you specify by leaving the line blank.

nodocprint

Specifies whether to allow printing of documentation using the D option. The default is docprint which you specify by leaving the line blank.

notimedisplay

Specifies whether to display the time on the top line of the menu. The default is `timedisplay` which you specify by leaving the line blank.

News and Mail Menu Features

Introduction

The CX system's upper case options include the Mail and News options. The Mail option accesses the default UNIX mail program, *mailx*. When new mail exists for a user, a highlighted message, "You have new mail" appears on the menu. You can, if desired, change the mail program that the option accesses.

Similarly, the News option is designed to access a News program. You can set up this option to provide news information (e.g., system maintenance schedules) to menu users.

Different Mail Program

To set up the Mail menu option to access a different mail program (e.g., elm or pine), you create a script in the \$CARSPATH/modules/util/commands directory called *mail*. You should design the *mail* script to execute your institution's desired mail program. You then install the script in \$CARSPATH/install/util. Do the following:

1. Add the *mail* file in \$CARSPATH/modules/util/commands:
% **make add F=mail**
2. Write a script that executes the desired mail program.
3. Test install the script.
% **make tinstall F=mail**
4. After testing the script and its installation, check in and install the script.
% **make ci F=mail**
% **make install F=mail**

News Program

To set up the News option to access a news program (e.g., the UNIX news utility), you create a script in the \$CARSPATH/modules/util/commands called *news*. You should design the *news* script to execute the news program. You then install the script in \$CARSPATH/install/util. Do the following:

1. Add the *news* file in \$CARSPATH/modules/util/commands:
% **make add F=news**
2. Write a script that executes the news program.
3. Test install the script.
% **make tinstall F=news**
4. After testing the script and its installation, check in and install the script.
% **make ci F=news**
% **make install F=news**

The news program must be executable; otherwise, the menu displays a message indicating that news is not available at this time.

Note: An alternative to this menu feature is to provide news information in users' login processes. In this scenario, news information automatically appears whenever the user logs in. This also does not require the user to select a menu option in order to view the news.

Nomenu Feature: Controlling Menu Access

Introduction

The system administrator can control the access of menu users through use of the *nomenu* feature. Using this feature, you have the ability to stop menu access of all or specific databases. When a user logs in, the CX checks for the existence of *nomenu_\$CARSDb*, which applies to a specific database, then *nomenu*, which applies to all databases. If either file exists, the menu immediately logs the user out.

Nomenu Files

The *nomenu* and *nomenu_\$CARSDb* files are text files into which you can type a message to be displayed to the users when they access the menu. You are not required to enter text because the menu also displays a message, “No menu access permitted at this time” when it finds either of the files.

- You use the *nomenu* file when you want to restrict menu access of all databases.
- You use the *nomenu_\$CARSDb* file when you want to restrict menu access to a specific database. The file is based on the value of the \$CARSDb variable, not the \$CARSV or \$CARSPATH variables.

You place either file in the \$CARSPATH/events directory.

Menu Process

The CX uses the *nomenu* feature in the following way.

1. When a user logs in, the menu first looks for the *nomenu_\$CARSDb* file in \$CARSPATH/events. (The \$CARSDb variable specifies the database currently accessed by the user.)
2. If the menu does not find the database-specific file, the menu looks for the *nomenu* file in the \$CARSPATH/events directory.
3. If the menu locates either file, the menu does the following:
 - Displays “No menu access permitted at this time.”
 - Displays the contents of the *nomenu* file.
 - Logs the user out.

SECTION 13 - CX SYSTEM STANDARDS

Overview

Introduction

Jenzabar has developed standards and conventions to assist developers and clients in keeping the components of Jenzabar CX uniform and standard. This section provides you with standards for the following components of the system:

- Data dictionary
- Data structure
- Program name abbreviations
- Schemas
- User interface for:
- Menu source
- Menu options
- Program screens
- PERFORM screens
- Comment macros
- PERFORM screens
- Entry Library screens
- ACE reports
- Menu options
- Programming style
- Software maintenance
- Program documentation

Use of the Standards

While the original intent of the system standards was to assist Jenzabar developers in programming functions, client institutions can make use of the standards when performing local modifications to their systems.

CX Data Dictionary Standards

Introduction

The database dictionary is the core of control for the database. The system uses the INFORMIX Database Management System (DBMS) to add additional capability by tracking more information relating to data elements and relations. The purpose of the data dictionary is to:

- Define schema files and fieldnames.
- Define schema field characteristics.
- Control representation of data.
- Indicate programs, reports, screens using file, field, index.

Definition

The general definition of a data dictionary is that it is a catalog of all data types to provide their names and structures, and information about data usage. Advanced data dictionaries have a directory function that enables them to represent and report on the cross-references between components of data and business models. A data dictionary provides for:

- Agreement on data element definitions
- A common source of data for programs, jobs, modules, files, fields trusted by all users
- Enforcement of data standards
- The avoidance redundancy of data across the system
- The detection of currently existing data redundancies
- Better control of changes in data
- Assistance in communication about data
- Assistance to security and control
- Assistance to end users

CX-Specific Definition

The CX data dictionary is a means of cataloguing the data elements in the database for programmers, account managers, and Jenzabar coordinators. The data dictionary provides for:

- Enforcement of data standards
- Enforcement of schema standards
- Enforcement of permissions
- Assistance in training clients

In particular, the data dictionary is used to:

- Coordinate definitions of fields and files
- Reduce redundancy of data between files
- Reduce redundancy of data within files
- Control data abbreviations
- Control security of fields
- Control security of files
- Control use of data field, file
- Control field, file modifications, additions, deletions
- Enforce use of schema abbreviations during make process.

INFORMIX Data Files

The data dictionary defines elements of INFORMIX data files.

Field Names

The data dictionary defines the following for field names in the Database Field record (dbfield_rec):

- Schema file name

- Database file name
- Field name
- Type (char, int, etc)
- Length (if type character)
- Text descriptions (3 lines of 40 char)
- Status (Current, Renamed, Deleted, Future)
- Status Date
- Type of index (blank, Primary, Duplicates, No Duplicates)
- Special field permissions
- ACE using field index
- PERFORM using composite fields
- Application software searching on index
- Includes of data values -currently in dbfield_text lines
- Key search words
- How data created (operator entered, schema makes)
- How is data updated (operator update, schema makes)
- How is data deleted (operator update, schema makes)
- When can data be deleted

INFORMIX File Names

The data dictionary defines the following for files in the Database File record (dbfile_rec):

- Schema file name
- Database file name
- Track Code (C,A,S,D,F)
- Name of primary index
- Text descriptions (2 lines)
- Permissions
- Application software use of file
- Key search words

Relationships Between Files

The data dictionary defines the following for file relationships in the Database Relationships record (dbrel_rec):

- Name of the two database fields
- Type of relationship between the first field to the second field (Join or Master)

Composite Fields

The data dictionary defines the following for composite fields in the Database Composite record (dbcomp_rec):

- Name of composite field
- Names of up to 8 database fields comprising the composite field

ACE Reports

The data dictionary defines the following for ACE reports:

- Listing of files (alpha) with definition lines
- Fields (alpha) by file (alpha) by track code
- Fields (alpha)
- Files by application program
- Indexes (alpha) with application program
- Indexes (alpha) with ACE report
- Indexes (alpha) with PERFORM screen

PERFORM Data Entry Screens

The data dictionary defines the following for PERFORM screens:

- Query on each field in each INFORMIX file
- Update selected fields (those modified through DBSTATUS)

Application Software

The data dictionary defines the following for application software:

- Key word search processor, with output printed or on the screen
- Test against schema abbreviations

Dictionary Data Schemas

The data sources for the data dictionary are specified in schema files located in \$CARSPATH/schema. The schemas are:

- dbfield (dbfield_rec) for database elements
- dbfile (dbfile_rec) for database files
- dbrel (dbrel_rec) for database relationships
- dbcomp (dbcomp_rec) for database composites

To set up the data dictionary schemas, use the *make* processor to:

- Create data for dbfield_rec and dbfile_rec (builds)
- Update data for dbfield_rec and dbfile_rec (rebuilds and renames)

Schema Definitions

The schema files define the elements of the data dictionary

Database Files (dbfile)

The schema specifies the following:

- Link to UNIX file name
- Data file name
- Composite keys
- Programs using file
- Fieldnames
- Permissions

Database Fields (dbfield)

The schema specifies the following:

- INFORMIX file name
- UNIX file name
- Fieldname
- Field type
- Field length
- Three lines of text describing fieldname
- Status of fieldname (current, deleted, renamed)
- Index type (Primary, duplicates, no duplicates, blank)

Database Relationships (dbrel)

The schema specifies the following:

- Primary field name
- Secondary field name
- Type of relationship of the first field name to the second field - Join or Master

Database Composite (dbcomp)

The schema specifies the following:

- Composite field name

- Names of all database elements comprising the composite field

Entering Data

- Currently, PERFORM data entry screen
- Eventually schema make processor

Retrieving Data

- PERFORM data entry screen
- ACE reports
- Eventually Key Word Search Processor

Data Structure Standards

Introduction

Utilizing standard formats for data structure and contents can provide consistency which will result in more efficient access and update to data. Standards for data encompass issues from table contents to name field consistency. In addition, you should adopt conventions and standards in the area of data security & integrity as well as data dictionary information.

Data Tables

The following are the standards for data tables.

1. All table code values will be in upper case letters
2. All code values will have a corresponding text field to describe the code.
3. Codes will follow industry standards where applicable. For example, US State codes will conform to the US Postal Service abbreviations and country codes will follow the internationally accepted values.
4. Most of the data for tables based on industry or CX standards will be provided with the system installation.
5. You can add codes from a client's previous system that do not meet the standards. You can add them as an additional field in the schema for cross reference.
6. Tables will not contain blank records unless a blank value is an acceptable value in the table
 - It is more acceptable to have "dummy" records than blank values. For example, state code of '--' for foreign countries, or an assessment code of 'NONE' to be used with crs and sec recs tuit_code where there is a lookup to the assessment table.
 - If a blank code exists in the table, there must be a description for it (e.g., Major Table ' ' is 'Undeclared').

Data Records

The following are the standards for data records.

1. Data records will use codes from tables whenever possible rather than free-format comment fields.
2. All logical fields will be upper case (e.g., Y for yes, N for no).
3. Predefined values will be treated like codes.
 - They will be upper case values.
 - Used mostly as status and type codes where the values are limited.

Comments

The following are the standards for comments.

1. Free format entry
2. Used to provide specific data about the record
3. Used as a notation, reference

Names

The following are the standards for names.

1. All names will be of the format 'Last, First Middle,Suffix'. The name must be in this format to use the ACE functions; `_first_name()`, `_last_name()`, and `_full_name()`.

2. Whenever possible, a record should only contain an id_no that joins with the id_no of the id_rec for displaying of the name. Use of name in multiple records takes away from the concept of a true "relational" database.

Social Security Numbers

1. Will be in the format: 123-12-1234
2. PERFORM screens will use a picture clause of "###-##-####" and a default of " - - "

Phone Numbers

1. Will be in the format: 111-222-3333 unless from a foreign country .
2. Only use picture clause in PERFORM screens where foreign phone numbers will not be entered.

Data Integrity/Security

The following are the standards for data integrity.

1. Converted data will be verified to conform with CX data standards
2. Converted data verification of accuracy is the responsibility of the client.
3. Data will be backed up on a daily basis
4. Data integrity is the responsibility of the client
5. Data security is the responsibility of the client

Data Dictionary Files

The following are the standards for data dictionary files.

1. Whenever schema files are created or fields are added, the file will be entered into the dbfile_rec and each of the data elements will be entered into the dbfield_rec.
2. Whenever a schema field is renamed in the database, the status for the field and the status date for the field will be updated in the dbfield_rec.
3. Whenever a schema field is deleted from the database, the status and the status date for the field will be updated in the dbele_rec. Do *not* delete the field name from dbele_rec.
4. Fields and files that are unique to a client will be entered and maintained by the client in the dbcfield_rec and dbcfile_rec.
5. There are additional files, that are also associated with the data dictionary, but they are currently not being maintained.
6. Eventually, the schema make process will update each of these files.

Standard Data Abbreviations

The following are the standard date abbreviations.

<u>Abbreviation</u>	<u>In Place Of</u>
AACRAO	American Association of College Registration and Admissions Officers
AC	accounting voucher
ACT	American College Testing Program
AD	add/drop slip
ADC	Aid to Dependent Children (finaid)
AFDC	Aid to Families with Dependent Children (finaid)
AX	american express
CA	cash
CBF	campus based funds
CC	credit card
CEEB	College Entrance Examination Board (now College Board)
CK	check
CR	cash receipt document
CSS	College Scholarship Service
CT	cash transfer voucher
CWSP	College Work Study Program
F/A	Financial Aid
G/L	General Ledger
GSL	Guaranteed Student Loan
HHS	Health and Human Services Department
IRS	Internal Revenue Service
JC	job costing voucher
MC	master charge
MO	money order
NACUBO	National Assoc. of College and Univ Business Officers
NASFAA	Nat. Assoc of Student Financial Aid Administrators
ND	endowment voucher
NDSL	National Defense/Direct Student Loan
NP	notes payable voucher
NR	notes receivable voucher
NSL	Nursing Student Loan
PC	purchasing voucher
PR	payroll voucher
PS	personnel voucher
RG	registration form
SAR	Student Aid Report (Pell Grant Program)
S/A	student accounts
SB	student billing voucher
SEOG	Supplemental Educational Opportunity Grant
SSI	Supplemental Security Income
VA	Veteran's Administration
VS	visa

Program Name Abbreviations

Introduction

Because some environments limit you to a 14 character filename, Jenzabar established program name abbreviations for the naming of menuopt files.

The following are guidelines to follow when creating program abbreviations:

- Use the standard abbreviations whenever possible on the earliest syllable possible
- If you need to remove more words, start removing vowels

Files In BINPATH

The following are program abbreviations for files in BINPATH.

acquery	acqu
admentry	adme
admstats	adms
adr	adr
audctc	audc
audent	aude
bgtalloc	bgta
bgtbasis	bgtb
bgtinstall	bgti
bgvoucher	bgv
billing	bill
cgrep	cgrp
ckabort	ckab
ckpost	ckps
ckrecon	ckre
ckslct	cksl
cseentry	cse
csstp	csst
daaudit	daau
dacvtmemo	dacv
daentry	dae
ddtp	ddtp
degaud	dgau
delentry	dele
dirdep	dird
docvoid	docv
employ	emp
employ2	emp2
f1099bld	99bd
f1099form	99fm
f1099tape	99ta
f1099tp	99tp
faaudit	faau
faentry	fae
faneed86	fa86
faneed87	fa87
filepost	fpst
finance	fin
fixpost	fixp
forward	fwd
giftpst	gftp
giftrcpt	gfr
glaudit	glau
glbalfwd	glbf
glclsg	glcl
grading	grdg
grdrpt	grdr
grvoid	grvd
idaudit	idau
interest	int
lead	lead
pay	pay
phbill	phbl
phchg	phch
phld_rolm	phld
praudit	prau

prckadj	prck
prdinit	prdi
prodentry	prde
progaudit	pgau
prstart	prst
purch	purc
purchaudit	puau
purchinv	puin
rcventry	rcve
readvt	rdvt
regaudit	rgau
regist	reg
reglist	regl
regno show	regn
requery	reqe
saaudit	saau
sabalfwd	sabf
sae	sae
sortpage	srtp
stmt	stmt
stuentry	stue
subbstat	sbst
tickler	tick
timeent	tent
tprog	tprg
trans	trns
trnsadd	trna
trnsent	trne
vchrecover	vchr
voucher	vch
w2build	w2bd
w2form	w2fm
w2tape	w2ta
w2tp	w2tp

Files In UTLPATH

The following are program abbreviations for files in UTLPATH.

acego	aceg				
aceprep	acep				
addlogin	adlg				
aim	aim	*	ansitar	ansi	
arappend	arap				
auditmail	audm				
bcheck	bchk	*	border	bord	* brand brnd
buildinstr	bldi				
carslock	crsl				
catb	catb				
chbyte	chby				
checkindex	chki				
chgrp	chgp				
chgmod	chmd				
chgown	chow				
cinstall	cins				
cleanlock	clnl				
coladd	cola	*	connect.sav	conn	
copyw	cpw	*	correct	corr	
cpcmp	cpcm				
cpdir	cpdr				
cpio	cpio	*	cptypes	cpty	* crashrecovery adme
curdate	curd				
cut	cut				
cutsheet	cuts				
dbbuild	dbbd				
dbreport	dbrp				
dbstatus	dbst				
delay	dly	*	enter1	e1	
enter2	e2				
fastup	fup				
fixlen	fixl				
formbuild	frmb				
fps	fps				
ftx	ftx				
fullpage	flpg				
grepwhatis	grwh				
hn	hn	*	homeinit	hmin	
horizon	hrzn				
informer	inf				
labels	lbls				
latitude	lat	*	listlock	lstl	
lpff	lpff	*	lpr	lpr	* lps lps
lpt	lpt	*	makeall	mkal	
makedef	mkdf				
makeinit	mkin				
menu	menu				
menucsh	mnuc				
menukill	mnuk				
movew	mvw				
newlogin	nwlg	*	noshell	nosh	* paste past
pdefs	pdfs				
perform	perf				
pmsort	psrt				
postmrg	pstm	*	print	prt	
printmenu	prtm				

remove	rm					
removew	rmw	*	reordpm		repm	
repgen	rpgn	*	rlpr	rlpr		
runat	rnat					
runcpp	rncp					
scrbuild	scrib					
setdb	stdb					
setmod	stmd					
setown	stow					
setuid	stui					
sigchange	sigc					
skip	skip	*	slave	slv	*	slv.wyse75
snapconfig	sncn					slvw
snapsys	snsy					
splitpage	sppg					
splits	spts					
splittee	spte					
sush	sush					
symlock	syml					
symunlock	symu					
timecmp	tmcm					
tstlock	tstl					
tx	tx					
ulistlock	ulst					
vt	vt					
wpvi	wpvi					

Schema Standards

Introduction

Standards for schemas provide for consistency in the naming of fields and files which makes the data more available to the end-user. The standards for schemas include file and field naming conventions, file locations, the contents of the schema file, etc.

Naming of Files and Fields

Jenzabar suggests that the INFORMIX relation name and the UNIX filename match whenever possible. The schema UNIX filenames should be a maximum of 10 characters and those files which contain schemas for "tables" should begin with 't'.

All schema field names must follow the abbreviations within this document.

Note: We are aware that this is not currently the case; however, any new additions to the database will adhere to these standards.

A suitable prefix will be assigned for each relation and will be pre-appended to all field names within that relation. Prefixes for "tables" will begin with 't'. Underscores will be used to separate the prefix from the rest of the field name. Portions may also be separated with underscores for readability (e.g., tmajor_text, ctc_no).

Location

Note the following file locations:

- The UNIX schema file will be located in the following path:
\$CARSPATH/schema/<track>/{schemafilename}
- The UNIX data file will be located in the following path:
\$CARSPATH/data/<track>/{datafile.dat,datafile.idx}

In the schema file, the location will be identified relative to the schema file (e.g., location "../data/student/tsess").

Note: The UNIX data file name will be the same as the UNIX schema file name.

Table or Record

A "table" is defined as a data file whose elements are rarely changed, containing a code (multi-character uppercase field) by which the records are accessed and a text field (typically 24 or more characters) to describe the code.

A "record" is defined as data file whose elements are constantly updated. It may also include code values from tables.

Testing for Correct Naming Conventions

The appropriate Track Coordinator will review all schema files before they are built within their track.

The Schema File

The display below is a copy of the skeleton of the schema. Following the display is a description of each section in the schema file, including a description of the elements that make up that section.

```

[DATABASE database-name]

TABLE table-name
DESC "description string"
LOCATION "dbspace name"
LOCKMODE { ROW | PAGE }
PREFIX "prefix used for makedef"
ROWLIMITS { <integer> | ?? } : { <integer> | ?? }
STATUS "status"
TEXT "text description string"
TRACK "Track code"

COLUMN column-name [TYPE] <column-type>
[DEFAULT <default-value>] [NOT NULL]
[<constraint-def> CONSTRAINT constraint-name]
COMMENTS "comment string"
DESC "description string"
HEADING "heading string"
TEXT "text string"

...<More column definitions, if any>

CONSTRAINTS
<constraint-def> CONSTRAINT constraint-name

...<More constraint definitions, if any>

INDEX
[ UNIQUE ] index-name ON (column-name [DESC] [,
column-name [DESC] [ ... ]])

...<More index definitions, if any>

GRANT
<access-type> TO ( user-name [ , user-name [ ... ] ] )

...<More access definitions, if any>

TRIGGER
[AUDIT ([<audit-column-list>]) [IN "<audit-server-name>"]
[CAPTURE FOR (<action-type-list>)]
[FOR CHANGE OF (<trigger-column-list>)]
GRANT SELECT TO (<user-list> )
[ON INSERT {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
[ON UPDATE {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]
[ON DELETE {BEFORE|FOR EACH ROW|AFTER} <triggered-action>]

```

Header Section

This section contains the standard schema header managed by RCS, including both header and log information.

Database Section

The first line in a schema file, after revision information, defines the database where the schema will be used. The entry of the database name must be preceded by the key word *database* (e.g., **database jenza**).

Table Section

In the TABLE section, you specify the INFORMIX database table that the schema defines. For example, *st_table* (State table) is specified in the *tst* schema file. Even though the INFORMIX software is not case sensitive, you should specify the table name in all lower-case letters to ensure that the system works correctly.

1. The text description of the file name. This entry is limited to a maximum of 24 characters and should be a user-friendly English text version of the file name (eg. 'Profile Information' might be the English version text for the file name 'profile_rec').
2. The location of the file. The pathname to the file is to be entered in quotes (e.g., `"../data/common/profile"`).
3. The LOCKMODE attribute, which specifies the mode that the engine will use to lock a record

in the table.

Note: Generally, you should use ROW level locking for a table. However, in cases where a table is rarely modified, but many rows are changed at a time, you can use PAGE level locking as a way to control the number of locks used in the engine during the changes to the data in the table.

4. The PREFIX attribute, which specifies the table prefix used by the MAKEDEF utility to create the structures used by C code.
5. The ROWLIMITS attribute, which specifies the initial size of the table, and the expected growth of the table. DBMAKE expects two integer values separated by a colon (:), or ?? to indicate using the default value for that integer. ROWLIMITS ??:?? indicates the default value for both initial size and expected growth.
6. As rows are added to a table, the database allocates disk space to the table in units called *extents*. Each extent is a block of physically contiguous pages in the dbspace. There are two types of extents: *initial* and *next*. The database allocates *initial* extent when the table is first created in the database. The database allocates a *next* extent whenever the current space for the table has been used up. The integers that you specify are used for calculating the size of extents in the database.
 - The first integer indicates the *initial* number of rows that will be in the table when it is created and loaded. This number provides *dbmake* with the information to create a sufficiently large initial extent (when the table is created) to hold this number of rows in a single extent. This can reduce the possibility of the table using several extents just for the initial loading.
 - The second integer indicates the maximum number of rows expected to be in the table. You can use this number to indicate the speed of growth for the table. For example, the State table (st_table) does not increase in size, so you can use the default value. The General Ledger Transaction record (gltr_rec) in the financial area will continue to increase in size, so you should indicate a large number for this second integer.
7. The default size of an extent is eight (8) pages, which translates to 16 KB on most platforms. Since the database engine does not automatically know table sizes, table space cannot be preallocated. Therefore, the database adds extents only as they are needed. The ROWLIMITS attribute provides a way to indicate the initial size of a table and to indicate the expected rate of growth of a table. Because the engine does have a limit for the number of extents a table may have, the ROWLIMITS attribute provides a way to automatically reduce the number of extents needed for the table.

Note: If you use a dbspace that does not have enough contiguous space for an extent of the specified size, the database engine allocates the largest available contiguous block for the extent. Thus, the engine can create extents that are smaller than your specified number in the schema. The *dbmake* utility attempts to use your specified initial number of rows (the first number in the rowlimits attribute value), or the actual number of rows in the table, whichever is larger, to calculate the size of the initial extent for the revised table when:

- You are rebuilding a table because you changed a field size or type
 - You are rebuilding a table because you added or deleted a field
 - You invoked the *dbmake* -R option
8. This feature provides a method for reducing the number of extents in use by the table for already existing rows, as long as a large enough contiguous space is available. It also maximizes the amount of contiguous space used for the table, which is more efficient for the applications accessing that table.
 9. The status of the file. If the status is "Active", this line is optional. If the status is "Inactive", entry of this line is required. Valid statuses include the following:

- Active File is in use
- Inactive File exists but is not being used

10. The text description of the file. This entry may include either one or two lines, each with a maximum of 60 characters. Or, the entry may be one line of 120 characters. However, if a 120 character line is used, the system will divide it into two lines, dividing at the white space closest to 60 maximum characters. The description should contain user-friendly English that describes the purpose of the file (eg. 'Contains personal information for individuals in the Id records' might be the description of the profile_rec file).
11. The track that uses this file. A track as used in the system is a major administrative area within the campus structure. Type track names out in full with the initial letter capitalized, and enter them in quotes (eg., "Common"). Valid track names include the following:
 - A(dmissions)
 - C(ommon)
 - D(evelopment)
 - F(inancial)
 - M(anagement)
 - S(tudent)

Column Section

In the COLUMN section, you define each field in the schema, including the following basic information about each field:

- The field name
- The type of field. Valid types include the following:
 - character (contains letters, numbers and symbols)
 - long (contains integers)
 - integer (contains whole numbers)
 - logical (single character column, expected to have a value of Y/N)
 - date (format mm/dd/yy)
 - serial (fields are assigned sequential serial numbers)
 - double (contains signed floating point numbers)
 - money (formatted as \$000.00)
 - float (contains single precision floating point numbers)
- The length of the field, specified in a parameter, for example: addr1(24)
- The description, heading, and text entries for the column, which are stored in the Database Field record (dbfield_rec)

Text and Description

Follow the following standards for Text and Description:

IQ displays the text descriptions of all the fields of a selected view alphabetically. When deciding upon the text of a field, keep in mind what the user is going to see (an alphabetically arranged list of fields) and what word the user will most likely look for first to find this field. That word should be the first word of the field text.

With field headings, keep this important question in mind, "What heading would I typically like to see over this field?". Keep in mind the amount of space required for displaying the field. It is understood that different heading may be desired for the same field in different reports based on how that field is being used.

- The heading should typically be one word.
- The heading should be abbreviated, if an excessive amount of space will be wasted due to a heading that is longer than the field being displayed.
- Do not use abbreviations in headings if the spaces necessary for the data allow the use of the unabbreviated form of the word.

- Eliminate words in the text of the headings wherever possible rather than use abbreviations. Remember that the title of the report in many cases will make the repetition of the text field in the header wordy.

Example: text: Count of Prospects Moved
 heading: Cnt Prsp Moved changed to Prospects Moved

Note: The heading entry is not utilized currently.

The description entry consists of two 60 character lines giving a total of 120 characters maximum to describe the field. The description should be in user-friendly English and should define how the field is used. For example, the description of the stuac_cl field might read 'Identifies the academic classification during this session for this student'. The description entry may be used as a prompt when this field is accessed by some CX programs.

Common Types of Fields

Many examples of fields exist that appear in numerous files: Session Code, Year, Program, Last Update Date, Contact, Fund, Account, etc. Whenever a field is encountered that appears in other files as well, the field text should include an indication of which file that this field is from.

- Program - Admission
- Program - Exam
- Session - Admission
- Session - Exam

Note: Use of past tense with ed is up to track.

The following are standards for fields.

Keys

The heading and the text should be:

- Primary Composite Key
- Primary Key
- Composite Key
- Composite Key 1
- Composite Key 2
- Composite Key 3

Numbers

In text do not use numbers unless necessary. In Headings use numbers and eliminate spacing between text and number.

Example: text: Major - First
Example: Major - Second
Example: heading: Major1
Example: Major2
Example: text: Enroll Status - Second
Example: heading: Enr Status2

ID Numbers

The abbreviation should be uppercase ID and Number (eg ID Number).

Text

Enter the ID number in text with the ID Number first. ID numbers should always indicate type, i.e. Spell out number in the text.

- ID Number

- ID - Counselor
- ID - Previous
- ID - Secondary
- ID - Employment Business
- ID - Match Gift Business

Heading

Enter ID Number in headings with the ID number last.

- ID
- Counselor ID
- Previous ID
- Secondary ID
- Employment Business ID
- Match Gift Business ID

Dashes

Use (-) if there is room. Abbreviate or dash for clarity.

Example: Contact Status - Lead Ctc (too long >24 char limit)
 Contact Status- Lead Ctc (next choice)
 Contact Status-Lead Ctc (next choice)

Dates

Do not use ed for dates.

Text

The text should not be as short as Add Date or Date because this is not clear enough. Date should follow the modifier. The location of the file reference is a track decision. Use begin instead of beg.

Example: Position Begin Date or Begin Date - Position
 ID Update Date or Update Date - ID

Headings

Generally headings should not include the word date. It is self-evident that the field is a date.

Example: text: Host Last Contact Date
 heading: Last Contact or
 Last Ctc - Host or
 Last Ctc

Code & Description

- In text use the word code
- In text use the word description rather than text
- In headings do not use the word code
- In headings do not use the word description

Example: taid_tbcodes
 text Aid Code
 heading Aid
 taid_text
 text Aid Description
 heading Aid

Note: Probably both will not appear in the same report. If they do, the field lengths will delineate.

Flag(s)

Can we easily show that it is a logical field in the text? Use uppercase Y/N with dash rules applied.

- Transfer - Y/N
- Transfer - Y/N
- Transfer -Y/N

Miscellaneous Abbreviations

GL for General Ledger

Journal for Voucher

Record Number for serial (other than id_no)

Number should be abbreviated as No in headings.

Constraint Section

In the CONSTRAINT section, you specify constraints, which are rules that must be satisfied whenever a program attempts to insert, update, or delete data in a table.

Index Section

The INDEX section specifies those indexes required for:

- A specific application program
- An ACE report defining the primary index
- A composite join(s) in a PERFORM screen
- Creating data integrity (indexes without duplicates)

Note: The system assumes that Indexes are ascending unless you declare the DESC (descending) attribute.

Grant Section

The GRANT section specifies access permissions for users and groups. Access types include:

- alter
- control
- delete
- index
- insert
- read
- select

You specify triggers and audit trails within this section.

Note: To help make merging easier, you should keep access-types in alphabetical order.

Keys

The following are the standards for keys.

Primary Key:

- If composite key is a primary key, the suffix will be '_prim'.
- If serial, the suffix should be '_no' (it may be necessary to use a suffix of '_serial' on some files because of conflicts with another field needing a suffix of '_no').
- Each file will have the primary key explicitly defined within the schema. A primary key will be allocated in the file even if not specified, so it is better to make that key accessible.

Composite Key

Other than composite primary keys, the suffix will be '_keyx' where 'x' is 1, 2, 3, etc. The first composite will have the suffix '_key1' followed by '_key2', etc.

Indexes

Indexes will not be created unless required for a specific application program, ACE report, defining the primary index, composite joins in PERFORM screens, or to create data integrity (indexes without duplicates) The Track Coordinators must approve any additional indexes before they are created. Unnecessary indexes are a CPU & disk resource drain, therefore keep them to the bare minimum.

Note: If duplicate records are permissible for the key, use 'allow dups' rather than just 'dups' for security reasons and consistency.

Use of Indexes:

The use of indexes should be commented by program, ACE report, and PERFORM screen.

Naming of Suffixes

1. _code implies that there is a table from where the valid data values are specified
2. _tbcodes implies that the code field is part of a table
3. _id implies that the field joins with id_no
4. _date implies type date
5. _time implies type integer for a date
6. Numeric numbering only if field used as part of an array. Use one underscore separating numeral from the rest.
7. (Problem: W2, 941, W4, 1040, etc. with payroll, taxes)
8. No numerics at beginning of field name
9. All field names will consist of lower case letters and underscores
10. _amt implies type money, but use it only when you need to clarify the meaning of the field name
11. 1_no where _no appears at the tail end of the field name implies type serial, link with type serial, or an identification value (e.g., ss_no for Social Security Number)
12. 1id_no for Id Number
13. 1_no_ as an accumulator where _no_ is preceded by the schema abbreviation and followed by the abbreviation of what is being counted.

Standard Types and Lengths

1. Codes : length 2,4,8 {values will be uppercase}
2. Text descriptions for codes: length 24
3. Some exceptions where longer fields are needed - either 32 or 40
4. Logical fields: length 1
5. Hours: type double
6. Dates: type date
7. Time: type long

Note: Use type int for type integer

Use type char for type character

Standard Schema Abbreviations

The following lists the standard schema abbreviations.

In Place Of	Field	Text	Heading
abbreviation	abbr	Abbreviation	Abbrev
academic	acad	Academic	Acad
academic advisor	adv	Academic Advisor	Acad Adv
academic year	ay	Year - Academic	Year
accept	acc	Accept	Accept
accessment	access	Access	Access
accomplishment	accomp	Accomplishment	Accomp
account, accounting	acct	Account, Accounting	Acct
accounts payable	ap	Accounts Payable	Acct Pay
accounts receivable	ar	Accounts Receivable	Acct Rec
accumulate	accum	Accumulate	Accum
achievement	achieve	Achievement	Achiev
acknowledgement	ack	Acknowledgement	Acknowledge
acquire	acq	Acquire	Acquire
actual	act	Actual	Actual
add/drop	adrp	Add/Drop	Add/Drop
address	addr	Address	Addr
adjusted, adjustment	adj	Adjusted, Adjustment	Adj
admissions office	adm	Admissions Office	Adm
advance	adv	Advance	Advance
advisor	(NOT USED)	Advisor	Adv
agency	agc	Agency	Agency
agreement	agree	Agreement	Agree
agriculture	agri	Agriculture	Ag
alimony	alim	Alimony	Alim
allocate	alloc	Allocate	Alloc
allow, allowable, allowance	allow	Allow, allowable, allowance	Allow
alternate	alt	Alternate	Alt
alternate address	aa	Alternate Address	Alt Adr
alumni, alumnae, alumnus	alum	Alumni	Alum
American College Testing - ACT	act	ACT	ACT
amount	amt	Amount	Amt
annual	an	Annual	Annual
anonymous	anon	Anonymous	Anon
appeal	appl	Appeal	Appeal
applicant	app	Applicant	Applic
application	app	Application	Applic
apply	app	Apply	Apply
approval	appr	Approval	Approv
asset	asset	Asset	Asset
assignment	assgn	Assignment	Assgn
assistant	asst	Assistant	Asst
associated student body	asb	Associated Student Body	ASB
association	assoc	Association	Assoc
attempt	att	Attempt	Att
attend	att	Attend	Att
audit	au	Audit	Au
authorize	auth	Authorize	Auth
automatic	auto	Automatic	Auto
available	avail	Available	Avail
avenue	ave	Avenue	Ave
average	avg	Average	Avg
award	awd	Award	Award, Awd
axis	axis	Axis	Axis
balance	bal	Balance	Bal
basis	basis	Basis	Basis
begin, beginning, start	beg	Begin, Beginning	Beg
benefit	ben	Benefit	Ben
bi-monthly	bmo	Bi-Monthly	Bi-Mon
bi-weekly	bwk	Bi-Weekly	Bi-Wk
block	blk	Block	Blk
bookeeping	bkpg	Bookeeping	Bkpg
break	brk	Break	Brk
budget	bgt	Budget	Bgt
budget amount	bgtamt	Budget Amount	Bgt Amt
budget calendar	bgtcal	Budget Calendar	Bgt Cal
budget summarization	bgtsum	Budget Summarization	Bgt Sum
build	bld	Build	Build
building	bldg	Building	Bldg
business	bus	Business	Bus
calculate, calculation	(Not Used)	Calculate, Calculation	Calc
calendar	cal	Calendar	Cal
campaign	camp	Campaign	Camp
campus	(Not Used)	Campus	Campus
campus based funds (financial aid)	cbf	Campus Based Funds	Campus Funds
cancel	cancel	Cancel	Cancel
career	career	Career	Career
career position	crpos	Career Position	Career Pos

career skill	crskl	Career Skill	Career Skl
carry forward	cfwd	Carry Forward	Carry Fwd
catalog	cat	Catalog	Cat
category	ctg	Category	Ctgy
center	cntr	Center	Cntr
chapter	chap	Chapter	Chap
charge	chg	Charge	Chg
check	ck	Check	Ck
church	church	Church	Church
circumstance	circ	Circumstance	Circumstance
classification	cl	Class	Class
classroom	clrm	Classroom	Room
client	clnt	Client	Client
close	cls	Close	Close
college	col	College	Coll
combination	combo	Combination	Combin
comment	comment	Comment	Comment
communication	comm	Communication	Comm
comparison	(NOT USED)	Comparison	Cmpr
compensation	comp	Compensation	Comp
complete	cmpl	Complete	Cmpl
comply, compliance	cmPLY	Comply, Compliance	Comply
compute	cmpt	Compute	Compute
computer	cmpr	Computer	Computer
concentration	conc	Concentration	Conc
condition	cond	Condition	Cond
conference	conf	Conference	Conf
confirm	conf	Confirm	Confirm
constituent, constituency	cons	Constituent, Constituency	Const
constituent status	consstat	Constituent Status	Const Stat
constraints	constr	Constraints	Constraints
contact	ctc	Contact	Contact
continue, continuing	contin	Continue, Continuing	Cont
contract	ctrc	Contract	Contract
contract-to-date	ctd	Contract-to-date	Contract-to-date
contribution	cont	Contribution	Contrib
control	ctrl	Control	Control
core guideline	core	Core Guideline	Core
corporation	corp	Corporation	Corp
correspondence	corresp	Correspondence	Corresp
correspondent	sender	Correspondent	Correspdt
counselor	cnslr	Counselor	Counselor
count	cnt	Count	Count
country	ctry	Country	Country
county	cty	County	County
course	crs	Course	Course
course work	cw	Course Work	Crs Wk
credit	cr	Credit	Credit
credit hours	hrs	Credit Hours	Credit Hrs
criteria, criterion	(Not Used)	Criteria, Criterion	Crit
cross reference	crref	Cross Reference	Cross Ref
cumulative	cum	Cumulative	Cum
currency	crncy	Currency	Currency
current	cur	Current	Current
current year	cy	Year - Current	Year
customer	cust	Customer	Customer
daily	dy	Daily	Daily
date added	add_date	Added Date	Added
dean of students	dean	Dean of Students	Dean/Students
debit	dr	Debit	Debit
deceased	decsd	Deceased	Deceased
decision	dec	Decision	Decision
declare	decl	Declare	Declare
decrement	decr	Decrement	Decrement
deduction	ded	Deduction	Deduct
default	def	Default	Default
defer	dfr	Defer	Defer
deferred gift	pldg	Deferred Gift	Defer Gift
degree	deg	Degree	Degree
delivery	deliv	Delivery	Delivery
denomination	denom	Denomination	Denom
department	dept	Department	Dept
dependent, dependency	dep	Dependent, Dependency	Depend
deposit	dep	Deposit	Deposit
description	desc	Description	Desc
designation	desg	Designation	Desig
desire	des	Desire	Desire
detail	dtl	Detail	Dtl
development	dev	Development	Dev
directed study	ds	Directed Study	Dir Stdy
director, directory	dir	Directory	Dir
disbursement, disburse	disb	Disbursement, Disburse	Disb
discount	disc	Discount	Disc
discretionary	(Not Used)	Discretionary	Disc

display	dsply	Display	Disp
dispose	disp	Dispose	Disp
distribution	dist	Distribution, Distrib	Dist
dividend	div	Dividend	Div
division	div	Division	Div
divorced {do not use}	div	Divorced	Div
document	doc	Document	Doc
donor	donor	Donor	Don
donor accounting	da	DA	DA
double time	dt	Double Time, Dbl Time	DT
each	each	Each	Ea
earn	earn	Earn	Earn
earned income credit	eic	Earn Income Cred	EIC
education	ed	Education	Educ
electricity, electronics	elec	Electricity, Electronics	Elec
emergency	emer	Emergency	Emer
employ	emp	Employ	Emp
employee	empl	Employee	Empl
employer	empr	Employer	Empr
encumbrance	enc	Encumbrance	Enc
ending	end	End, Ending	End
engineer	enr	Engineer	Engr
english	eng	English	Eng
enroll, enrollment	enr	Enroll	Enroll
entry	ent	Entry	Ent
equivalent	eqv	Equivalent, Equiv	Equiv
error	err	Error	Err
estimate	est	Estimate	Est
ethnic	ethnic	Ethnic	Ethnic
event	evnt	Event	Evnt
examination	exam	Exam	Exam
example	ex	Example	Ex
exceptional financial need (finaid)	efn	Except Fin Need	EFN
exclusive	excl	Exclusive	Excl
exemption	exempt	Exempt	Exempt
expect	exp	Expect	Exp
expected family contribution(finaid)	efc	Exp Family Cont	EFC
expense	exp	Expense	Exp
experience	exper	Experience, Exper	Exper
extension, extend, extent	ext	Extension	Ext
facility	facil	Facility	Facil
factor	fctr	Factor	Fctr
faculty	fac	Faculty	Fac
family	fam	Family	Fam
family financial statement (finaid)	ffs	Family Financial Stmt	FFS
federal	fed	Federal	Fed
federal income tax (FIT)	fit	Fed Income Tax	FIT
federal insurance compliance act(FICA)	fica	FICA	FICA
file name	file	File	File
financial	fin	Financial	Fin
financial aid	fa	Financial Aid	Finaid
financial aid form	faf	Fin Aid Form	FAF
financial aid transcript	fat	Fin Aid Transcript	FAT
financial statement	fs	Financial Stmt	Fin Stmt
finish	end	Finish	End
fiscal	fscl	Fiscal	Fscl
fiscal year	fy	Fiscal Yr.	FY
fiscal year-to-date	ftd	Fiscal Yr to Date	FTD
fix	fix	Fix	Fix
forecast	fcst	Forecast	Fcst
foreign	fgn	Foreign	Fgn
form	frm	Form	Frm
form of payment	pay_form	Form of Payment	Paymt Form
foundation	fnd	Foundation	Found
frequency	freq	Frequency, Freq	Freq
freshman	fr	Freshman	FR
from	fr	From	Fr
full-time	ft	Full-Time	FT
full-time equivalent	fte	Full-Time Equiv	FTE
function	func	Function	Func
fund balance	fb	Fund Balance, Fund Bal	Fund Bal
funding	(Not Used)	Funding	Fndg
general ledger	gl	General Ledger, GL	GL
general ledger acct	gla	GL Account	GL Acct
general ledger define	gld	GL Define	GLD
general ledger entry	gle	GL Entry	GL Ent
general ledger transaction	gltr	GL Transaction, GL Trans	GL Trans
general ledger unit code	(NOT USED)	GL Unit Code	GL Unit
giving	gvg	Giving	Gvg
grade	grd	Grade	Grd
grade point average	gpa	GPA	GPA
graduate, graduation	grad	Graduate, Graduation	Grad

gross pay group	gp	Gross Pay Group	Gross Group
group id	gid	Group ID	Group ID
groups	grps	Groups	Grps
health	hlth	Health	Hlth
high	high	High	High
high school	hs	High School	HS
history	hist	History	Hist
holiday	hol	Holiday	Hol
hours	hrs	Hours, Hrs	Hrs
housing	hsg	Housing	Hsg
id number	id	ID Number	ID
income	inc	Income	Inc
incorporated	inc	Incorporated	Incorp
increment	incr	Increment	Incr
independent	indep	Independent	Ind, Indep
independent study	is	Independent Study	Ind Stdy
index	idx	Index	Idx
indicator, indication	ind	Indicatory, Indication	Ind, Indic
information	info	Information, Info	Info
initial, initiate	init	Initial, Initiate	Init
inquiry	inq	Inquiry	Inq
install	install	Install	Install
institution	inst	Institution, Instit.	Inst
instruction	instr	Instruction, Instruct	Instr
insurance	ins	Insurance	Ins
intend	plan	Intend	Plan
interest	int	Interest	Int
internship	intern	Intern	Intern
interrupt	intr	Interrupt	Inter
interview	intvw	Interview	Intrvw
inventory	inven	Inventory	Inv
investment	invest	Investment, Investmt	Invest
invoice	inv	Invoice	Invoice
involvement	involve	Involve	Invl
item	item	Item	Item
journal	(NOT USED)	Journal	Jrnl
junior	jr	Junior, Jr.	Jr
lab	lab	Lab	Lab
label	lbl	Label	Lbl
large	lg	Large	Lg
last	last	Last	Last
lead	lead	Lead	Lead
lender	lnd	Lender	Lnd
letter	let	Letter	Let
level	lev	Level	Lev
liability	liab	Liability	Liab
library, librarian	lib	Library, Librarian	Libr, Lib
lifetime-to-date	ltd	Lifetime to Date	LTD
line 1 (e.g. addr_line1)	line1	Line 1	Line1
load	load	Load	Load
local income tax	lit	Local Inc Tax	LIT
location	loc	Location	Loc
mail room	mail	Mail Room, Mail Rm	Mail
maintenance	maint	Maintenance	Maint
management	mgmt	Managment, Managemt	Mgmt
manual	man	Manual	Man
marital	mrtl	Marital	Mrtl
market	mkt	Market	Mkt
marketing	mktg	Marketing	Mktg
match matching	mtch	Match, Matching	Mtch
matriculate, matriculation	matric	Matriculate, Matric	Matric
maximum	max	Maximum	Max
maximum lifetime	(Not Used)	Maximum Lifetime	Max Life
mechanic	mech	Mechanic	Mech
meeting	mtg	Meeting	Mtg
membership	mem	Membership	Memb
memorial	mmrl	Memorial	Mem
method	meth	Method	Meth
minimum	min	Minimum	Min
minute	min	Minute	Min
month, monthly	mo	Month, Monthly	Mth
month-to-date	mtd	Month to Date	MTD
nature	nat	Nature	Nat
net pay	nt	Net Pay	Net
newspaper	news	Newspaper	News
next	next	Next	Next
non-catalog course work	ncat	Non-Catalog Course Wrk	N-Cat CW
non-taxable income	nontax	Non-Taxable Income	N-Tax Inc

notes payable	np	Notes Pay	Notes Pay
notes receivable	nr	Notes Recv	Notes Recv
notify, notification	notif	Notification, Notif.	Notif
number	no	Number	No
nurse, nursing	nurs	Nurse, Nursing	Nurse, Nurs
object	obj	Object	Obj
occupancy	occ	Occupancy	Occ
occupant	occ	Occupant	Occ
occupation	occ	Occupation	Occ
office	ofc	Office	Ofc
official	ofcl	Official	Ofcl
old	o(-)o	Old	Old
organization	org	Organization, Organ.	Org
original	orig	Original	Orig
other	oth	Other	Other, Oth
output	o(.)out	Output	Out
overtime	ot	Overtime	OT
package	pkg	Package	Pkg
paid	pd	Paid	Pd
parameter	(NOT USED)	Parameter	Parm
parent	par	Parent	Par
parental contribution (finaid)	pc	Parental Contribution	Par Contr
park	prk	Park	Prk
parking	prkg	Parking	Prkg
parttime	pt	Parttime	PT
payment	pay	Payment	Pmt
payroll	pr	Payroll	Payroll
pension	pens	Pension	Pens
percent, percentage	pct	Percent, Percentage	Pct
perennial	peren	Perennial	Peren
period	prd	Period	Prd
period-to-date	ptd	Period-to-Date	PTD
permanent	perm	Permanent	Perm
permit	prmt	Permit	Prmt
personal	prs	Personal	Prs, Pers
personnel	pers	Personnel	Pers
perspective	persp	Perspective	Persp
physical plant	plant	Phys Plant, Plant	Plant
plan	plan	Plan	Plan
points	pts	Points	Pts
policy	pol	Policy	Pol
position	pos	Position	Pos
post, posting	pst	Post, Posting	Pst
preference, preferred	pref	Preference	Pref
prenotification	prenotif	Prenotification	Prenotif
preparation	prep	Preparation	Prep
prepare	prep	Prepare	Prep
preparer	prep	Preparer	Prep
prerequisite	prereq	Prerequisite	Prereq
present	present	Present	Pres
president	pres	President	Pres
previous	prev	Previous	Prev
primary	prim	Primary	Prim
print, printing	prnt	Print	Prnt
priority	prior	Priority	Prior
probation	prb	Probation	Prb
process, procedure	proc	Process, Procedure	Proc
program	prog	Program	Prog
programmer	pgmr	Programmer	Pgmr
project	proj	Project	Proj
prospect, prospective	prsp	Prospect, Prospective	Prsp
publicity	pub	Publicity	Pub
purchase order	po	Purch Order	PO
purchasing, purchased	purch	Purchasing	Purch
purge	purge	Purge	Purge
purpose	purp	Purpose	Purp
quality	qual	Quality	Qual
quantity	qty	Quantity	Qty
quarter-to-date	qtd	Qtr-to-Date	QTD
quarterly	qt	Quarterly	Qtrly
race	ethnic	Ethnic	Ethnic
real estate	re	Real Estate	RE
reapply	reapp	Reapply	Reapp
receipt	rcpt	Receipt	Rcpt
receiver	rcv	Receiver	Recv
recipient, correspondee	recip	Recipient	Recip
recommendation	recom	Recommendation	Recommend
reconcile, reconciliation	recon	Reconcile	Recon
record	rec	Record	Rec
recruit	rcrt	Recruit	Rcrt

reference	ref	Reference	Ref
refund	rfrnd	Refund	Rfrnd
registered, registration	reg	Registration	Regist
rejected	rej	Rejected	Rej
relationship	rel	Relationship	Rel
remainder	rmdr	Remainder	Rmdr
remark	rem	Remark	Rem
reminder	rem	Reminder	Rem
repeat	rep	Repeat	Rep
report	rpt	Report	Rpt
representative	rep	Representative	Rep
request, requisition	req	Request	Req
require	req	Require	Req
reserve, reservation	rsv	Reserve	Rsv
residence, resident	res	Residence, Resident	Res
resource	resrc	Resource	Resrc
response	rsp	Response	Rsp
responsible, responsibility	resp	Responsible	Resp
restrict	rstr	Restrict	Rstr
resume	resm	Resume	Res
return	ret	Return	Ret
revenue	rev	Revenue	Rev
review	rvw	Review	Revw
revision	rev	Revision	Rev
sabbatical	sabb	Sabbatical	Sabb
salary	sal	Salary	Sal
salutation	salut	Salutation	Salut
salvage	slvg	Salvage	Slvg
savings	sav	Savings	Svgs
schedule	schd	Schedule	Schd
Scholastic Achievement Test (SAT)	sat	SAT	SAT
school	sch	School	Sch
score	score	Score	Score
screen	scr	Screen	Scr
secondary	sec	Secondary	Sec
section	sec	Section	Sec
semester	sess	Semester	Sess
semi-monthly	sm	Semi-Monthly	SM
senior	sr	Senior, Sr	Sr
separate	sep	Separate	Sep
service	serv	Service	Serv
session	sess	Session	Sess
shorthand	shand	Shorthand	Shand
skill	skl	Skill	Skill
social security	(NOT USED)	Social Security	Soc Sec
social security number	ss_no	Social Security Number	Soc Sec No
solicit, solicitation	sol	Solicitation	Sol
sophomore	so	Soph, Sophomore	So
source	src	Source	Src
span	span	Span	Span
special	spec	Special	Spec
sponsor	spon	Sponsor	Spon
spouse	sp	Spouse	Sp
square feet, square footage	sqft	Square Ft	Sq Ft
staff	staff	Staff	Staff
standard	(NOT USED)	Standard	Std
state	st	State	St
state income tax (SIT)	sit	State Inc Tax	St Inc Tax
statement	stmt	Statement	Stmt
station	stnt	Station	Stn
status	stat	Status	Stat
statutory	sty	Statutory	Sty
step	step	Step	Step
straight	strt	Straight	Strt
street	str	Street	St
string	strg	String	Strg
student	stu	Student	Stu
subprogram, sub-program	subprog	Subprogram	Subprog
subsession, sub-session, module, block	subsess	Subsession	Subsess
subsidiary	sub	Subsidiary	Sub
subsidiary	subs	Subsidiary	Subs
subsidiary account	sa	Subsidiary Account	Subs Acct
subsidiary account	suba	Subsidiary Account	Subs Acct
subsidiary balance	subb	Subsidiary Balance	Subs Bal
subsidiary entry	sube	Subsidiary Entry	Subs Ent
subsidiary total	subt	Subsidiary Total	Subs Tot
subsidiary transaction	subtr	Subsidiary Transaction	Subs Trans
suite	suite	Suite	Suite
summarize	sum	Summarize	Sum
summer	su	Summer	Summ
supplemental	suppl	Supplemental	Supl
supplier	splr	Supplier	Splr

support	supp	Support	Supp
table	tb	Table	Tbl
taken	taken	Taken	Taken
taxable	tax	Tax	Tax
taxable income	taxinc	Taxable Income	Tax Inc
teachers insurance annuity association	tiaa	TIAA	TIAA
telephone	phone	Phone	Phone, Ph
temporary	temp	Temporary	Temp
term	sess	Session	Sess
terminate	term	Terminate, Termination	Term
terms of payment	pay_terms	Terms of Payment	Pmt Trms
textbook	text	Text	Text
tickler	tick	Tickler	Tick
total	tot	Total	Tot
total family income (fina id)	tfc	Total Family Income	Tot Fam Inc
track	trk	Track	Trk
transaction	tr	Transaction	Trans
transcript	trans	Transcript	Trans
transcript comment	tc	Transcript Comment	Trans Comm
transfer	trf	Transfer	Trans
transfer work	tw	Transfer Work	Trans Wk
tuition	tuit	Tuition	Tuit
typing	typg	Typing	Typing
uncollected	uncol	Uncollected	Uncol
unemployed	unempl	Unemployed	Unemp
unit	unit	Unit	Unit
update	upd	Update	Upd
user id	uid	ID - User	ID
vacation	vac	Vacation	Vac
value	val	Value	Val
vehicle	veh	Vehicle	Veh
vendor	vnd	Vendor	Vnd
veteran	vet	Veteran	Vet
volunteer	vol	Volunteer	Vol
voucher	vch	Journal	Jrnl
waived	wvd	Waived	Wvd
waiver	wvr	Waiver	Wvr
week, weekly	wk	Week, Weekly	Wk
weight	wt	Weight	Wt
with	w	With	W
withdraw	wd	Withdraw	WD
words per minute	wpm	Words Per Minute, WPM	WPM
work	(NOT USED)	Work	Work, Wrk
year	yr	Year - Academic	Year
year	yr	Year - Award	Year
year	yr	Year - Calendar	Year
year	yr	Year - Fiscal	Year
year-to-date	ytd	Year-To-Date	YTD
zip code, postal code	zip	Zip Code	Zip

User Interface Standards: Menu Source

Introduction

These user interface standards are for menu source (menusr) files. Use these standards when creating or modifying this type of file.

General Conventions

The following are general conventions for menu source (menusr) files.

1. The term "master" will only be used in the "CX Master Menu".
2. Use the terminology "main" menu in the title line (TI) of the primary menusr and all secondary menusr when they are menus.

Example: TI=Alumni/Development: Main Menu [primary menusr]
TI=Alumni Association: Main Menu [secondary menusr]
TI=Development: Main Menu [secondary menusr]
TI=Donor Accounting: Main Menu [secondary menusr]

3. The term "office" will not be used in the title (TI), short (SD) or long description (LD) lines.
4. Long descriptions (LD) will end with the word Menu or Menus as appropriate.

Example: LD=Registrar Menu

5. The short descriptions of the parent menu will be used for the long description of the active menu if it is a collection of sub menus.

Example: LD=Registrar, Student Services, Financial Aid, Placement Menus

6. The preferred order for listing the reports sub-menu is alphabetical. You may group them logically if that is more appropriate.
7. Group tables alphabetically within function.

Example: Registrar: Table Maintenance Menu
[a] Registrar (A-L) [d] Common (D-F)
[b] Registrar (M-Z) [e] Common (G-O)
[c] Common (A-C) [f] Common (P-Z)

8. List menuopt files in the order of the processed steps. If a process contains five steps, the menuopts should appear on the menu in the order of occurrence.
9. Likewise, a menu screen containing sub-menus should also appear in the order and/or frequency of occurrence. For example, a menu option containing "Session Processing" functions should appear after the daily processing functions.
10. If a menu requires a sub-menu for "Table Maintenance", it should appear as the last menu option.

Example: Student Management: Registrar Main Menu
[a] Course/Class Schedules [h] Transcripts
[b] Data Entry [i] Program/Degree Audit
[c] Registration [j] Forms
[d] Block Registration [k] Letters

[e]	Class Lists	[l]	Reports
[f]	Grading	[m]	Session Processing
[g]	Anonymous Grading	[n]	Table Maintenance

11. The left portion of the title (TI) of sub-menus will retain the name of the module associated with the sub-menu functions.

Example: TI=Alumni/Development: Main Menu	[primary menusr]
TI=Alumni Association: Main Menu	[secondary menusr]
TI=Alumni Association: Forms Menu	[tertiary menusr]
TI=Alumni Association: Query by Form	[tertiary menusr]
TI=Alumni Association: Reports Menu	[tertiary menusr]

Punctuation

The following are general conventions for menu source (menusr) punctuation.

1. Menu options which are menus will have an "*" to the left of the option to denote that it is a menu.

Example: * (a) Recruiting/Admissions

2. This feature will be controlled through the menu processor and may be enabled or disabled. This capability will be provided in a future SMO.
3. Prepositions which are used in short (SD), long (LD) or run (RD) description titles will be in lowercase.

Example: by, for, in, of, to, with.

4. Articles which are used in short (SD), long (LD) or run (RD) description titles will be in lower case.

Example: a, an, the.

5. The word "and" will always be spelled out. The "&" sign will not be used.
6. The slash "/" will be used as the standard separator where space limitations occur in the short description (SD).

Example: SD=Majors/Minors/Concentration

7. Abbreviations should not be used on the menu as a general rule. The (SD), (LD), (RD), and the Title (TI) should not use abbreviations if at all possible.
8. The term "Reports" will be used throughout the system. If necessary, an abbreviation of a word is allowed, but it must be a truncation of the word and be followed by a ".".

Example: SD=Program/Program Regist.

9. This example indicates that the menuopt will call the "Program and Program Registration Tables. The acceptable abbreviated version of our example contains the COMPLETE spelling (including vowels) of the word followed by the ".". An abbreviation "Rgst", or any version of this kind is NOT ACCEPTABLE.
10. Colons ":" will not be used in long descriptions (LD) or short descriptions (SD).

Macros

The following are general conventions for menu source (menusr) macros.

1. The m4_keepif/m4_keepend macros will be used in menusr to handle optional menu

options (eg: block registration), depending upon whether the client has purchased or will be using particular modules or options. The macros will start with ENABLE as a standard convention. In the file \$CARSPATH/macros/custom/student, the following macro would be defined:

Example: `{*** Defines whether block registration is enabled ***}
`m4_define(ENABLE_FEAT_BLK_REG,`Y')`

2. In the file \$CARSPATH/menusr/student/regist/menudesc the corresponding lines will appear:

Example: `MNU_OPT(regist/programs/stue)
MNU_SUB(student/regist/regist)
`m4_keepif(ENABLE_FEAT_BLK_REG,`Y')
MNU_SUB(student/regist/blockreg)
`m4_keepend'
MNU_SUB(student/regist/classlist)`

3. In this example, if the ENABLE_FEAT_BLK_REG macro is set to Y in the \$CARSPATH/macro/custom/student file, then the "MNU_SUB(student/regist/blockreg)" option will be displayed on the menu.
4. The ENABLE type macros are used within the "`m4_keepif and m4_keepend" structure within a menudesc file.

Example: ``m4_keepif(ENABLE_MOD_REGIST,`Y')
`MNU_SUB(student/regist)
`m4_keepend'
`m4_keepif(ENABLE_MOD_FINAID,`Y')
`MNU_SUB(student/finaid)
`m4_keepend'
`m4_keepif(ENABLE_MOD_STUSERV,`Y')
`MNU_SUB(student/stuserv)
`m4_keepend'
`m4_keepif(ENABLE_MOD_PLACEMENT,`Y')
`MNU_SUB(student/placement)
`m4_keepend'`

User Interface Standards: Menu Options

Introduction

Interface standards are for menu option (menuopt) files. Use these standards when creating or modifying this type of file.

General Conventions

The following are general conventions for menu option (menuopt) files.

1. Database names of files and records (eg: stu_acad_rec) will not appear in SD's, LD's, RD's or run description text. Instead, the complete file name will be used.

Example: student academic record

2. The letter production system (LPS) and the forms production system (FPS) will only be referred to in the LD and RD lines.

Example: Prints Admissions Letters Using the Letter Production System (LPS)

3. The long (LD) and run (RD) description titles should be the same.
4. All runtime (RD) description lines (header) will be one line and will be underlined on the following line with dashes. The underline will be the same length as the header line.
5. Normal upper/lower case convention will be used in the runtime (RD) description text. If information is to be emphasized, a "NOTE: " will be used.

Example: RD=

```
`m4_center(' Print Student Data Sheets', 80)'  
`m4_center('-----', 80)'  
`m4_center(' This is the final step in generating student ', 80)'  
`m4_center(' data sheets (SDS). ', 80)'  
  
`m4_center(' The steps for selecting students and for creating', 80)'  
`m4_center(' the student data sheets must be completed prior ', 80)'  
`m4_center(' to running this option. ', 80)'  
  
`m4_center(' NOTE: Use form type "sds" for SDS without billing ', 80)'  
`m4_center(' information. ', 80)'  
`m4_center(' Use form type "sdsbill" for SDS with billing ', 80)'  
`m4_center(' information. ', 80)'
```

Note: Included in the note should be information relating to steps that must be completed prior to the current one.

6. The menuw.s file in \$CARSPATH/system/etc will be used for password protection in menusrc and enuopts.

Example: To enable the password in a menusrc or menuopt file:

PW=@REGIST

The corresponding entry in the \$CARSPATH/system/etc/menupw.s file would be:

REGIST:<password>:

7. A six character alpha descriptor (eg: REGIST) will be used with the "@" to denote the unique item in the menupw.s file. This will replace the numbers used previously.
8. All "PP=" lines will begin with "Enter" unless the prompt is a question.

Punctuation

The following are punctuation standards for menu option (menuopt) files.

1. Abbreviations "eg:", "ex:", "ie:", or "etc." will not be used in the short (SD), long (LD), or run (RD) description titles. Examples will be included in the text area of the run description (RD).
2. To denote "for example", "eg:" will be used as the standard. These should ONLY be used with parameter prompt lines (PP). The example will appear at the end of the sentence.

Example: Enter the subsidiary balance code, eg: FA90.

3. In all runtime (RD) description lines (header), the first letter of each word will be capitalized (excluding conjunctions and articles).
4. No special capitalization will be used in the runtime (RD) description (text). Record and file names will not be capitalized.
5. All "PP=" lines will use normal upper/lower case convention.
6. All "PP=" lines will end in either a "." or a "?".

Macros

The following are macro standards for menu option (menuopt) files.

1. The `QUERY_MAINT' macro is used with the (LD) and (RD) to indicate maintenance of tables/records. This macro will be associated with all menuopts which call PERFORM screens that create/update tables and/or records.
2. This macro is defined in (\$CARSPATH/macros/user/common) and an example of its definition is as follows:

Example:

```
{*** General Heading for Table/File Manipulation ***}
`m4_define(`QUERY_MAINT', `Query/Maintain')
```

3. The `QUERY_MAINT_LINE' macro is used with the `QUERY_MAINT' macro. This macro represents the header line within the (RD) taken by the `QUERY_MAINT' macro, and is the same length as the `QUERY_MAINT' macro. It too is defined in (\$CARSPATH/macros/user/common):

Example:

```
`m4_define(`QUERY_MAINT_LINE', `-----')
```

4. Example of `QUERY_MAINT' and `QUERY_MAINT_LINE' used together:

Example:

```
RD=      `m4_center(QUERY_MAINT Session Table, 80)'
`m4_center(QUERY_MAINT_LINE`-----', 80)'
```

5. These lines will expand to:

Example: Query/Maintain Session Table

6. The `RD_WIDE_OUTPUT' macro will be used as part of the runtime (RD) description in all menuopts where wide paper is required to print the report.

Example: RD= `m4_center(`Print Course Catalog', 80)'
 `m4_center(`-----', 80)'
 `RD_WIDE_OUTPUT'

7. This will expand to:

Example: Print Course Catalog

Note: This report prints on WIDE paper. You must load the appropriate paper and change the formtype for the printer.

8. The `m4_center' macro will be used in all runtime (RD) description lines, including underlines and text, to center information.

9. If another menu option is to be referenced in the body of RD text, the `m4_getoptdesc' macro will be used to bring in the short description (SD) for that option.

Example: RD=
 `m4_center(`Create Class Lists for All Courses for One Catalog', 80)'
 `m4_center(`-----', 80)'

 `m4_center(`This is the second step in generating class lists.', 80)'
 `m4_center(`Use "m4_getoptdesc(utilities/programs/fps.reglst)", 80)'
 `m4_center(`option to print the list. ', 80)'

10. The `m4_getoptdesc' macro above will expand to:

Example: "Print Classlists/Waitlists".

11. If a macro is to be used within a `m4_center' macro, it must be expanded before it is passed to the `m4_center' macro. If the parameter does not contain commas, it does not need to be quoted.

Example: (Parameter does NOT contain commas)
 `m4_center(QUERY_MAINT Session Table, 80)'
 `m4_center(QUERY_MAINT_LINE'-----', 80)'
 (Parameter does contain commas)
 `m4_center(QUERY_MAINT `Majors, Minors and Concentrations', 80)'

User Interface Standards: Program Screens

Introduction

These use interface standards are for program screen (progsr) files. Use these standards when creating or modifying this type of file.

General Conventions

The following are general conventions for program screen (progsr) files.

1. The words FORM and SCREEN will not appear on entry menus or screens.
2. The word ENTRY will be used for screen headings.

Example: ID ENTRY

3. Every screen will have a title that is capitalized and centered. Titles will not be underlined.

```

                                ID ENTRY
ID No..... [id          ]          SS No..[ssno          ]          Add Date... [adddate
]
Title..... [titl^titl_text          ]          Last Upd... [upddate ]
Name.... [iname          ]          Telephone.. [phone ]
Address. [addr1 ]
[addr2 ]
City..... [icity ]
State/Zip [st][zip          ] Country..[ctr ]
-----
-----
attributes
id = id_no, optional;
ssno = ss_no, optional,
comments = "COMMENT_SS_NO";
titl_text = title_text, optional;
titl = title, optional,
comments = "COMMENT_TITLE_TBCODE COMMENT_TBL",
default = "TITLE_DEF",
lookup titl_text joining *title_table.title_tbcodes,
upshift;
iname = name, optional,
comments = "COMMENT_NAME",
required;
addr1 = addr_line1, optional,
comments = "COMMENT_ADDR1";
addr2 = addr_line2, optional,
comments = "COMMENT_ADDR2";
icity = city, optional,
comments = "COMMENT_CITY",
required;
st_text = st_text, optional;
st = state, optional,
comments = "COMMENT_STATE COMMENT_TBL",
default = "ST_DEF",
lookup st_text joining *st_table.st_tbcodes,
upshift;
zip = zip, optional,
comments = "COMMENT_ZIP",
upshift;
country_text = ctry_text, optional;
ctr = country, optional,
comments = "COMMENT_COUNTRY COMMENT_TBL",
default = "CTRY_DEF",
lookup country_text joining *ctry_table.ctry_tbcodes,
upshift;
rct = res_ctry, optional,
comments = "COMMENT_RES_CTRY COMMENT_TBL",
default = "CTRY_DEF",
lookup country_text joining *ctry_table.ctry_tbcodes,
upshift;
cty_text = cty_text, optional;
cty = res_cty, optional,
comments = "COMMENT_RES_CTY COMMENT_TBL",
default = "CTY_DEF",
lookup cty_text joining *cty_table.cty_tbcodes,
upshift;
ts = res_st, optional,
comments = "COMMENT_RES_ST COMMENT_TBL",
default = "ST_DEF",
lookup st_text joining *st_table.st_tbcodes,
upshift;
phone = phone, optional,

```

```

comments = "COMMENT_PHONE";

adddate = add_date, optional,
default = today,
nouupdate;

upddate = last_upd_date, optional,
default = today,
nouupdate;

prof_last_upd_date = prof_last_upd_date, optional,
default = today,
nouupdate;

end

```

4. Every field will contain a comment.
5. Comments will begin with "Enter" unless the comment is a question.
6. There will be NO HARDCODING of values in progscrs with the possible exceptions of (Y) or (N). Macros will be used extensively for defaults, includes, comments, formats and examples.
7. The phrase "Valid values are:" will not be used in comments. Instead the values will be displayed in the comment.

Example: comments = "Enter joint code. (I)nformal, (F)ormal, (J)oint, (N)one."
8. If there are more valid values than can be displayed on one line, then an example macro (_EG) will be used.

Example: comments = "Enter the type of accomplishment, ACCOMP_TYPE_EG".
9. This will expand to:

Example: comments = "Enter the type of accomplishment, eg:
ACADEMIC,ATHLETIC."
10. A template is used for creating screens containing ID information. This template should also be used with PERFORM screens where possible.
11. The attributes section of progscr files will maintain the following structure:
 - Each editing clause, with the exception of the optional clause, will be listed one per line in alphabetical order and indented 5 spaces (the first letter of the edit clause is printed on the fifth position from left)
 - The optional clause will immediately follow the database field name (on same line)
 - The lookup and joining clauses will be listed together on one line
12. The menu progscr file will only contain ONE space in the selection field. This allows you to load the screen simply by typing the number. The <RETURN> is not required to call the screen.

```

-----
                                REGISTRAR DATA ENTRY MENU
-----
1.  Students           [scr1      ]
2.  Program Enrollment [scr2      ]
3.  Faculty           [scr3      ]
4.  Parents           [scr4      ]
5.  Schools           [scr5      ]
6.  Churchs           [scr6      ]

Enter Selection:      [a]

```

Note: The "Enter Selection:" field "[a]" contains only one space. This structure will access any of the data screens by entering a number.

Punctuation

The following are punctuation standards for program screen files.

1. Comments will use normal upper/lower case convention.
2. Comments will end with either a "." or a "?".
3. The "(" will be used to indicate the valid values to be entered.

Example: Enter addree style. (I)nformal, (F)ormal, (M)aiden, (N)ickname.

4. The first character of each screen descriptor will be capitalized.

Example: Alternate Address Code..[a1]

5. The colon will not be used in comments. The only exceptions will be after the word "format" and with the use of "eg".

Example: Format: mm/dd/yy.

Enter calendar year, eg: 1990.

Macros

The following are macros standards for program screens.

1. Within the area of "COMMENT" macros, several are used exclusively by the progscr files.
2. The "COMMENT_QUERY" macro is used to indicate how to activate the name search sub-routine.
3. This macro is defined in the file (\$CARSPATH/macros/custom/comment):

Example: `m4_define('COMMENT_QUERY', `Enter (0) for name query.')

4. An example of how the attribute section of the progscr file will use this macro is shown below:

Example: adv_id = adv_id, optional,
 comments = "COMMENT_ID advisor. COMMENT_QUERY",
 lookup adv_id_name joining id_rec.id_no;

5. This comment will expand to:

Example: "Enter ID# of advisor. Enter (0) for name query."

Note: "COMMENT_QUERY" and "COMMENT_TBL" will always be used in conjunction with other "Comment" macros or general comment text.

6. The "COMMENT_TBL" macro is used to indicate how to activate the "Table Lookup" feature within progscr files. It is defined in (\$CARSPATH/macros/custom/comment).

Example: `m4_define('COMMENT_TBL', ` Use CTRL T for table lookup.')

Note: This macro is used with other "COMMENT" macros or general comment text within the attribute section of a progscr file, as shown below:

Example: stuac_prog = stuac_prog,
 comments = "COMMENT_PROG COMMENT_TBL",
 lookup text1 joining prog_table.prog_tbcode,
 nouupdate,
 scroll = (c0, c1, c2, c3),
 upshift;

7. This will expand to:

Example: "Enter program code. Use CTRL T for table lookup."

Note: "COMMENT_TBL" (in its `m4_define' clause) begins with a space as its first character. Therefore, only ONE space is required to separate this macro with other macros or general comment text in order to achieve double spacing between comments.

8. View only screens (noentry, nouupdate) will use the "COMMENT_VIEW_ONLY" macro advising that the screen is view only. It is defined in (\$CARSPATH/macros/custom/comment):

Example: `m4_define('COMMENT_VIEW_ONLY', `This is a view only screen. No data entry is permitted.')

9. The following progscr displays how the macro is used within the attributes section of the file:

```
screen
{
===== GPA/HOURS (STUDENT ACADEMIC) RECORD ===== Record[z0]of[z1] = Sess Year Prog
Sub Class Stat Clear Updated Grd Add Rank Size Reg-Hrs-Audit
2nd Line - Hours: INTEND WAIT ATT EARN PASS QUAL AUDIT QUAL PTS GPA
[a0 ^b0 ^c0 ^d0 ] [e0] [a] [m] [f0 QUAL PTS GPA ] [g ] [g0^h0
^i0 ] [j0 ] [k0 ]
[l0 ^m0 ^n0 ^o0 ^p0 ^q0 ^r0 ^s0 ^t0 ]

[a1 ^b1 ^c1 ^d1 ] [e1] [b] [n] [f1 ] [h] [g1^h1
^i1 ] [j1 ] [k1 ]
[l1 ^m1 ^n1 ^o1 ^p1 ^q1 ^r1 ^s1 ^t1 ]
}
end

attributes
z0:,
nouupdate;
z1:,
nouupdate;
stuac_sess = stuac_sess,
comments = "COMMENT_VIEW_ONLY",
include = ( PREV ),
nouupdate,
scroll = ( a0, a1 ),
upshift;
stuac_yr = stuac_yr,
comments = "Year.",
nouupdate,
scroll = ( b0, b1 );
text1 = prog_text, optional;
stuac_prog = stuac_prog,
comments = "COMMENT_PROG COMMENT_TBL",
lookup text1 joining prog_table.prog_tbcode,
nouupdate,
scroll = ( c0, c1 ),
upshift;
```

Note: This macro is required ONLY for the first comment clause in order to be enabled. This will expand to...


```

F1 execute. CTRL C abort. CTRL F screen forward. CTRL B screen back.
CTRL O 'add' line. CTRL E erase line. CTRL U detail windows. TAB 'insert' mode.
STUDENT DATA ENTRY
ID No.....      13594   SS No..  374-82-4073   Add Date...    09/01/89
Title.....      MS     Ms.           Last Upd...    00/00/00
Name.....  Jackson, Sheri Telephone..  5026515297

===== GPA/HOURS (STUDENT ACADEMIC) RECORD ===== Record 1 of 1 = Sess Year Prog Sub
Class Stat Clear Updated Grd Add Rank Size Reg-Hrs-Audit
2nd Line - Hours: INTEND WAIT ATT EARN PASS QUAL AUDIT QUAL PTS GPA
FA 1989 UNDG FR C 01/13/89 F 0 0
18.0 0.0
0.0 54.0 3.0 0.0 0.0 18.0 18.0 0.0 18.0

This is a view only screen. No data entry is permitted.

```

10. The "COMMENT" macros will generally be defined according to the database field name. The field name will always be capitalized and will be preceded by "COMMENT_".

11. In the attributes section the following screen item appears:

```

Example: d = deceased, optional,
           comments = "COMMENT_DECEASED"
           default = "N",
           nouupdate;

```

12. The comment in the "comment" macro file will be defined as:

```

Example: `m4_define(` COMMENT_DECEASED', `Is the individual deceased?')

```

13. When the "COMMENT" macros are used for screen items that are dependent on table values (lookups are used for validation), the comment macro will be defined according to the database field from which the lookup is being done.

```

Example: text2 = ofc_text, optional,
           nouupdate,
           scroll = ( g0, g1, g2 );
           hold_ofc_add_by = hold_ofc_add_by,
           comments = "COMMENT_OFC_TBCODE COMMENT_TBL",
           lookup text2 joining *ofc_table.ofc_tbcodes,
           scroll = ( f0, f1, f2 ),
           upshift;

```

14. There will be limited instances when the "COMMENT" macros will not be defined according to the field name or the lookup field name. These macros are more generic in nature.

```

Example: `m4_define(` COMMENT_BLANK_ALL', ` , blank for all')

```

User Interface Standards: PERFORM Screens

Introduction

These user interface standards are for PERFORM screen files. Use these standards when creating or modifying this type of file.

General Conventions

The following are general conventions for PERFORM screen files.

1. PERFORM screens which are used solely for "query by form" purposes will have "noentry,noupdate" assigned to each attribute to disable any function other than query.
2. Every PERFORM screen will have a capitalized title which is centered.

Example: `m4_center('DEPARTMENT TABLE', 80)'

3. The title will not be underlined.
4. The naming convention for "tables" will always have the word table last.

Example: CLASSIFICATION TABLE

5. Simple table PERFORM screens will display the code and description in vertical fashion and centered on the screen.

```
INTEREST TABLE
Code..... [I1      ]
Description.. [I2                ]
```

6. A minimum of two periods ".." will separate the screen descriptor and field name.
7. Wherever feasible, fields will be displayed in vertical fashion to facilitate ease of entry and flow of cursor movement.
8. When screens consist of multiple tables, the appropriate name will precede the code for clarity.

```
SESSION TABLE
Session Code. . . . . [f1      ]
Description. . . . . [f2                ]
Calendar Year Order. . . [f4      ]
Academic Year Order. . . [f5      ]
Course Tuition. . . . . [ f      ]

===== SUBSESSION TABLE =====
Subsession Code. . . . . [b1      ]
Description. . . . . [b4                ]
```

9. If more than one record is represented on the screen, a "==" line will separate the records where possible. The only exception is where one record is completely part of another record.
10. This example indicates how multiple records will be separated by "==" lines. The course record (crs_rec), meeting record (mtg_rec), section record (sec_rec), and facility table (facil_table) are represented by this screen.
11. Notice that the facility table outline below does not have "==" line separating it from the meeting record since the entire facility table is contained within the meeting record.

```

** 4: facil_table file**
                                COURSE SCHEDULE
Number.... ----- Title ----- Ind Study Allowed..
Catalog... Dir Study Allowed..
Program... Fac Consent Req...
Dept..... Repeatabl.....
Division.. Min Hrs... 0.00 Tuit Code.. Times Repeatabl.. 0
CIP..... Max Hrs... 0.00 Fee Code... Repeat Hours... 0 0.00
Guideline. Grading..... Bill Code.. Days to Complete. 0
Level. Size. 0 Counting..... Days to Drop.... 0
===== COURSE SECTION =====
Section.... Restriction.. Title....
Session... Requirement.. Faculty.. 0
Year..... 0 Tuition Code. Grading.. Max Reg... 0 Reg... 0
Subsess.... Fee Code.... Method... Max Wait.. 0 Wait.. 0
Bill Code.... Begin.... 00/00/00
Hours... 0.00 Exam Period.. End..... 00/00/00 Days to Complete. 0
===== COURSE MEETING TIME =====
Campus....[ ] Period Code.. Type.. Faculty... 0
Building.....[ ] Begin Time.... 0
Room.....[ ] End Time..... 0 Days... S M T W T F S

```

12. When screens consist of multiple tables and/or records, the title of each section will be centered and capitalized and placed one line below the "=" line. The record information (field names) should begin two lines below the section title so that there is one blank line between the section title and the record information. See "SESSION TABLE" example.
13. An exception to this structure is when there is not adequate space available on the screen to accommodate the entire desired information. If this is the case, the section titles should be centered and placed within the "=" lines. The titles will be preceded and followed by blank spaces. See "COURSE SCHEDULE" example.
14. The attribute section of PERFORM screens will maintain the following structure: Each editing clause will be listed one per line in alphabetical order and indented 5 spaces (the first letter of the edit clause is printed on the 5 position from left).
15. The "lookup" and "joining" clauses will be listed together on one line.

```

attributes
c1 = *crs_no = sec_crs_no = mtg_crs_no,
autonext,
comments = "COMMENT_CRS`''CRS_EG",
required,
upshift;
c2 = crs_cat = sec_cat = mtg_cat,
autonext,
comments = "COMMENT_CAT`''CAT_EG",
default = CAT_DEF,
lookup from *cat_tbcode,
required,
upshift;
c3 = crs_prog,
autonext,
comments = "COMMENT_PROG",
default = PROG_DEF,
lookup from *prog_tbcode,
required,
upshift;
c4 = crs_dept,
autonext,
comments = "COMMENT_DEPT",
lookup c5=dept_div from *dept_tbcode,
upshift;

```

User Interface Standards: Comment Macros

Introduction

These user interface standards are for comment macros. Use these standards when creating or modifying this type of file.

General Conventions

The following are general conventions for comment macros.

1. COMMENT macros will be used in program screens and PERFORM screens, and will be referenced in parameter prompts (PP=) in menuopts. There are two types of comment macros, complete and partial. A complete macro is self-contained. It may contain more than one macro.

Example: ``m4_define('COMMENT_ZIP', 'Enter permanent address zip code.')`
``m4_define('COMMENT_INT', 'Enter interest code. COMMENT_TBL')`

2. You must always use a partial comment macro with another macro or you must always add additional words to it to make the comment complete.

Example: ``m4_define('COMMENT_ID', 'Enter ID# of')`
``m4_define('COMMENT_DATE', 'Format: mm/dd/yy.')`

3. When using more than one macro in a sentence, they must either be separated by a space or by alt quotes "" to allow the macros to expand properly.

Example: `comments="COMMENT_ID counselor. COMMENT_QUERY"`
``m4_define('PP_FS_YR', 'COMMENT_FS_YR`COMMENT_FS_YR_EG')`

4. When defining an example macro (eg: MARITAL_EX), "EX" will be used as a standard to denote "example".

5. Examples will be displayed at the end of a sentence.

Example: PP=Enter the subsidiary balance code, eg: FA90.

6. The following defines the quoting of ``m4_center`'s first parameter. If the parameter does not contain commas, it need not be quoted at all.

Example: ``m4_center(Line containing macro MACRO_NAME, 80)`

7. In this case, MACRO_NAME is expanded before being passed to ``m4_center`'.

Note: The expansion of MACRO_NAME cannot contain commas.

8. If a comma must be used, it must be quoted:

Example: ``m4_center('Line 2, containing macro 'MACRO_NAME, 80)`

9. As long as the macro name is not included within the quotes, the line will be centered properly. If the macro MACRO_NAME itself contains commas, then the ``m4_defn` command must be used.

Example: ``m4_define('MACRO_NAME', 'm1,m2,m3')`m4_center('Line 2, containing macro 'm4_defn('MACRO_NAME'), 80)`

10. The command ``m4_defn` causes M4 to print the definition of the named macro. This usage keeps the commas within MACRO_NAME from interfering with the ``m4_center` macro.

PERFORM Screen Standards

Introduction

Standards for PERFORM screens provide consistency for the user and the modifier of the screen source. By establishing standards for PERFORM screens, both Jenzabar and the user institutions can take advantage of the readability and functionality consistencies. Utilization of standard macros and definitions provides the user of the PERFORM source with a head start in modification and enhancement.

Access

General screens used by different tracks will be located in
\$CARSPATH/modules/common/screens

Screens specific to a module will be located in \$CARSPATH/modules/{module}/screens

Source Code: Documentation Header

The Documentation Header is the definition of what data the screen provides and how you are to use the screen. Use the standard header; once it has been put in the screen, *make* will maintain the log messages.

The following is an example of the documentation header.

```
Standard header before 'database Jenza':
{
  Definition of screen- files used, when used, etc
  Revision Information (maintained by 'make' DON'T CHANGE)
  -----
  $Header: perform,v 8.0 95/04/22 10:23:07 root Developmental $
  $Log:    perform,v $
          Revision 8.0 95/04/22 10:23:07 root
          Release I of CX System
          Revision 7.300 92/05/15 12:30:40 patricia
          SMO#:11301:
          standards
          Revision 7.3 92/04/10 13:46:15 patricia
          SMO#:11301:
          standards
          Revision 7.3 92/02/07 19:31:16 carter
          SMO#:11301:
          standards
          Revision 7.3 91/10/30 14:24:33 jack
          SMO#:11301:
          standards
          Revision 7.2 91/10/25 11:25:18 jack
          SMO#:11301:
          standards
          Revision 7.1 91/10/07 14:35:36 fisher
          SMO#:11301:
          standards
          Revision 7.0 90/07/13 16:40:57 root
          Release G of CX System
          Revision 6.0 88/11/22 09:24:48 anthony
          SMO#:10877G:
          move standards documents into the product
          Revision 6.0 88/11/01 11:30:01 11:30:01 dale (Dale Anglin)
          SMO#:10877G:
          move standards documents into the product
  }
  At the end of the 'instructions' section:
  FRM_DEFLOC($$)
  FRM_DEFREV($$)
  FRM_DISPLAYREV
  end
```

Source Code: Format of screen

The following are standards for the format of PERFORM screens.

- Have a header at the top of the screen whenever possible to define the screen to the user
- Have clearly defined labels for each data field displayed on screen
- Separate files on the screen with a line of "=====" between the files
- Avoid cramming all data fields into 20 lines just to keep the file to one screen while minimizing the number of screens
- Use only the fields needed from the file - if fields are not needed, do not display them. The use of the screen and the end users are key when designing the screen.
- Develop screen for application purpose, not just to display the data file
- Use Of ID Rec As Part Of Screen: If the operator is to be allowed to enter ID records from the screen, all fields from the ID record must be included

Note: If the user is only to have query capabilities on the ID record, the 5 or 2 line standard screen should be used with each field containing the NOUPDATE,NOENTRY attributes.

Make no lines longer than 80 characters within the source file - this includes the attribute definitions.

Source Code: Attributes

The following are standards for attributes of PERFORM screens.

- Shifting: Use upshift for all codes and logical fields
- Comments: Have a comment for each field, describing the field, mention if required, list include values, mention if from table
- Table Lookup: Test entry with table lookup only when necessary. Lookup text for codes, names for id numbers.
- Include: If values are not in a table
- Default: Use standard macro name. Use 'today' for add_date, effective_date, etc. Use defaults whenever possible (saves entry time/keystrokes). Defaults can sometimes be used with NOUPDATE, NOENTRY to save even more keystrokes.
- Required: For fields used in reporting statistics (e.g., Enrollment reports)
- Format: For all type double fields and some money fields
- Picture: For social security. If the client is not using foreign phone numbers, the picture clause can be used with the phone field.
- Autonext: On every field
- Noupdate/Noentry: If fields are not protected by the schema and are only to be used for query and display reasons
- Verify: Requires operator to enter value twice. May be used with Social Security numbers and other critical data fields.
- Right: If value to be entered should be right-justified (e.g., Section numbers). Make sure that if the data for the field is to be entered right-justified, that all other data in that field in the database has also been entered the same way.
- Zerofill: To fill with '000' instead of blanks (e.g., phone numbers)
- Reverse: If field is to be displayed in reverse video

Source Code: Instructions (joins)

The following are standards for joins in PERFORM screens.

- Use composite joins when possible
- If id_rec used, have it listed first in the joins

Compilation

Use *make* processor to compile PERFORM screens. Note the following:

- Executing *make* with a filename of ALL will formbuild all screens where the screen source has a more recent date than the .frm file

- Executing *make* with a specific filename will formbuild only the specified screen source

Note: See *Using the Make Processor in CX Implementation and Maintenance Technical Manual* for more information.

Menu Definition

The Individual who creates the PERFORM screen will create the menu definition file.

Note: Location of menu option file will be
\$CARSPATH/menuopt/<module>/screens/screenname

Make sure that the menuopt file is then called by a menusrc file - edit the appropriate menudesc file and 'make install' the menu master.

The following is an example of a menu option.

```
SD=  
LD=  
DC=DC_PRINT DC_PATH/{module}/{sub_directory}/docname.doc  
PR=RUN_SCREEN  
PP=  
PA=FRM_PATH/module/screenname
```

Testing

The following are the testing standards for PERFORM screens.

Jenzabar (In-house)

1. The individual who creates the screen will be the first to test it in-house - then someone who is familiar with the concept will test it again in-house.
2. Test first from the UNIX shell, then from the menu.
3. Test for aesthetics.
4. Additional individuals will test the screen before it is released for testing at the beta site. A testing report will be completed for the Database Coordinator.
5. Test the following commands on each field; Query, Add, & Update. Compare results with schema permissions and with noupdate/noentry attributes in screen file.
6. Test Master/Detail.
7. Data entry screen should match source document to be used with data entry.
8. Test by using standard CX logins for the typical end user.

Beta site

1. The Account Manager and Coordinator will test the screen at the beta site.
2. Test from the menu.
3. Test using typical end user logins.

Client sites

1. Check off when it is running at the client site - the responsibility of the project manager.
2. If necessary, customize for client to match their source document(s).

Support

The following are the support standards for PERFORM screens.

- Jenzabar will provide all screens with each release/revision
- The Project Manager and Coordinator will be the jointly responsible for all screens
- Jenzabar will compile all screens through 'make' to utilize RCS to check for differences between CX screens and client-modified screens
- Jenzabar will provide menuopt files for each screen and the initial menusrc files
- The client may modify menusrc files any time after the initial installation, reorganizing where the screen(s) is(are) to appear on menus
- Menuopt files should not need modification

- If the client creates a new screen, the client will be responsible for the corresponding menuopt file

Entry Library Screen Standards

Introduction

This section establishes standards for the creation and maintenance of Entry Library (libentry) program screens.

Introduction of Entry Library Features

Entry program features include:

- Simultaneous access of multiple files
- Screens may be designed into a similar format as the input form
- Ability to scroll multi-record files
- Can handle composite key lookups
- Has on-screen table lookup capabilities
- Generates file views at load time (program will recognize new fields upon reloading the program without the need of re-compiling the source code)
- Uses database file permissions to determine if user has read or read-only permissions to the file
- A form or menu of forms may be specified as a parameter to limit (or expand) a users access to entry forms

Differences Between Libentry Screens and PERFORM screens

Screen files consist of the following two sections: screen and attributes. The only difference in the screen definition sections between libentry and PERFORM screen files is that entry screens have only one screen definition per screen form file. Multiple screen forms are achieved through the screen naming convention discussed in the standards section.

Need for autonext attribute

Entry screens do NOT need the autonext attribute. All fields default to autonext.

Field entry order

A major difference is the order in which you enter fields. In PERFORM, the order that the fields appear in the attribute section is the way the cursor moves in add, update or query mode. In the Entry screens, the group order clause defined at the end of the attribute section determines the cursor movement. Fields are entered in the order in which they appear in the grouporder clause. For example:

```
grouporder: group = (field1, field2, field3, field4);
```

Different Lookup Syntax

The lookup feature has a little different syntax. The lookup text name is ALWAYS present and defined BEFORE the field with the lookup attribute.

LIBENTRY lookup syntax is:

```
textscrname = text_name[,optional];
fieldscrname = field_name,
lookup textscrname joining [*]tablename.tablecode;
```

In PERFORM, the syntax is:

```
fieldscrname = field_name,
lookup textscrname = text_name from [*]tablecode;
```

Note: The optional word 'optional' in the libentry example signifies that the textscrname is NOT required to be displayed on the form. In this case the text_name is only displayed in the table lookup feature.

The optional '*' is used the same in both examples. If used, it requires that the value entered is also in the table in order to be valid. An implementation difference is that entry library accepts blanks or zero as a valid value without checking for existence in the table. The 'required' attribute is used on the field if blank or zero is not a valid value.

Multi-field lookup

The following is an example of a multi-field lookup:

```
LIBENTRY multi-field lookup syntax:
textscrname1 = text_name1[,optional];
textscrname2 = text_name2[,optional];
fieldscrname = field_name,
    lookup textscrname1 textscrname2 joining [*]tablename.tablecode;
```

PERFORM multi-field lookup syntax:

```
fieldscrname = field_name,
    lookup textscrname1 = text_name1 from [*]tablecode,
    lookup textscrname2 = text_name2 from [*]tablecode;
```

Composite field lookups

Perform cannot do composite field lookups. An example of a composite field lookup in an entry library program is given below. When a composite field lookup is defined, the composite field **MUST** be used in the grouporder clause. The fields (in this example 'camp', 'bldg', and 'room') should **NOT** appear in the grouporder clause.

```
screen
{
.
.
.
[camp^bldg^room][room_desc      ] [room_ph  ]
}
end

attributes
.
.
.
room_desc = tfacil_text, optional;
room_ph = tfacil_phone, optional;
camp = fac_camp,
    default = "MAIN", upshift,
    comments = "Campus office is located on";
bldg = fac_bldg,
    default = "ADMN", upshift,
    comments = "Building office is located in";
room = fac_room,
    comments = "Room number of office";
office_key: group = (camp, bldg, room),
    lookup room_desc, room_ph joining *facil_table.tfacil_prim;

grouporder: group = (... , office_key, ...);
end
```

Optional attribute

Values may be defaulted into new record fields without the fields being displayed in the screen. This is accomplished using the 'optional' attribute. In screens where the ID record can be added, but the field for deceased does not need to be displayed, the following line could be used:

Example: deceased = deceased, upshift, default="N", optional;

Locations for Forms and Detail Windows

The libentry form screens and specialized detail windows are in the modules area under progscr. For example:

ADMENTRY

\$CARSPATH/modules/admit/progscr/admentry

CSENTRY

\$CARSPATH/modules/develop/progscr/csentry

CTCENTRY

\$CARSPATH/modules/finaid/progscr/ctcentry

IDENTRY

\$CARSPATH/modules/common/progscr/identry

STUENTRY

\$CARSPATH/modules/regist/progscr/stuentry

LIBRARY

\$CARSPATH/modules/Lib/progscr/libentry

These locations are where customized forms and detail windows for the related program should reside. If the program cannot locate the desired form in its corresponding progscr directory then the LIBRARY's directory will be searched. The LIBRARY's directory holds all form and detail window definitions that could be used by any of the entry programs. Some files in this directory are church_1, sch_1, accomp, ctc, and dontot.

File Naming Conventions

Forms should have an underscore number appended to the base name. For example, if the base name is 'longapp' the file names should be longapp_1, longapp_2, longapp_3, etc.. If only one screen is needed for the form the '_1' should still be appended.

Detail windows (scroll screens) always consist of one and only one screen. The detail window form name should NOT be appended with the '_1' convention.

Screen Field Naming Conventions:

Don't use a series of numbers for screen field names. The group order clause is easier to change when more descriptive screen names are used.

Exception to above, detail window field names may be a series since the grouporder clause does not use these field names directly.

Screen File: The Attribute Section:

The following conventions should be used in the attribute section:

Note: These standards can apply to PERFORM screens.

Typical attributes that are used are:

- scroll
- noentry (do not use except for "match" fields)
- nouupdate
- blank
- reverse (use sparingly, someday defaulted fields)
- optional (will be highlighted with reverse attribute)
- required
- default start on a new line indent 4 spaces
- include
- upshift
- lookup/joining start on a new line indent 4 spaces
- commentsstart on a new line indent 4 spaces

They should be used in this order. If the scroll attribute is being used and specifying many scroll fields such that there would not be enough space on that line to also specify 'noentry', 'optional', or 'required' as needed; then the scroll attribute should be placed on the next line indented 4 spaces.

Note: A comment line should be included on all fields, with the exception of fields that are the result of a lookup. Lookup display fields do not need a 'comments' line. If a 'comments' line will not fit on one line then the format should be:

Example: comments=

"Enter the Individuals Name with the following format 'Last, First M., Suffix'"

Tips for Creating Entry Screens

Note the following tips:

- A screen field name should not begin with *end*. When the field is defined in the attributes section, it causes a premature end to the attributes section.
- Lookup fields should appear in the attribute section before they are used in a lookup clause. There is a good probability that lookup fields may not be displayed on the screen; so lookup fields should have the optional attribute specified and the attribute line defining the occurrence of this optional field needs to occur before the lookup clause references this field.
- When lookups are being done on a form, it does not matter whether table lookup fields are listed in the group order clause. However in detail windows the scroll group clause must contain all fields (including table lookup fields) that are being scrolled.

ACE Report Standards

Introduction

Standards for ACE reports allow consistent reading, output, & execution ACE report files. By establishing standards for ACE reports, both Jenzabar and the client can take advantage of the readability and functionality consistencies. Utilization of standard macros, definitions, and sections provide the user of the ACE source a head start in modification and enhancement.

Definitions

Note the following definitions:

- Report: A selection of data, usually from multiple files, in an organized format designed for management data needs (e.g., Balance Sheet in the fiscal area).
- Roster: A selection of data records in a specified sorted order. There will usually be one or two lines per record (e.g., Personnel Directory).
- Form: A standard output with very little variation between printed output copies where there will be one data record per form. i.e. Registration form.

Access

ACE Reports are located under the 'reports' function directories under each module (eg \$CARSPATH/modules/admit/reports).

General reports used by different tracks will be located in \$CARSPATH/common/reports.

Source Code: Documentation Header

The Documentation Header is the definition of what data is provided by the report and how the report is to be used. Use the standard header; once it has been put in the report, *make* will maintain the log messages.

The following is an example of the documentation header.

```
{
Revision Information (Automatically maintained by 'make' - DON'T CHANGE)
-----
$header$
$$
}
```

Note: If the report has changed names or source location, make sure that the previous name/location is either mentioned in the log message or a comment entered to that effect.

Source Code: Defined Variables, Parameters, and Functions

The Define macro includes all CX ACE functions - It must be included in each ACE report define statement. The following functions are expanded in the **REP_DEFINE** macro.

```
REP_DEFINE
_getcars          - to call environmental variables
_midstring        - center a piece of text
_first_name       - extract first name from id_rec name field
_full_name        - extract name (First M Last) from id_rec name field
_last_name        - extract last name from id_rec name field
_dashdays        - use with array of days of the week
_justify          - justifies 3 parts of line (left,center,right)
_formatrcs        - formats RCS header, source lines
_toupper          - changes lower case to upper
variable text     type char          length xxx
  where 'xxx' is the width of the page
```

Note: Other macros exist for ACE reports used with LPS and WP

Source Code: Output definition

Output definition does not direct the output of the source; the script calling the report performs this function. Use a macro for output definition.

Note: A macro exists for standard reports and for "side-ways" (110) reports. Use the REP_FORMAT(132) for wide reports. Other macros exist for LPS and WP

The following are other macros for LPS and WP.

REP_OUTPUT

top margin 3
bottom margin 3
right margin 80
left margin 0
page length 66

REP_OUTPUT_SIDEWAYS

top margin 3
bottom margin 3
right margin 110
left margin 0
page length 51

Source Code: Read Statements (Including JOIN, WHERE clauses)

The following are standards for read statements.

- Use multiple read statements if reading fields from several files.
- Avoid sequential reads - Caused by using unindexed fields in the where clause, "<>" conditions in the where clause, or use of "or" in the where clause.
- Avoid reading the total record - only read the fields needed in sorting and formatting.
- The individual creating the report must have a thorough understanding of the data structures involved in the report (joins with files, etc).
- List each data element read on a separate line - this will aid in the 'make' processor comparing source files.
- List each where clause section on a separate line.
- List each join clause section on a separate line.
- If an assign statement is used, make SURE each relation (temporary file) created has a primary (or unique) set of tuples in it (probably include primary key in fields read).

Source Code: Sort Clauses

If sorting first by name, sort next by id_no. This sort those records with the same name.

Source Code: Report Format

The following are standards for report format. The following is an example report format.

```
- this includes the CARSUNIT, CARSNAME, CARSADDR
  on three lines
REP_DATE - Date, page number
REP_TIME - time
REP_HEADER_NEW(title)
- CARSUNIT
- Date, CARSNAME, Page
- Time, CARSADDR, Report Name
- Title
```

Header Macro: REP_HEADER

- Begin header with REP_FORMAT_WIDE if the report is 132 columns instead of 80
- The report macros can be locally customized if desired.

- The main title line will be in all upper case letters. If it is not passed to the header macro, use the midstring function to center it on the page.
- All subtitles will have the first letter of each word in upper case, the rest in lower case. Make sure that the **REP_JUSTIFY** macro or the midstring function is used to center the subtitles.
- All values passed as parameters will be printed in the header section of the report as subtitles.
- Do not clip values in the header unless they are part of a character variable that will be centered.
- Try to stay away from first page headers - use the generic "page header" as much as possible.

Body

- If printing an individual's name, print the id number in the first column and sort by name then id_no
- If numbering the individuals on the page in the left column, print the id_no in the columns directly after the name.
- There will be macros available to condense conditions frequently used, i.e. printing name and address, testing for second line of the address.
- Present a logical flow of clauses, for example:

```
before group of
on every record
after group of
on last record
```

- Test for (and avoid) division by 0 if there are such calculations involved.
- Print out error conditions when it is known that they exist. With "if, then" tests, use "else" to determine what will be done if the condition is not met.
- When using multiple before and after group of clauses, list the before group of clauses in the order in which the fields are sorted and the after group of clauses in reverse sort order.
- Use "skip x lines" instead of multiple "print" statements for blank lines within the report.
- There is a "need" statement that can be used for line counts to insure that page breaks occur at logical places.
- Rosters will be columnar formats.
- Summary reports will be in matrix formats.
- Numeric fields will have defined lengths, formats with "using" statements. This is particularly the case with years and id_no. For example:

```
a. Id number : 'using "#####"'
b. Year: 'using "####"'
c. Voucher number: "#####"
```

- Structure of the body for indentation will follow similar standards as for C program code. For example:

```

col l: format
Ctrl T: clause header (before group of, etc)
Tab: lines under clause header
Tab, Ctrl T: continuation of Tab lines
ex:
format
  before group of
  let x=0
  if ( ) then
    begin
    end
  else if ( ) then
    begin
    end
  let
on every record
  if ( ) then
    begin
    end
  else
end

```

Footer

- Page breaks (line counts) should occur at the end of a record, not in the middle.
- Have page totals in accounting reports

Last Page

- Include REP macros for printing location. **REP_LAST_REC** should appear in the **on last record** clause and the **REP_TRAILER** macro should appear in the **page trailer** clause.
- Include grand counts and column totals
- Use "-----" for group totals
- Use "======" for grand totals
- Don't use subtotals if count = 1

Compilation (Translation)

Use *make* processor to compile ACE reports. See *Using the Make Processor* in *CX Implementation and Maintenance Technical Manual* for more information. Test the report using the *acego* filename to ensure that the report is not reading data sequentially. The menu calls a script that will call the report and run the report using *acego -q*.

Note: Most commonly used script will be "runreports". This requires the formtype, filename, parameters, and the output device or file

ACE reports for LPS or Labels will be executed through either *runletters* or *runlabels* that will call all the necessary processes and direct the output to the correct location.

Those ACE reports that are compiled at execution time because they require user input of the selection or sort criteria will be executed using "runsort". The source for these reports will be kept in *\$CARSPATH/modules/<module>/commands/* as *filename.ace*

Executing ACE Reports

When executing, an ACE report should not read data sequentially if indexed fields can be used. Remember the following about the report's output:

- A report does not just consist of data from the files, in printed format (except when printing tables).
- Only at the client's request and expense are "dump" reports to be written
- Use one ACE report if both a summarized output and detailed output are desired where both reports require output meeting the same criteria

Menu Definition

The individual who writes the ACE report is responsible for creating the menu option file.

Note: The menuopt file will be located in
\$CARSPATH/menuopt/<module>/reports/reportname

The following is an example of an menu option.

```
SD=  
LD=  
DC=DC_PRINT DC_PATH/<module>/<subdir>/file.doc  
PR=RUN_REPORTS (or RUN_SORTS, etc)  
PP=  
PA=-f  
PP=  
PA=FT_STANDARD  
PP=  
PA=ARC_PATH/<module>/filename  
PP= {for all parameters required} Be sure to use macros in  
PA= {for all parameters required} user/mnu  
PP=PP_OUTPUT  
PA=PA_OUTPUT  
PW= (if restricting use of the report)  
SP (if it is to be a scheduled process)
```

Testing

The following are the testing standards for ACE reports.

Jenzabar (In-house)

1. Use Quality Control Verification forms in testing; give the completed form to the Database Coordinator when testing has been completed.
2. The individual who writes the report will be the first to test it in-house, followed by another person who knows something about the subject matter.
3. Test first from the shell. Save output.
4. Test next from the shell using the appropriate script. Save output.
5. Test then from the menu as self. Save output.
6. Test from menu as end user. Save output.
7. Check output for aesthetics.

Beta Site Testing

1. The Account Manager and Jenzabar Coordinator will test the report at the beta site.
2. Run the 'make' processor on the report - create the local version and make using client defined macros.
3. Verify counts and totals with INFORMER queries.
4. Test from the menu as an end user.

Client sites

Check off when the report has had the local version created and tested at the client site. This is the joint responsibility of the Jenzabar Coordinator and the Account Manager.

Support

The following are the support standards for ACE reports.

- All standard menuopt files for reports will be provided with each release/revision of the system and will initially be the responsibility of Jenzabar
- Anything located in the menuopt directories with the same name as a CX menuopt file may be overwritten in the next release
- All menudesc files for reports will be the responsibility of the client

Menu Option Standards

Introduction

This section describes the standards for menu option files for CX. The standards are to be used when creating or modifying any menuopt file.

Menu Option Tags

The following describes all the valid tags which can be used within a menuopt. Some of the tags are optional and will not be found in each menuopt. When the symbol "#" follows a tag in this document, it denotes an integer value. All tags are upper case and the menu program expects the tags to be used as described in this document. All tags are position independent, though this document describes their use in a logical order.

SD

Short Description. The default value defines the short description for the menuopt which appears on the menu. The SD tag is mandatory and only one may be defined per menuopt. All significant words should have the first letter of the word capitalized to adhere to Jenzabar standards.

Example: SD: optional,

```
default = "Move Graduates to Alumni";
```

SP

Schedule Process. The default value of the SP tag contains three comma-separated values which act as defaults for the schedule process window. The first value denotes the default time, the second denotes the default day, and the third value of "Y" or "N" denotes the default background process prompt. As illustrated in the following examples, the keyword "schedtime" can be used and menu will use the value of "schedtime" as defined in \$CARSPATH/system/etc/menuparam.s. If "schedtime" is not defined in this file, MENU will default to "1100P".

Example: SP: optional,

```
default = "schedtime,,Y";
```

SP: optional,

```
default = "7:00P,sunday,Y";
```

SP: optional,

```
default = "NOW,,N";
```

OUTPUT

Output. The default value of the OUTPUT tag contains two comma-separated values which act as defaults for the schedule process window. The first value denotes the default output mode which is either "file", "more", or a valid printer. The second denotes the default file which is only valid if the mode is "file".

Example: OUTPUT: optional,

```
default = "${CARSPRINTER},";
```

OUTPUT: optional,

```
default = "file,outputfile",
```

WARN

Warning. The WARN default text will be displayed in a dialog box prior to the schedule process window. The WARN default cannot exceed 74 characters. As illustrated in one

example, WARN can be used to notify the user that the process output requires wide paper by setting the default to the WARN_WIDE_OUTPUT macro.

Example: WARN: optional,
 default = "Execute the 'Update Stats' option first.";
WARN: optional,
 default = "WARN_WIDE_OUTPUT";

RD#

Run Description. The default values define the run description for the menuopt. The default values will appear in numerical order, defined by the appended integer. The appended integer must be unique within the set of RD tags. The RD tags are optional. Within menu, the help command (Cntrl-W) will display the run description in a box.

Example: RD1:, optional,
 default = "This option updates all the student records.";

PR

Process Run. The default defines which process to execute. It is typically either an application program or a script. The PR tag is mandatory and only one may be defined per menuopt.

Example: PR: optional,
 default = "RUN_PROG_INFORMER";

LU#

Look Up. The LU tag is used for table lookup. It defines the table and column which is being looked up. The columns are displayed within the standard table lookup box. All LU# tags must have an "optional" attribute associated with it and an associated PA# tag that contains a lookup joining clause.

Example: LU5 = table.lookup_column, optional;
PA5 : optional,
...
lookup LU5 joining *table.column;

PA#

Prompt Answer. The default defines a possible value to pass to the process being executed. Comments are displayed on line 24 of the terminal. The arguments are passed to the process in the order defined by the integer appended to the PA tag. Macros should be used for defaults, comments, and includes whenever feasible. The length, comments, and optional attributes are mandatory for each PA tag. All the valid attributes for the PA tag are described in the following section.

Example: PA7: optional,
 upshift,
 length 4,
 lookup LU7 joining *table.column,
 comments="COMMENT_TABLE_COLUMN. COMMENT_TBL",
 default="COLUMN_DEF";

GET_ORDER

Get Order. The GET_ORDER group specifies the order the cursor will follow through the fields on the menuopt screen. By not specifying GET_ORDER, the cursor flow will flow in a logical order. This tag is optional and is used very seldom.

Example: GET_ORDER: group=(PA3,PA4,PA5,PA7), autonext;

Menu Option Attributes

The following reviews all the available attributes used to describe the characteristics of a tag with the menuopt screen.

optional

The optional attribute MUST be included in all tags for the menuopt screen.

length

A length MUST be specified on all PA# tags. If the tag is related to a database column, the length must equal the column length defined in the schema. The length value can be equated to a macro for the financial module columns where the defined length in the schema is a macro.

Example: length GL_CNTR_LEN,
length 40,

upshift

The upshift attribute forces all input to upper case.

dwshift

The dwshift attribute forces all input to lower case.

type

The type attribute should only be used for numerical columns (i.e., integer, double).

comments

The comments attribute MUST be defined for any PA# tag appearing on the menuopt screen. It should be equated to a COMMENT macro if possible. If a table lookup, blank for all, or use of wildcards are valid, then the macros COMMENT_TBL, COMMENT_BLANK_ALL, COMMENT_WILDCARD MUST be specified respectively.

Note: The COMMENT_TBL macro should appear in any comment which the PA# tag contains a table lookup or an include attribute.

Example: comments="COMMENT_ACAD_PROG, COMMENT_BLANK_ALL,
COMMENT_TBL",

lookup

Table lookups should be used when a valid table exists. Lookups can be forced by preceding the table name with an asterisk, though this will still allow a blank value if not used in conjunction with the "required" attribute.

Example: lookup LU# joining *major_table.major,

default

A default should be specified for all appropriate tags. The default should be equated to a default macro whenever possible.

Example: default="DEF_MAJ",

include

The include attribute must be specified for tags where a lookup attribute is not specified. An include macro should be used whenever possible. Integer ranges are valid as seen in the following examples.

Example: ACAD_YR_INCL,

```
include=(1:10000),
include=(MGRD,FGRD),
```

required

The required attribute forces the user to enter a non-blank value.

Example: required,

Menu Option Standards

The following describes the basic menuopt standards Jenzabar used when developing a menuopt. The following standards were used as a guideline.

1. Database names of files and records (e.g., stu_acad_rec) will not appear in SD's, RD's window. Instead, the complete file name will be used.

Example: student academic record

2. The letter production system (LPS) and the forms production system (FPS) will only be referred to in the run description.
3. Normal upper/lower case conventions will be used in the runtime (RD) description text. If information is to be emphasized, a "NOTE: " will be used. Included in the note should be information relating to steps that must be completed prior to the current one.

Example: RD1:, optional,

```
default = "This is the final step in generating student  ";
```

RD2:, optional,

```
default = "data sheets (SDS).  ";
```

RD3:, optional,

```
default = "  ";
```

RD4:, optional,

```
default = "NOTE: Use formtype "sds" for SDS without billing  ";
```

RD5:, optional,

```
default = " information.  ";
```

4. The menupw.s file in \$CARSPATH/system/etc will be used for password protection in menusrc and menuopts. For example, to enable the password in a menuopt file:

Example: PW: optional, default="@REGIST";

5. The corresponding entry in the \$CARSPATH/system/etc/menupw.s file would be:

Example: REGIST:<password>:

6. A six character alpha descriptor (e.g., REGIST) will be used with the "@" to denote the unique item in the menupw.s file. This will replace the numbers used previously.
7. Comment lines or run descriptions will not contain the uses of "eg:", "ex:", "ie:" or "etc." except in special cases. An example of a special case would be when illustrating how to enter and use wildcards.
8. The COMMENT_TBL macro will be used within every comment line where space allows and either a table lookup is performed or an include statement is indicated. Leave 1 space between the ending of the comment and the COMMENT_TBL macro since the macro begins with a blank space.

Example: PA4: optional,

```
...
comments="COMMENT_CLASS_YR COMMENT_TBL";
```

9. All prompts and prompt macros **MUST** be presented in column format starting in column 20 when space will allow. All prompt macros **MUST** be 31 characters in length, padded with trailing periods with a minimum of 2 periods following text.
10. The Runtime (RD) description window will not contain a header or title, the text will be in paragraph form. The run description is specified in the screen by using "RD#" (where the # represents an integer) attributes in numerical order.
11. No special capitalization will be used in the runtime (RD) description window. Record and file names will not be capitalized.

Example: RD1:, optional,

```
default = "This is the final step in generating student ";
```

RD2:, optional,

```
default = "data sheets (SDS). ";
```

12. If another menu option is to be referenced in the body of RD text, the `m4_getoptdesc' macro will be used to bring in the short description (SD) for that option.

Example: RD1:, optional,

```
default = "This is the second step in generating class lists. ";
```

RD2:, optional,

```
default = "Use `m4_getoptdesc(utilities/programs/fps.reglst) ";
```

RD3:, optional,

```
default = "option to print the list. ";
```

The `m4_getoptdesc' macro above will expand to "Print Classlists/Waitlists".

Note: Since the text returned by m4_getoptdesc is of variable length, this may cause alignment problems within the run description since all lines within the box are centered.

13. All menuopts screens will contain a title, which will be the same text as the short description. The menuopt screen title **MUST** be capitalized. The `m4_center' macro command should be used and it should start at column 20 to align with the menuopt prompts.

Example: screen

```
{
`m4_center(`MOVE GRADUATES TO ALUMNI',40)'
}
```

14. All menuopts screen prompts should start in column 15. Whenever possible a macro should be used for the prompt.

Example: screen

```
{
`m4_center(DETAIL,40)'
PP_OUTPUT[PA5]
PP_SORT_FIELD[PA7]
PP_PERSON[PA9]
```

PP_FS_RAN[PA11]-[PA13]

PP_FS_YR[PA16]

}

15. The attributes section of the menuopt should be in standard format. All attributes should appear in alphabetical order, one per line with blank lines between tags, and the optional attribute should appear on the tag line. Use the "stdscr" script to automatically standardize the attribute section of the screen.
16. The attributes LD (long description) and DC (documents) are not valid in the screen menuopts.

Related Scripts and Menu Option Testing

This describes two scripts which aid in the development of the menuopt run descriptions. The two scripts are "txt2rd" and "rd2txt" and they can be called from within the VI editor to assist with developing the RD# tags. This section also explains how to test the menuopt to make sure it is functioning properly before installing it and affecting end users.

txt2rd script

The txt2rd script will transform a block of text into the format required for a run description. For example, within the VI editor if the following paragraph was entered:

This option will move create pledge reminders for all alumni with outstanding pledges within the class year range specified.

By simply marking the first line as "a" and the last line as "b" by using the VI mark (m) command, then executing the VI command ":a,b ! txt2rd", the above paragraph will be substituted with the following:

```
RD1:, optional,
  default = "This option will move create pledge reminders  ";
RD2:, optional,
  default = "for all alumni with outstanding pledges within  ";
RD3:, optional,
  default = "the class year range specified.          " ;
```

rd2txt script

The rd2txt script will transform a block of RD# tags into a block of text. For example, within the VI editor if the following RD# lines existed in a menuopt:

```
RD1:, optional,
  default = "This option will move create pledge reminders  ";
RD2:, optional,
  default = "for all alumni with outstanding pledges within  ";
RD3:, optional,
  default = "the class year range specified.          " ;
```

By simply marking the first line as "a" and the last line as "b" by using the VI mark (m) command, then executing the VI command ":a,b ! rd2txt", the above section will be substituted with the following:

Note: This option will move create pledge reminders for all alumni with outstanding pledges within the class year range specified.

Testing a Menuopt

To test a menuopt prior to installing it, use the "-o" option to MENU. All menuopts are now program screens which are translated into a binary file. After translating a menuopt file with the "make F=file" command, you can test the menuopt by passing the full path of the menuopt binary file to the "-o" option of MENU. For example:

```
% cd $CARSPATH/menuopt/regist/others
% make F=clone
% menu -o ./clone.opt
```

Note: The ".opt" is required because the ".opt" denotes the translated binary file for the menuopt. The "." precedes the filename because a full path is required. The installed version of a menuopt could be tested as follows:

```
% menu -o $OPTPATH/regist/others/clone.opt
```

Menu Option Examples

The following are three menuopt examples. These examples illustrate how to properly use the menuopt tags previously described and the associated tag attributes. The first example is a financial report, the second example is a menuopt calling the application program TRANS, and the third example is a typical perform screen menuopt.

Menu Example 1


```

{
Revision Information (Automatically maintained by 'make' - DON'T CHANGE)
-----
$Header: menuopts,v 8.0 95/04/22 10:23:51 root Developmental $
-----
}

screen
{
`m4_center(DETAIL,40)'

PP_OUTPUT[PA5]
PP_SORT_FIELD[PA7]
PP_PERSON[PA9]
PP_FS_RAN[PA11]-[PA13]
PP_FS_YR[PA16]
PP_FUND_RAN[PA17]-[PA18]
PP_FUNC_RAN[PA19]-[PA20]
PP_OBJ_RAN[PA21]-[PA22]
`m4_keepif(GL_SUBFUND_ENABLE,`Y')'
PP_SUBFUND_RAN[PA23]-[PA24]
`m4_keepend'
PP_NONDSPL_OBJ[PA25]
PP_SUBTOTAL[PA26]
}
end

attributes

SD: optional,
    default = "Detail";

SP: optional,
    default = "schedtime, ,N";

OUTPUT: optional,
    default = "${CARSPRINTER},";

WARN: optional,
    default = "WARN_WIDE_OUTPUT";

RD1:, optional,
    default = "Page breaks on Object with a detail line for each Program.";

RD2:, optional,
    default = "Report can be sorted by responsible person. ";

PR: optional,
    default = "RUN_REPORTS";

PA1: optional,
    default = "-f";

PA2: optional,
    default = "FT_WIDE";

PA3: optional,
    default = "OTH_PATH/accounting/acctctl.oth";

PA4: optional,
    default = "-DREP_OBJ_TYPE";

PA5 : optional,
    upshift,
    length 10,
    OBJ_TYPE_INCL(OBJ),
    default = "OBJ_TYPE_DEF(OBJ)",
    comments="COMMENT_OUTPUT_COL. COMMENT_TBL";

PA6: optional,
    default = "-DSORTFIELD";

PA7 : optional,
    dwshift,
    length 4,
    include=(prim,sec," "),
    comments="COMMENT_SORT_FLD";

PA8: optional,
    default = "-DREP_WANT_NORESP";

```

```

PA9: optional,
    comments = "COMMENT_RESP_PER",
    default = "Y",
    LOGICAL_INCL,
    length = 1,
    upshift;

PA10: optional,
    default = "-DREP_FISCAL_PRDBEG";

PA11: optional,
    comments = "COMMENT_FS_CODE_BEG. COMMENT_TBL",
    default = "FS_CODE_DEF",
    FS_CODE_INCL,
    length = 4,
    required,
    upshift;

PA12: optional,
    default = "-DREP_FISCAL_PRD";

PA13: optional,
    comments = "COMMENT_FS_CODE_END. COMMENT_TBL",
    default = "FS_CODE_DEF",
    FS_CODE_INCL,
    length = 4,
    required,
    upshift;

PA14: optional,
    default = "-DREP_ATYPE";

PA15: optional,
    default = "BGT";

PA16: optional,
    comments = "COMMENT_FS_YR. COMMENT_TBL",
    default = "FS_YR_CUR",
    include = ( FS_YR_VALID ),
    length = 4,
    type integer;

LU17 = fund_table.txt, optional;

PA17: optional,
    comments = "COMMENT_FUND_BEG. COMMENT_TBL",
    default = "BEG_FUND",
    length = GL_FUND_LEN,
    lookup LU17 joining *fund_table.fund;

LU18 = fund_table.txt, optional;

PA18: optional,
    comments = "COMMENT_FUND_END. COMMENT_TBL",
    default = "END_FUND",
    length = GL_FUND_LEN,
    lookup LU18 joining *fund_table.fund;

LU19 = func_table.txt, optional;

PA19: optional,
    upshift,
    comments="COMMENT_CNTR_BEG. COMMENT_TBL",
    default = "BEG_FUNC",
    lookup LU19 joining *func_table.func,
    length GL_FUNC_LEN;

LU20 = func_table.txt, optional;

PA20: optional,
    upshift,
    comments="COMMENT_CNTR_END. COMMENT_TBL",
    default = "END_FUNC",
    lookup LU20 joining *func_table.func,
    length GL_FUNC_LEN;

LU21 = obj_table.txt, optional;

PA21: optional,
    upshift,
    comments="COMMENT_ACCT_BEG. COMMENT_TBL",
    default = "BEG_OBJ",
    lookup LU21 joining *obj_table.obj,
    length GL_OBJ_LEN;

```

```

LU22 = obj_table.txt, optional;
||
PA22: optional,
  upshift,
  comments="COMMENT_ACCT_END. COMMENT_TBL",
  default = "END_OBJ",
  lookup LU22 joining *obj_table.obj,
  length GL_OBJ_LEN;

LU23 = subfund_table.txt, optional;

PA23: optional,
  upshift,
  comments="COMMENT_PROJ_BEG. COMMENT_TBL",
  default = "BEG_SUBFUND",
  lookup LU23 joining *subfund_table.subfund,
  length GL_SUBFUND_LEN;

LU24 = subfund_table.txt, optional;

PA24: optional,
  upshift,
  comments="COMMENT_PROJ_END_BLANK",
  default = "END_SUBFUND",
  lookup LU24 joining *subfund_table.subfund,
  length GL_SUBFUND_LEN;

PA25: optional,
  default = "Y",
  include = (Y,N,O),
  comments="COMMENT_NONDSPL_OBJ",
  length = 1,
  upshift;

PA26: optional,
  comments = "COMMENT_SUBT_SCHGRP",
  default = "N",
  include = (S,G,N),
  length = 1,
  upshift;

end

```

Menu Example 2

```

{
Revision Information (Automatically maintained by 'make' - DON'T CHANGE)
-----
$Header: menuopts,v 8.0 95/04/22 10:23:51 root Developmental $
-----
}
screen
{
    `m4_center_clipped(CREATE/EDIT TRANSCRIPTS,40)'
    `m4_center_clipped(PP_NO_PARMS, 40)'
}
end

attributes
SD: optional,
    default = "Create/Edit Transcripts";

RD1: optional,
    default = " Functions: ";

RD2: optional,
    default = "";

RD3: optional,
    default = " Print Unofficial Transcript";

RD4: optional,
    default = " Create a Transcript ";

RD5: optional,
    default = " Edit a Transcript ";

RD6: optional,
    default = " Update Student Statistics ";

PR: optional,
    default = "BIN_PATH/trans";

PA1: optional,
    default = "-f";

PA2: optional,
    default = "MORE";

PA3: optional,
    default = "-L";

PA4: optional,
    default = "${CARSSITE}";

PA5: optional,
    default = "-e";

end

```

Menu Example 3

```

{
Revision Information (Automatically maintained by 'make' - DON'T CHANGE)
-----
$Header: menuopts,v 8.0 95/04/22 10:23:51 root Developmental $
-----
}
screen
{
    `m4_center_clipped(CAMPAIGN/APPEAL/DESG,40)'
    `m4_center_clipped(PP_NO_PARMS, 40)'
}
end
attributes
SD: optional,
    default = "Campaign/Appeal/Desg";
RD1: optional,
    default = "All campaign records, any appeals associated with the campaign,";
RD2: optional,
    default = "and all designation records are available through this screen. ";
PR: optional,
    default = "RUN_SCREEN";
PA1: optional,
    default = "FRM_PATH/develop/campaign";
end

```

Programming Style, Standards, and Conventions

Introduction

The purpose of the Programming Style, Standards, and Conventions is to set several standards and conventions for use when developing software to improve subsequent maintenance and enhancements. This set of standards should be appropriate for any project using C.

Note: While these standards are not all inclusive, when unusual situations arise, you should consult experienced C programmers or code written by them which follows these rules.

This section was initially created for internal Jenzabar use and assumes that the reader has some knowledge of programming and the C language.

Design Guidelines

The optimal approach to software design and development at Jenzabar requires that the developer work with resources from product services to define the requirements and specifications for the development project. Whenever possible, the project requirements are provided by an individual (usually the product manager) who has in-depth understanding of the client environment in the product area undergoing modification.

The programmer/analyst or project leader will then work from the requirements to produce program specifications. The project leader and product manager should work closely together throughout the project to ensure that requirements are met and to resolve together any unforeseen problems and issues.

The most important aspect of program design is to get ongoing feedback from knowledgeable persons throughout the process.

Program Design

An essential part of design for most larger projects is the creation or modification of the entity-relationship diagram (ERD). These will be maintained on the PC network under the users/dev directory. In this directory, there are subdirectories for each module area. The ERD's should be stored under the appropriate area.

- A design directory should be created under the program-specific source directory on the UNIX system. For example, design notes and specifications for the "grading" program should be stored under \$CARSPATH/src/regist/grading/DESIGN. This design directory contains program and function specifications deemed to be necessary and helpful by the project leader/analyst.
- A testing directory should be created under the program specific source directory on the UNIX system as well. This directory will contain program-specific testing scenarios. Whole modules testing (setup scripts and scenarios) will be stored and maintained under RCS in a <modulename>/TESTING subdirectory.

Each program testing directory will contain, at a minimum a testing scenario describing:

- Where the program is located on the menu
- How to load the program
- How to exit the program

The testing directory should contain instructions or scripts to initialize the program environment so a new user will have permissions to execute the tests described in the test scenarios.

Use of Standard CX Functions

Many of the functions that are normally common across applications are already defined in a standard CX library. If you need a particular function but can't find it in the standard libraries, ask others if such a function already exists. Should you find yourself in a position of having to code a

function you feel might be widely useful, talk to someone in the systems/utilities group about the feasibility of having this new function added to an existing library.

Also, don't underestimate the usefulness of reviewing existing code for use of standard functions, such as: `param_parse`, `esql_server`, `esql_init`, `prog_init`, `scr_init`, etc. Most application programs are uniform in their inclusion of certain standard functions.

In addition to using functions already defined in `$CARSPATH/Cislib` or `$CARSPATH/Lib`, do use application libraries wherever feasible for functions that may be needed across programs but within modules areas.

Use of Transaction Processing

Any program that updates several records that are interdependent must use transaction processing to ensure data integrity. For example, if a process must create a master and detailed set of records that describe a transaction, the program must be designed so that these records either all get successfully written, or none are written.

A transaction should be as small as possible while still maintaining the integrity of the related records. A transaction should never include any user input or request for user input within it; this ensures that the length of the transaction will be determined only by machine time to execute the transaction.

Audit on Summary Fields

Any module that contains summary fields in the database should have an audit program that validates the values stored in those summary fields. The audit program should review the detail records that support the summary fields, identify any discrepancies, and optionally correct in the database any incorrect summary values.

ESQL Guidelines

Any file that uses ESQL must have a ".ec" extension. Such files are preprocessed before going to the C compiler. Some things to keep in mind when using ESQL are:

- Any program that uses ESQL statements must call `esql_init` in its initialization sequence; this ensures, that money types will be properly handled.
- Any program that uses `libscr` should call `esql_server` function on all files to initialize the `dbview` structures needed by the screen package.
- Always use `sqldec.m4` macros for specific field name lists; don't do a `select * from table`.

Program Arguments

Program arguments are passed using a dash followed by a letter, followed optionally by a value. If no value is given, the option acts as a turn on/off feature in the program. The library function, `param_parse` should always be used to parse program arguments and to store the argument values in specified variables. If `param_parse` should fail, `param_usage` should always be called to provide a usage message to standard output.

Note: Jenzabar recommends that, whenever possible, the use of parameters should be independent of one another. Try to avoid a situation where if one parameter (that is optional) is specified, another parameter (also optional) must be specified. The usage message is normally of no help for the user here, and such usage can be frustrating and confusing.

General Guidelines for Program Arguments

There are several program arguments whose letter identifier is standard across applications. The argument are as follows:

- `-d` display only (logical)

- -d [date]
- -i [ID or ID string]
- -L [site]
- -p program
- -s [session]
- -T [tick code]
- -u [update]
- -y [year]

Program Arguments for Entry Programs

Entry programs use the same basic set of program arguments. This set of arguments can be found in the `def.c` of any entry program. They include:

- -o [office added by]
- -m [default menu screen]
- -f [name of desired form to invoke, instead of menu]
- -t [today's date, or effective date]
- -P [path for screens]
- -a auto mode (logical)
- -F force query (logical)
- -M [menu title]
- -q allows additional query restrictions (logical)
- -D [debug level]
- -S [pause level]

Naming Conventions

It is important to be consistent in naming programs, files, functions, and variables to increase understand-ability and ease of maintenance.

Program Names

Program names should be 10 characters or less in length. Data entry type programs that do NOT use `libentry` should end in "ent" (such as `crsent`, `regent`, etc). Data entry programs that do use `libentry` should end in "entry" (such as `stuentry`, `cseentry`, `identry`, etc). The name you give a program will be around a long time, so select something that is intuitive and simple.

File Name Length and Suffixes

The source files in the system, in general, will be 10 characters before any "dot" extension (or suffix) is added. This is to allow our code to exist on systems that have a 14 character filename limitation. Also, some UNIX programs require certain suffix conventions for names of files to be processed. The following list includes the most common file suffixes:

- Include header file names end in `.h`
- ESQSQL source file names must end in `.ec`
- C source file names that contain no ESQSQL code must end in `.c`
- Relocatable object file names end in `.o`

Header Files

Header files are files that are included in other files before compilation by the C preprocessor. Some are defined at the system level such as `stdio.h` which must be included by any program using the standard I/O library. Header files are also used to contain data declarations and defines that are needed by more than one program. Do not use absolute pathnames for header files. Use the `<name>` construction for getting them from a standard place, or define them relative to the current directory with "name" (this is seldom done). The contents of header files should be functionally organized, i.e., declarations for separate subsystems should be in separate header files. Also, if a set of declarations is likely to

change when code is ported from one machine to another, those declarations should be in a separate header file or conditionally compiled in/out with "#if*" & "#endif" statements.

C Source Files

Suffixes of .c are required for C source files that have no ESQL statements. This is expected by the C compiler, cc. It is expected that any filename that contains this suffix can be compiled. Any file that includes ESQL statements must have a .ec extension. Any file with this extension will be processed by the ESQL preprocessor which results in creation of a .c file, which will be processed by the C compiler.

Note: Application programs will probably be made up of more than one source file. When combining functions into source files, group them according to similar purpose. For example, one might include initialization routines into the file named init.c, the mainline program in the main.c file, etc.

Object Files

Filenames containing .o are the object output from the C compiler. These files are linked together with the loader, ld, to form executable programs.

Variable Naming Conventions

Individual projects may have their own naming conventions. There are, however, some general rules to be followed.

- Initial and trailing underscores should not be used for any user-created names. UNIX uses leading underscores for names that the user should not have to know (like the standard I/O library). This convention is reserved for system purposes. Also note that the major CX library routines use a 3 character library name as their prefix (eg scr, dmm, ptp, etc).
- Macro names and define names should be in all capital letters
- typedef names should begin with capitals and continue with lower case letters
- Variable names, structure tag names, and function names should all be in lower case
- Variable names should be meaningful and readable. Avoid single character variable names with the possible exception of generally assumed loop indexes (eg, i, j, k). Use embedded underscores to delineate "words" in the name.

Function Names

Within an application, functions should be named in a consistent manner so that functions of similar purpose are similarly named (for example, all sort functions to end in "cmp", all get functions to begin with "get").

Functions defined in application libraries should follow this naming standard: if the function is to be available for use outside the library, prepend the function name with a 3 character library identifier and an underscore (for example, a function available to the registration applications from Libreg would begin with "reg_"; if the function is only to be used within the library, it should begin with an underscore.

Common CX Files for Program Development

Several files are included in virtually all application programs. These include the macro file, definition file, declaration file, make files, include files, and source files.

Macro file (mac.h)

The mac.h file contains preprocessor include and define statements, typedef statements, and structure template definition (not allocation) statements. Any macro substitution defines should be put here. These include numeric and character constant substitution (e.g., return error codes, statuses, etc.) as well as functional type macros (e.g., combining frequently used patterns into single macros). In addition to preprocessor statements, declarations of structures can be placed in this file. These declarations do not allocate the structures with names, they just define what the structure looks like. You may include various database

record "mac" files here as well as others. Use C comments to describe and delineate these various items in mac.h. This file is included in all source files during compilation via the dec.h file (see below).

Definition File (def.c)

The def.c file contains the declaration of (external) variables (including structures) that are to be available to all source files in the program. These variables can be initialized in this file as well. As with other C source, good use of comments and white space is encouraged to improve readability. The def.c file also includes mac.h and other appropriate database record "def" files. This file is utilized by the "makedec" command to create the dec.h file (see below).

Declaration File (dec.h)

The "makedec" command creates the dec.h file from the def.c file. The execution of this command is usually done within the make procedures in the source directory. Initializations are stripped and extern class is prepended to all variables. Inclusion of "def" files is modified to be an inclusion of appropriate "dec" files. This file should be included by the preprocessor in all C program source files.

Make Files

The UNIX Make processor is a program that utilizes dependencies specified in a "Makefile" and file modification times to perform actions such as compiling, loading, and installing programs. The use of makefiles for programs has provided an effective means of ensuring that the proper compilations are performed subsequent to changes that affect programs.

Include Files

Include files can be used to centralize information that multiple files will use. This aids in keeping that information up to date.

These files may also include compiler directives. It is recommended that compiler options that enable/disable program functionality not be used if at all possible. Such options make testing of various options more difficult and time consuming, since re-compilation is necessary to enable the options. Program arguments or table values should be used instead of compiler options wherever possible. If compile options are used, they should be driven by m4 macro definitions given in ENABLE type of macros.

Also, it is recommended that ccp macro values that are available for change by the user in include files be given by m4 macro values. The goal is to allow the user to make their desired modifications to these values by modifying an m4 macro file (under \$CARSPATH/macros) rather than an include file (under \$CARSPATH/include). Therefore a define in an include file might look like:

Example: #define REG_TICK "REG_TICK_CODE"

where REG_TICK_CODE is a macro defined in the macros directory.

Application include files are located in include/applic subdirectory. The include/util directory contains system and proprietary library includes. The include/custom directory is being phased out, to be replaced by the include/applic directory.

Any library include files must include ANSI function declarations. All include files designed to be installed in \$INCPATH and included by other libraries and applications will have an ANSI function declaration section that will declare ANSI prototypes for any function to be used by other programs or libraries. The ANSI section will be separated from the rest of the library with an `__STDC__` ifdef, as in the following example:

```
#ifdef __STDC__
int msg_add(int, char *);
int msg_clear(int);
int msg_dmmerr(struct dmm *);
int msg_errhandle(int, int, ...);
int msg_fprint(int, FILE *);
int msg_get(int, char *);
int msg_mqueue(int);
int msg_namgrp(int, char *);
int msg_number(int);
int msg_nextgrp(void);
int msg_print(int, char *);
int msg_progname(char *);
int msg_sendmail(char *, char *, char *);
int msg_debug(char *, char *, ...);
#else /* pre-ansi function declarations */
void msg_progname();
#endif /* __STDC__ */
```

Building a Make File

A Makefile is a text file containing targets, dependencies, and actions. In general, the action is performed when a dependency of the target has a newer modified time than the target. Makefiles are created for all CX programs (using 'Makefile' not 'makefile' for the name of the control file). This allows a programmer to make changes to a several source files and not have to worry about keeping track of those changes for compilation. When a file is changed, its modified time is updated. This implies that the source would then be newer than the object file. **Make** will cause the action to generate an object from the source, typically, a call to compile the source. Once the object is updated, it is newer than the executable program, therefore the action for creating an executable program will be done. This usually takes the form of linking all the objects into the program. **Make** is not limited to program compilation. Other targets and actions can be specified which will print source files, run the lint command, archive source into RCS format, and much more.

When new sources or directories are created and a makefile is desired, the "makeinit" command will create the makefile. This command will default to creating a makefile for a program, but can be given an argument to specify some other type of makefile.

Source File Organization

A file consists of various sections that should be separated by one or more blank lines. Although there is no maximum length requirement for source files, files with more than 800 to 1000 lines are cumbersome to deal with. Compilations will go slower, edit time and search time is slower, etc. Also, lines longer than 80 columns do not print normally on 8.5 inch paper and should be avoided. Excessively long lines which result from deep indentation are often a symptom of poorly-organized code.

The suggested order of sections for a file is as follows:

- The standard CX header should be the first thing in the file. A description of the purpose of the routine(s) in the file (whether they be functions, external data declarations or definitions, or something else) is more useful than just a list of the object names.
- Any header file includes, such as `dec.h`, should follow the header
- Any additional defines and typedefs that apply to the file as a whole are next (most of these should probably be in the `mac.h` file)
- Any additional variable definitions/declarations (most of these will be in the `mac.h` file)
- The function(s) come last. They should be in some sort of meaningful order. Top-down is generally better than bottom-up, and a breadth-first approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible after their calls).

General Coding Structure Rules

The following provides general guidelines on comments, arrangement of code, dealing with constants and expressions, and other issues related to coding appearance and structure.

Indentation

CX relies on vi as the primary editor for source and text files. This editing environment allows the use of the TAB and control-T sequence for indenting text lines. The standard tab stops are module 8 (9, 17, 25, etc). The control-T sequence allows indentations to be module 4. In the following discussion, indentation columns will be based on every four, combinations of control-T and/or TAB (this can be modified in vi) using `'set sw=??'`

Braces

As a general rule, opening braces will be alone on the following line indented 4 after the first character of the compound statement, structure definition, control structure, etc. The closing brace will be in the same column as the opening brace, and typically, be alone on a single line as well. Statements on intervening lines will be lined up between the braces.

White Space

Good use of white space (blank lines, tabs, etc) results in code which is more readable and easier to understand. The structure of the code can also be "diagrammed" by proper indentation and white space. Jenzabar recommends that you use these improvements in all source code you write.

Comments

There are three types of comments that should be utilized - block comments, comments within code, and comments appended to statements.

Comments that describe blocks of code, data structures, algorithms, etc., should be in block comment form with the opening `/*` in column one, followed by a space and 5 hyphens. The closing comment is similar with the 5 hyphens first, followed by a space, and the `*/` in column 7 and 8. Additional emphasis could be specified with an 80 column row of hyphens, but this is usually not necessary.

```

/* -----
   Here is a block comment.
   The comment text should be tabbed over
   and the opening and closing slash, asterisk, hyphens
   combinations should each be alone on a line.
----- */

```

As a position alternative, block comments inside functions may be indented to four columns less than the code they describe, however this is not required. Short comments may appear on a single line indented over to the tab setting of the code that follows.

```

if (argc > 1)
{
    /* Get input file from command line. */
    if (freopen(argv[1], "r", stdin) == NULL)
        error("can't open '%s'", argv[1]);
}

```

Very short comments may appear on the same line as the code they describe, but should be tabbed over far enough to separate them from the statements. If more than one short comment appears in a block of code they should all be tabbed to the same tab setting.

```

if (a == 2)
    return(TRUE);           /* special case */
else
    return(isprime(a));     /* works only for odd a */

```

Compound Statements

Braces ({ and }) are used to group declarations and statements together into a compound statement or block so that they are syntactically equivalent to a single statement. The opening left brace and the closing right brace should each be alone on single lines, indented 4 columns passed the beginning of the compound statement. The enclosed list should be lined up with the opening and closing braces.

```

if (expr)
{
    statement;
    statement;
}
else
{
    statement;
    statement;
}
for (i = 0; i < MAX; i++)
{
    statement;
    statement;
}
while (expr)
{
    statement;
    statement;
}

```

It is possible to have a need for an empty loop body in for and while control structures. If this occurs, it should be stated explicitly by using a comment before the semicolon on the line after the statement. For example:

```

for (i=0; (c = getchar()) != EOF; putchar(c), i++)
    /* null */;
while (more_to_do())
    /* null */;

do
{
    statement;
    statement;
} while (expr);

```

The do-while control structure contains an exception to the standard closing brace rule because the resulting while portion would cause confusion. Therefore the while follows the closing brace for the do-while control structure.

```

switch (expr)
{
    case ABC:
    case DEF:
        statement;
        break;
    case XYZ:
        statement;
        break;
    default:
        statement;
        break;
}

```

Note: When multiple case labels are used, they are placed on separate lines. The case statements also follow the general indentation rule and begin in the same column as the opening and closing braces. The non- case statements are indented an additional 4 columns.

The fall through feature of the C switch statement should rarely if ever be used when code is executed before falling through to the next one. If this is done it must be commented for future maintenance.

Long Lines

Breaking long lines to less than 80 columns and keeping things readable is sometimes difficult. The following rules should help:

- Break after for loop semicolons
- Break after function argument parameter commas
- Break after if logical operators
- Any remaining situations, break after operators with low expression precedence
- Long printf format strings can be put on the left wall if absolutely necessary

You may handle the remaining portion of long lines by a few different methods. If that portion is just a few characters, it might be desirable to right justify it near the end of the previous line. Otherwise, standard indentation applies. Additionally, loop segments might be broken on semicolons and indented an additional 4 columns for each segment on the following line. For example:

```

if (status = find_routine(target_buffer, length, string_to_search,
                        search_length);
    {
    ...
    }
for (dmm_start(&id_table_dmm);
    pid_type = (struct id_type *)dmm_getp(&id_table_dmm);
    dmm_next(&id_table_dmm))
    {
    ...
    }

```

Expressions and Constants

The following are standards for expressions and constants.

Expressions

The preferred use of operators is +=, =, *=, etc. In general, all binary operators except . and > should be separated from their operands by blanks. Some complex expressions may be clearer if the 'inner' expressions are enclosed in parentheses and/or are not blank separated. In addition, C statement keywords that are followed by expressions in parentheses should be separated from the left parenthesis by a blank.

Blanks should also appear after commas in argument lists to help separate the arguments visually. On the other hand, macros with arguments and function calls should not have a blank between the name and the left parenthesis. In particular, the C preprocessor requires the left parenthesis to be immediately after the macro name or else the argument list will not be recognized. Unary operators should not be separated from their single operand. Since C has some unexpected precedence rules, all expressions involving mixed operators should be fully parenthesized.

```

a += c + d;
a = (a + b) / (c * d);
strp>field = str.fl - ((x & MASK) >> DISP);
while (*d++ = *s++)
    /* NULL */;

```

Constants

Use numerical constants only when the numeric value is actually the intent of the usage. Use the define feature of the C preprocessor to substitute meaningful names to the intended numeric value. This will also make it easier to administer large programs since the constant value can be changed uniformly by changing only the define. These constant declarations should occur in the mac.h or, if available for change by the user, in the application's include file. The enumeration data type is an alternative way to handle situations where a variable takes on only a discrete set of values, since additional type checking is available through lint.

There are some cases where the constant 0 (zero) may appear as itself instead of as a define. For example if a for loop indexes through an array:

```
for (i = 0; i < ARYBOUND; i++)
    ...
```

The code does not.

```
fptr = fopen(filename, "r");
if (fptr == 0)
    error("can't open %s", filename);
```

In the last example the defined constant NULL is available as part of the standard I/O library's header file stdio.h and must be used in place of the 0 (zero).

```
if (fptr == NULL)
```

Avoid character constants as well. If a particular character field can have a few specific values, define meaningful names for those values and use them in the code. The result will improve readability and maintainability.

Syntax Changing

Don't change syntax via macro substitution. It makes the program unintelligible to all but the perpetrator.

Embedded Assignments

There is a time and a place for embedded assignment statements. The ++ and -- operators count as assignment statements. So, for many purposes, do functions with side effects. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable. An appropriate example would be the following common code segment:

```
while ((c = getchar()) != EOF)
{
    process the character
}
```

Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example:

```
a = b + c;
d = a + r;
```

The code should NOT be replaced by the following even though the latter may save one cycle.


```
d = (a = b + c) + r;
```

Note: In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade. Note also that side effects within expressions can result in code whose semantics are compiler-dependent, since C's order of evaluation is explicitly undefined in most places. Compilers do differ.

Ternary Operator

There is also a time and place for the ternary (`? :`) operator and the binary comma operator. The logical expression operand before the `?` should be parenthesized:

```
(x >= 0) ? x : x
```

Nested `?:` operators can be confusing and should be avoided. There are some macros like `getchar` where they can be useful. The comma operator can also be useful in for statements to provide multiple initializations or re-initializations.

GoTo Statements

Goto statements should seldom be used. One acceptable use of the `goto` is in the main function when there is need of going to an error processing section of code before exiting the program. The use of the `goto` in the case where there is need to break out of several levels of `switch`, `for` and `while` nesting might indicate that the inner constructs should be broken out into separate functions with return codes.

Variable Definitions and Declarations

Each variable declaration should be on a separate line with a comment describing the role played by the variable in the function. If the variable is external or a parameter of type pointer which is changed by the function, that should be noted in the comment. All such comments for parameters and local variables should be tabbed (or control-T) so that they line up underneath each other. At least one blank line (and probably, two) should separate the declarations from the function's first executable statement.

Variable declaration statements follow the general indentation rule by being indented 4 columns more than beginning of the declaration statement.

For structure and union template declarations, each element should be alone on a line with a comment describing it. The opening and closing braces should be alone on individual lines in accordance with the general rule. The standard statement indentation applies, also. In addition, it is recommended that tabs and control-T's be used to separate the member name from the member type so that the names are aligned. For example:

```

/* -----
   defines for boat.type
----- */
#define      KETCH      1
#define      YAWL       2
#define      SLOOP      3
...
struct boat
{
    int  wl_length;      /* water line length in feet */
    int  type;           /* see below */
};

```

These defines may also be put right after the declaration of type, within the struct declaration, with enough tabs and control-T's after # to indent define 4 columns more than the structure member declarations.

```

struct boat
{
    int    wl_length;      /* water line length
                           in feet */
    int    type;           /* boat type and values */
#         define  KETCH    1
#         define  YAWL     2
#         define  SLOOP    3
};

```

When initializing structures the opening brace and closing brace and semicolon should appear on individual lines. The member values should be indented to the respective member level and also be one per line (indented additionally if more than one line is needed). If initializing a previously specified template structure the equal sign follows the variable name and structure tag name on the declaration line. Otherwise, the equal sign follows the variable name on the same line with the closing brace (vertically matches opening brace).

Variable type grouping of declarations should be used to improve readability. This grouping is from most to least significant type specification and separated by blank lines. For example:

```

struct pencil_type    assoc;      /* desc here */
struct telephone     it;          /* desc here */
struct terminal       wyse;        /* desc here */
double desksize = 63.5;          /* desc here */
float lighter_than_air;          /* desc here */
float ing_point;          /* desc here */
int i;                          /* desc here */
int j;                          /* desc here */

```

The tabs between the type and the name are optional in a large list because the blank lines help the reader. However, in short lists using few or no blank lines, or in template declarations, the tabs or control-T's are recommended. Also, if storage class specification is used, tabs or control-T's may be utilize to enhance the appearance of the declarations.

Constants used to initialize long type variables should be explicitly long using the trailing capital letter L. It also follows that floating point and double constants should be initialized explicitly with a decimal point even if the value is whole (eg 1.5 3.0 4.2).

In any file which is part of a larger whole rather than a self-contained program, local variable and functions should be explicitly declared as such by using the static storage class keyword. If there is a clear need for the variable to be accessed from another file, the variable definition should be moved to the def.c file.

Function Definition

Function headers, length, return values and variable declarations should follow standard guidelines.

Function Header

Each function should be preceded by a block comment called a function header comment that provides a summary description of the function's purpose. The format of the header is:

```
/* -----  
=====   
Procedure: <procedure name>  
Description: <brief description of the function's purpose>  
Inputs: <arguments to the program and accompanying description;  
         also considered inputs are globally referenced variables>  
         example:  
         stucw - passed student structure buffer  
         stu.tot_reg - globally referenced total hours reg'ed.  
         stu.tot_aud - globally referenced total hours audited.  
Returns: <list of ALL possible return values from function and what  
         each means>  
         example:  
         REG_OK - no errors  
         REG_ERR - student prog_enr_rec not found  
         REG_FAIL - fatal error.  
Outputs: <list any inputs whose value has been changed by this  
         function>  
         example:  
         stucw  
         stu.tot_reg  
         stu.tot_aud  
Notes: <any unusual issues in the function that should be  
         especially noted>  
=====   
----- */
```

Properly creating and maintaining the function headers is a significant help in program maintenance and clarity. It can also assist in the development process by forcing the developer to clarify the function's purpose and how it relates to other functions.

Function Return Types and Parameters

If the function returns a type of value other than "int", then its declaration should proceed the function name. If the function does not return a value then it should be given the return type of "void". If the value returned requires a long explanation, it should be given in the function header; otherwise it can be on the same line as the declaration after the formal parameters. Each parameter should be declared (do not default to "int"), with a comment on a single line.

Function Variable Declarations

The opening brace of the function body should be alone on a line beginning in column 1. All local declarations and code within the function body should be indented 4 columns. If the function uses any external variables (other than those in the **dec.h** or in the top of the file), these should have their own declarations in the function body using the **extern** keyword. If the external variable is an array the array bounds should be repeated as a comment on the declaration. There should also be extern declarations for all functions called by a given function (other than those in the **dec.h** or in the top of the file). This is particularly beneficial to someone picking up code written by another. If a function returns a value of type other than int, it is required by the compiler that such functions be declared before they are used.

A local variable should never be redeclared in nested blocks. In fact, avoid any local declarations that override declarations at higher levels. Local variables within blocks, other than at the beginning of functions, may cause trouble with the symbolic debuggers.

Function Length

Functions will differ in length as they are diverse in purpose. However, a function should generally be one to two pages in length and probably never be longer than three to four pages. Lengthy functions become hard for the reader to follow. Execution structure is not as obvious and will be confusing. Therefore, keep the functions small.

Function Endings

All functions will be non-brace terminated. Main line programs will be terminated with an exit call and functions will be terminated with a return call. The only exception to this is if the routine is terminated other than at the bottom. In this case, the intent should be commented and indicated at the bottom of the routine before the final closing brace.

Functions and Macros

Care is needed when interchanging macros and functions since functions pass their parameters by value whereas macros pass their arguments by name substitution. This difference also means that carefree use of macros requires care when they are defined. Remember that complex expressions can be used as parameters, and operator-precedence problems can arise unless all occurrences of parameters in the definition have parentheses around them. There is little that can be done about the problems caused by side effects in parameters except to avoid side effects in expressions (a good idea anyway).

The following examples illustrate some of these points.

```
skyblue(hour)
int hour;
{
    if (hour < MORNING || hour > EVENING)
        return(FALSE);           /* black */
    else
        return(TRUE);            /* blue */
}
```

In the above example, the dec.h file defines MORNING, EVENING, FALSE, and TRUE. NODE and NULL are defined in the dec.h file in the following example. Note, that NULL is defined in the standard include file stdio.h. CX takes advantage of this by including stdio.h in the dec.h file.

```
NODE *tail(nodep)
NODE *nodep;           /* pointer to head of list */
{
    register NODE *np;  /* current pointer advances to NULL */
    register NODE *lp;  /* last pointer follows np */
    np = lp = nodep;
    while ((np = np->next) != NULL)
        lp = np;
    return(lp);
}
```

Portability

The advantages of portable code are well known. This section gives some guidelines for writing portable code, where the definition of portable is taken to mean that a source file contains portable code if it can be compiled and executed on different machines with the only source change being the inclusion of possibly different header files. The header files will contain defines and typedefs that may vary from machine to machine. The following is a list of pitfalls to be avoided and recommendations to be considered when designing portable code:

Separate Portable and Non-Portable Code

First, one must recognize that some things are inherently non-portable. Examples are code to deal with particular hardware registers such as the program status word, and code that is designed to support a particular piece of hardware such as an assembler or I/O driver. Even in these cases there are many routines and data organizations that can be made machine independent. Jenzabar suggests that you organize the source file so that the machine-independent code and the machine-dependent code are in separate files. Then if you move the program to a new machine, it is a much easier task to determine what you need to change. It is also possible that code in the machine-independent files may have uses in other programs.

Avoid Dependence On Word Sizes.

The following describes general minimum and maximum sizes that should be expected of C types. These rules are not hard and fast, just rules of thumb.

<u>type</u>	<u>minimum</u>	<u>maximum</u>	<u>low numeric</u>	<u>high numeric</u>
short	8	16	+/-127	+/-32767
int	16	32	+/-32767	+/-2147483647
long	32	32	+/-2147483647	+/-2147483647

Note: The C type char has not been included because numerics should not be maintained in char type variables. Any unsigned type other than plain unsigned int should be typedefed, as such types are highly compiler-dependent. This is also true of long and short types other than long int and short int. Programs will have a central header file (mac.h) which supplies typedefs for commonly-used width-sensitive types, to make it easier to change them and to aid in finding width-sensitive code. If a simple loop counter is being used where either 16 or 32 bits will do, then use int, since it will get the most efficient (natural) unit for the current machine.

Beware of making assumptions about the size of pointers. They are not always the same size as int. Nor are all pointers always the same size, or freely interconvertible. Pointer-to-character is a particular trouble spot on machines which do not address to the byte.

Specific Bit Representation

Word size also affects shifts and masks. For example:

Example: `x &= 0177770`

The above clear only the three rightmost bits of an int on a PDP11. On a VAX it will also clear the entire upper half word. Use the following instead which works properly on all machines

Example: `x &= ~07`

The or operator (|) does not have these problems, nor do bit fields (which, unfortunately, are not very portable because of defective compilers).

Do not use code that takes advantage of the two's complement representation of numbers on most machines. Optimizations that replace arithmetic operations with equivalent shifting operations are particularly suspect. You should weigh the time savings with the potential for obscure and difficult bugs when your code is ported to another machine.

Special Character Expectations

Do not use signed characters. On the PDP-11, characters are sign extended when used in expressions, which is not the case on any other machine. In particular, getchar is an integer-valued function (or macro) since the value of EOF for the standard I/O library is 1, which is not possible for a character on the AT&T 3B or IBM PC. Code which assumes either that characters are signed or that they are unsigned is unportable. It is best to completely avoid using char to hold numbers. Manipulation of characters as if they were numbers is also often unportable.

Alignment Considerations

Alignment considerations and loader peculiarities make it very rash to assume that two consecutively declared variables are together in memory, or that a variable of one type is aligned appropriately to be used as another type. The DEC processors number the bytes from right to left within a word. Most other machines number the bytes from left to right. Hence any code that depends on the left-right orientation of bytes in a word deserves special scrutiny. The same applies to bit fields. Bit fields within structure members will only be portable so long as two separate fields are never concatenated and treated as a unit.

Boolean Testing

Do not default the boolean test for non-zero.

Example: `if (f() != FAIL)`

The above is better than the following even though FAIL may have the value 0 which C considers false.

Example: `if (f())`

This will help you out later when somebody decides that a failure return should be 1 instead of 0. A particularly notorious case is using `strcmp` to test for string equality, where the result should never ever be defaulted. The preferred approach is to define a macro such as STREQ:

Example: `#define STREQ(a, b) (strcmp((a), (b)) == 0)`

An exception is commonly made for predicates, which are functions that meet the following restrictions:

- Has no other purpose than to return true or false.
- Returns 0 for false, 1 for true, nothing else
- Is named so that the meaning of (say) a `true' return is absolutely obvious (example predicate name: `is valid` or `valid`, not `checkvalid`).

Numeric Values

Be suspicious of numeric values appearing in the code. Even simple values like 0 and 1 (used as false and true) could be better expressed using defines like FALSE and TRUE (see previous item). Any other constants appearing in a program would be better expressed as a defined constant. This makes it easier to change and also easier to read.

Function Argument Evaluation Order

Do not expect particular order of function argument evaluation. This may suggest that side effects are being used in the code. Avoid side effects if possible.

Project Dependent Standards

Individual projects may wish to establish additional standards beyond those given here. The following issues are some of those that might be addressed for a particular project.

- What additional naming conventions should be followed? In particular, systematic prefix conventions for functional grouping of global data and also for structure or union member names can be useful.
- What type of include file organization is appropriate for the project's particular data hierarchy?
- What procedures should be established for reviewing lint complaints? A tolerance level needs to be established in concert with the lint options to prevent unimportant complaints from hiding complaints about real bugs or inconsistencies.

User Interface Standards

It is imperative that interactive application programs use common standards for menus, windows, error and message handling, and other items that pertain to how the user views and uses the program.

Screens

The screen format, screen prompts and pop-up windows all have standard guidelines that should direct their development. Some general guidelines include:

- Do not use `scr_getc`

- If an application must do its own table lookup, use the scr_dortable function

Format

Below are important standards guidelines for program screen format:

- All screens should have a centered title; this title should be specified in the program screen via the SCREEN_INFO attribute

```
SCREEN_INFO: optional,
             gui_title = "Gifts";
```

- White space, rather than dashed lines, should be used to separate logical data groups on the screen
- Program screens are to be stored under the module: modules/<module name>/progscr/<program name>/<screen name>
- Every field must contain a comment
- Comments should begin with "Enter" unless the comment is a question
- The phrase "Valid values are:" will not be used in comments; instead the values will be displayed in the comment

```
example:
comments = "Enter salutation type. (I)nformal, (F)ormal."
```

- Minimize hardcoding of values in progscr; use macros whenever the value is referenced multiple times in different progscr or menu options, etc. Common comment macros include: COMMENT-QUERY (for name lookup), COMMENT-TBL (for table lookup), and COMMENT-YN (for giving yes-no options. Review existing progscr and the macros/custom/comment file for more information.
- If there are more valid values than can be displayed on one line, then an example should be used:

```
"Enter the type of accomplishment, ACCOMP_TYPE_EG".
which expands to
"Enter the type of accomplishment, eg: ACADEMIC, ATHLETIC."
```

- Comments will use normal upper/lower case convention
- Comments will end with either a ">" or a "?"
- The first character of each screen descriptor will be capitalized
- The colon will not be used in comments; the only exceptions will be after the work "format" and with the use of "eg"

```
Format: mm/dd/yyyy.
Enter calendar year, eg: 1990.
```

- The attributes section of progscr files should maintain the following structure: list one attribute per line in alphabetical order, indented 5 spaces; the one exception to this is the optional attribute which should immediately follow the database field name on the same line. The lookup and joining clauses should be listed on one line.

Prompts

Prompt lines for screens should always be generated in an application program using the library functions `scr_prompt2` and `scr_mesg2`. See the platform documentation for how these functions are to be used. Also, the last argument to the menu structures passed as parameters to these functions is the icon identifier. For all options in an application, either the default icon for that option is used, or an icon is specified in the menu definition that is appropriate for the option. When run in gui-mode, all options in an application should have an accompanying icon.

The standard options used for `scr_getset` functions are:

SCR_DONE	Finish
SCR_ABORT	Cancel

The commonly used options used for `scr_scget` functions are:

SCR_DONE	Finish
SCR_ABORT	Cancel
SCR_ADD	Add
SCR_ERASE	Delete

Windows

Pop up windows are used extensively in applications as a means to provide access to a wide range of data. In order for the gui interface to work properly, it is important that the `scr_pushwin` and `scr_popup` calls follow one another, without any other screen library calls between them.

When a pop up window is used to display a `progscr`, keep in mind that the number of lines in a screen determines its height, and the width is determined by the longest line in the screen definition. For a pop up window to display the border around the screen, it is necessary to have at least two blank spaces on both the left and right sides of the screen.

Output/Mail

In an application that produces a lot of output (like an audit program), use a header similar to the type used in ACE reports. Include the date and time, school, report title, etc. Audit output should be saved to a file in the user's home directory, or preferably, in the audit directory, or both.

In an application that performs some batch process, it is recommended that a status mail message be sent to the user upon completion of the program. This status mail might include number of IDs/students processed, etc.

- If the program terminates unsuccessfully, a mail message explaining the reason must be generated
- All mail handling in an application should be done using the `msg` functions provided in `libmsg`

Errors/Messages to User

It is important that you handle status and error messages consistently across applications. Also, be sure that any message, whether just informative or used to describe a fatal error condition, includes sufficient information to understand the condition described.

Which Function to Use When

Below are some guidelines for how to choose the appropriate library function to handle particular messages to the user:

- `scr_info`: use when you need to tell something to the user, and it is acceptable, or preferable, to have the user acknowledge this message by pressing the return key. This is primarily for messages meant to inform where it is important that the user be aware of the message content.

- `scr_dialog`: use when user response is required, and the response is not a simple yes or no, nor just an acknowledgment.
- `scr_yesno`: use when you need to ask the user a question that has only two possible responses - yes and no.
- `scr_askexit`: use when you are processing the command that causes the program to exit. This function will ask the user to confirm that he/she wishes to exit the program.
- `scr_perror`: use for field level errors that prevent the user from continuing to the next field.
- `scr_pstatus`: use for simple status messages where it is acceptable that the user may not be aware of them. These are normally considered "optional" kinds of information, such as messages that tell the user the screen is loading, table is loading, student data loaded, etc. `msg_errhandle`: use for all error messages that result in the termination of the program. Also use for warning or status messages that are important for the user to acknowledge.

How to Handle Errors

Program errors should be handled as close as possible to where the error occurs. If the error occurs 8 function levels down in the program, print the message at that time; do not leave error message handling to the calling function(s).

Always use the library function `msg_errhandle` to handle program errors. Error messages should include a description of the process or function having the error, and appropriate data values that might help identify the nature of the problem. For example, if an ESQL statement results in a row not found error, it is important to identify the data values that were being used. Some sample error messages follow:

```
(void)msg_errhandle(MSG_ERR_MAIL, _displ,
  "Database open error. Database: %s. Status=%d", dbname,SQLCODE);
(void)msg_errhandle(MSG_ERR_MAIL,_displ,
  "scr_print error: %s",scr_errm());
(void)msg_errhandle(MSG_ERR_MAIL,_displ,
  "Program enrollment record not found for ID: %d Prog: %s",
  p_e->id, p_e->prog);
(void)msg_errhandle(MSG_ERR_MAIL, TRUE,
  "Cursor error on %s. Status=%d", AUDIN_REC, SQLCODE);
```

Always provide the SQLCODE value in any error message describing an ESQL error or warning. Mail all fatal errors to the user. When the application receives an error from a library function, include the error message string available from the library (see second example, above).

Menus

All interactive application programs will use ring menus wherever possible to control option access by the user. The library function, `scr_getmenu`, will be used to handle the menu processing. Some guidelines for menu creation follow:

On the main ring menu, place the name of the process, in user terminology and capitalized, as the text for the ring menu (first argument to the `getmenu` function). Example:

```
REGISTRATION: Query Register ...
```

On subsequent menus, the mode description, also capitalized, is given on the menu line. Example:

```
REGISTER: DOWN ARROW add. TAB enroll. ...
```

Example C Code

When creating C-code, keep in mind the following:

- Make proper use of white space and comments so that the structure of the program is evident from the layout of the code
- When writing code is that it is likely that you or someone else will be asked to modify it or make it run on a different machine sometime in the future

The following is example C code.

```

/* -----
=====
      C Style Summary Sheet (Modified from H Spencer, U of Toronto)
      Block comment describes function
      Standard header should be above (or replacing) this comment
=====
----- */
#include "dec.h"                                Headers
#define STREQ(a, b)      (strcmp((a),(b)) == 0) /* ... */
#define ERROR           5                       /* ... */
struct bar
{
    Segno      alpha;      /* ... */
    int        beta;      /* ... */           Don't assume 16 bits
};
static char *foo = NULL;      /* ... */           Static whenever poss
/* -----
      Start routine here
----- */
static blech(a)
int a;                                /* ... */           Don't default int
{
    int        bar;          /* ... */
    extern    int    errno;  /* ..., changed in this routine */
    extern    char  *index();
    if (foobar() == FAIL)
    {
        return(ERROR);
    }
    while (x == (y & MASK))
    {
        f += ((x >= 0) ? x : x);           Parents improve reading
    }
    do
    {
        /* Avoid nesting ( ? : ) */
    } while (index(a, b) != NULL);
    switch (...)
    {
        case ABC:
        case DEF:
            printf("...", a, b);
            break;
        case XYZ:
            x = y;
            /* FALLTHROUGH */
        default:
            break;                               /* Limit imbedded = s */
    }
    if (!isvalid())
    {
        errno = ERANGE;
    }
    else
    {
        x = &y + z>field;
    }
    for (i = 0; i < BOUND; i++)
    {
        /* Use lint hpcax */
    }
    if (STREQ(x, "foo"))
    {
        x |= 07;
    }
    else if (STREQ(x, "bar"))
    {
        x &= ~07;
    }
    else if (STREQ(x, "ugh"))
    {
        /* Avoid gotos */
    }
    else
    {
        /* last else */
    }
    while ((c = getc()) != EOF)
        /* NULL */;
    exit(0);
}

```

Software Maintenance Standards

Introduction

The following are standards for Software Maintenance Orders.

Product Advisory

The Product Advisory is a periodic publication sent to Jenzabar coordinators on client campuses. The Advisory is distributed in two forms, a hard-copy format put together with MS Word and an electronic version distributed over Internet to all subscribers of the CARS-PA list.

The publication contains the following types of information:

- Known problems with the operation of the software, CX, Informix or UNIX
- Suggestions to improve system performance or to solve problems that several clients have experienced
- To provide minor program, report, script, etc., patches to correct an error condition. These are distributed prior to their release in a SMO.

For the programmer/analyst, the product advisory is most often used to provide a "fix" to a problem that can be performed by clients themselves with relatively simple instructions. In this way, a problem can be corrected temporarily without immediate need for a fix SMO. It is the analyst/project leader's responsibility to determine if an advisory item is required to deal with a problem.

The Product Advisory is prose in nature, describing the problem or situation and then providing the solution. Corrections to programs, screens, reports, etc., are provided as the client sees them in the respective file. Before and after images are given in most cases.

The Senior Product Manager is currently responsible for the publication of the Product Advisory. Information for a future advisory is submitted through e-mail, file, etc., as long as it is a ASCII text file that can be manipulated using UNIX tools. At the present time, vi, is used to format the electronically distributed advisory information. The information from the file is manipulated, enhanced, dated and put into the database.

Several rounds of update are made before any advisory is distributed. Suggestions for modification are solicited from the product managers, the developers, the quality assurance staff and the research staff.

Product Issues

The purpose of the product issue process is to alert the product managers and development staff of potential oversights, bugs or problems in general, that exists within the product.

The Response Center, in their callentry record can mark a client call for "Product Review". A report is run by the product managers that prints out all of the calls marked for product review within a specified date range. The product managers review the call record, Notes and Solution to determine the nature of the problem or referral. Within the callentry program are tables for update that are used to track the issue as it is reviewed. The review classifies the call as Referred, In progress, Solved, Pending, Done, Merge, Zap or Blackhole.

If the review finds that a product change must be made then an entry in the Suggest file is made. On the CX database, a file called "suggest" contains information on suggested enhancements and reported bugs within the system.

Entries are added to the suggest file based on issues from the response center as well as issues brought up internally. Tracking information can relate the entry back to the call, if it originated as a response center issue. Once a change has been made to the system on a suggest file item, the suggest entry is updated to reflect a solution has been added to the product, should an update be necessary.

Program Documentation Standards

Introduction

Program documentation contains an abstract and the following sections: introduction, procedures, parameters, compilation values, program flow, program errors, crash recovery, database input, database output, and output samples. The introduction and procedures sections are designed specifically for the end user, while the other sections are intended for the Jenzabar coordinator.

Note: These standards are intended for program documentation; the overall guidelines are the standards for CX documentation. Where the above guidelines do not specify exactly how to format any section, readability for the targeted audience is the key. Thus, the intent of program documentation is to explain a program so the coordinator will better understand how the program works and how to resolve program errors and problems.

Abstract

The abstract of the document is a short paragraph about the program and contents of the document. Observe the following guidelines:

1. Do not use program names (except system utilities like FPS, PERFORM, ...).
2. Keep it short -- three to six formatted lines.
3. Use key words later referenced in the document.
4. Do not define terms in the abstract.
5. Avoid use of terms other than standard CX terms.
6. Do not include phrases such as "This document contains".

Introduction Section

The purpose of the introduction is to give a brief overview of the purpose of the program and what is covered in the document. Observe the following guidelines:

1. Give the program name.
2. Keep it general but meaningful.
3. Use the heading "Introduction" instead of "Purpose", etc.

Procedures Section

The procedures section describes the options available and the information displayed in an interactive program. This section does not apply to background programs. For interactive programs, give screen examples and list the command options available in the program organized by each screen displayed. Observe the following guidelines:

1. Use the schema file and field names where appropriate.
2. Make the language as plain as possible.
3. Include any screens used by the program with informative data displayed.

Parameters Section

All parameters accepted by the program must be specifically documented. The coordinator will use this section to set up menu options, etc. The end user may also use it to gain an understanding of what menu options require. Observe the following guidelines:

1. Use the schema file and field names where appropriate.
2. Provide examples of the use of each parameter.
3. Include sample MENU parameter prompts (PP) and parameter assignment (PA) lines for each parameter.

Compilation Values

The compilation values used by the program that are included in the directory "\$CARSPATH/include/custom" should be listed and described in this section. If the coordinator can change any of the values, indicate what kind of decisions should and/or could be made. Also, include procedures on how to change the flag and remake the source.

Program Flow Section

A (block) diagram of the procedures followed by the program will be included in each document. The diagram will give a picture of the logic flow of the program in general terms, followed by a more detailed explanation. Observe the following guidelines in the diagram:

1. Keep the diagram simple -- it is not a detailed flowchart.
2. Number each procedure box in the diagram.
3. Link boxes in the diagram showing logic flow.
4. The detail following the diagram is intended to give the coordinator insight into how data is being used.
5. Refer to each box by its number in the diagram.
6. List FPS and SCR screen structure binds and any special field names used, selection criteria for database records, keys and "joins" used in record selection, and any important features or special use of data values.
7. Use schema file and field names where appropriate.
8. Leave out anything that is not useful to the coordinator.

Program Errors Section

Give an alphabetical listing of all error messages that the program generates. This does not include system or utility software errors except where the error is a common recurring error unique to a program, or the resolution of the error is based on program operation or data input. This section should also include a description of what the error means, what caused the error, and how to resolve the error. If any error is to be resolved by personnel at CISC (i.e. it is beyond the usual coordinator's ability and understanding), describe research procedures so that as much information as possible is found and recommend that they refer the matter to their account manager.

List the error messages so you can find them quickly and easily. For programs that have few error messages with short descriptions, just list the error messages and follow them with one or two short paragraphs. Where there are many errors or long discussions on the errors, Jenzabar recommends that you list the errors in an index format so the reader can look up the error easily in one section, and then refer to another section detailing the error.

Crash Recovery Section

Outline recovery procedures so the coordinator can recover the program to allow users to continue processing. List any specifics regarding data recovery and cleaning up any system files that are not complete. It may not be possible or desirable to outline recovery procedures that involve complicated decision making and data or file manipulation. In this case, recommend that

the coordinator look up as much information as possible, and refer the matter to their account manager.

Database Input and Output Sections

Each document will have a list of all schema files that are used by the program. The input section lists all schemas read as input, and the output section lists all schemas that are updated. This is only a quick reference, alphabetized list. No explanation of the purpose of the schemas should be included here since database interaction is covered in the "Program Flow" section above.

Output Samples Section

Any hard copy output that that the program generates should include a representative sample included. This section will only include standard CX package forms and reports.

INDEX

—
_varaccum function, 287
_vardef function, 285
_varget function, 286
_variaccum function, 289
_variget function, 287
_varistore function, 286
_varpctcor function, 289
_varpctold function, 289
_varstore function, 286

A

abbreviations
 data standards, 333
 schema standards, 347

absolute holds, 85

abstracts
 in program documentation, 417

academic year macros, 147

accessing
 Configuration table, 50

accessing
 Accomplishment table, 36
 ADR table, 38
 Alternate Address table, 40
 Building table, 44
 Citizen table, 46
 common macros, 139
 Communication table, 48
 Contact table, 53
 Country table, 56
 County table, 58
 Day table, 60
 Degree table, 62
 Denomination table, 64
 Division/Department table, 66
 Entry Selection table, 70
 Ethnic table, 73
 Exam table, 75
 Facility table, 77
 Form Entry program, form files, 166
 Form Order table, 80
 Handicap table, 83
 Hold table, 85
 ID Office Permissions table, 90
 Interest table, 92
 Involvement table, 94
 Marital table, 96
 Occupation table, 98
 Office table, 100
 Permission table, 102
 Privacy Field table, 104

 Privacy table, 104
 Relationship table, 106
 scearray functions, 283
 Sort Criteria table, 70
 State table, 108
 stored procedures, 26
 Subscription table, 110
 Suffix table, 112
 Tickler table, 114
 Title table, 116
 Veteran Chapter table, 120
 Zip Code table, 124

Accomplishment record, 34

Accomplishment table, 36

ACE
 in database dictionary, 329

Ace reports
 debugging, 280

ACE reports, 269–93
 acearray functions, 283
 after group, 380
 aggregate commands, 275
 before group, 380
 body, 380
 command summary, 270
 compiling, 381
 creating menu options, 382
 define statements, 378
 example sortpage macros, 216
 examples, 275–77
 execution, 381
 footers, 381
 formatting, 278
 formatting footers, 279
 formatting page headers, 278
 formatting page trailers, 279
 last page, 381
 location, 378
 mandatory sections, 270
 numeric fields, 380
 on every record, 380
 on last record, 380
 output, 381
 output definition, 379
 page specifications, margins, 274
 print commands, 274
 read statements, 379
 report format, 379
 running, 270
 sort clauses, 379
 SQL functions, 294
 standard header, 370, 378
 standards, 378
 support, 382

- testing, 382
- translating, 381
- translation, 381
- variables and functions, 273

acearray functions, 283–91, 283

- usage, 284
- variables, 284

Add-ID command

- on detail windows, 170
- results of selecting, 171

Addressee record, 34

Addressing record, 34

ADR table, 38

after group

- in ACE reports, 380

aggregate commands

- in ACE reports, 275

Alter table, 9

Alternate Address record, 34

Alternate Address table, 40

Alternate Recipient window, 166

analysis output, 18

attributes

- menu descriptions, 321
- menu options, 314
- PERFORM screens, 371
- tables, 13

audit columns, 22

audit script example, 211

audit scripts

- in Database Administration, 212

audit trail permissions, 23

audit trails, 21

auditing

- id records, 186

audits

- in Database Administration, 210
- in *mergeid* program, 194

B

batch mode

- mergeid* program, 206

batch mode, *mergeid* program, 194

before group

- in ACE reports, 380

Building table, 44

bulk mail

- in sortpage program, 218

business macros, 142

Business record, 34

C

C programs

- relationship to macros and includes, 128

campus building/facility macros, 142

carsu, 21

check functions, 226, 248

Church record, 34

Citizen table, 46

column types

- in tables, 16

columns

- default values, 17
- in tables, 16
- table, 31

commands

- alter table in DBMAKE, 9
- MKSPooler, 308
- PERFORM screens, 254
- sortpage program, 218
- used in ACE reports, 270

common enable macros, 139

common includes, 157

common macros, 139

common periodic macros, 141

common programs, 161–212

common records, 34–35

communication management macros, 142

Communication table, 48

community college macros, 146

compilation

- make** processor, 371

compilation values

- in program documentation, 418

compile process

- stored procedures, 28

compiling

- ACE reports, 381

Configuration table, 50

constraint analysis

- in schemas, 18

constraints. See DBMAKE constraints

- analysis output, 18
- field level, 19
- implementing, 20
- table level, 19

Contact BLOB record, 34

Contact Detail record, 34

Contact Image record, 34

Contact record, 34

Contact table, 53

controlling menu access, 325

Country table, 56

County table, 58

crash recovery

- idaudit, 191
- section in program documentation, 418
- sortpage program, 219

creating

- entry library screens, 377
- form definition files, 256

- menudesc files, 322
- menuopt files, 318
- menuoptions for ACE reports, 382
- schemas, 8
- screen definition files, 256
- spool queues, 308
- criteria,for merge ID, 193
- custom include files, 156
- custom macros, 135

D

- data flow diagrams
 - in program documentation, 418
 - sortpage, 217
- data integrity, 333
- data records
 - standards, 332
- data tables
 - standards, 332
- database
 - definitions in dupid program, 176
 - files in dupid program, 175
- Database Administration program, 192–212
- database dictionary, 327, 328
 - ACE reports, 329
 - application software, 330
 - definition, 328
 - elements, 330
 - files, 333
 - make processor for schema, 330
 - PERFORM screens, 329
 - source, 330
 - standards, 328
- database dictionary, 328
- Database Field record, 34
- Database File record, 34
- database input
 - section in program documentation, 419
- database output
 - section in program documentation, 419
- date macros, 143
- Day table, 60
- dbadmin. *See* Database Administration program
- dbmake
 - alter table processing, 9
 - attributes in schema, 13
 - constraints
 - conflict, 20
 - unique, 20
 - environment variable, 8
 - ISAM error, 20
 - options, 8
 - rebuilding check constraints, 20, 22
 - trigger names, 22
 - triggers, 22

- action statements, 22
 - syntax, 22
- unique indexes, 20
- DBMAKE
 - alter table command, 9
 - schema table section, 13
- debugging Ace reports, 280
- def.c
 - specifying ptp functionality, 241
- def.c file
 - linking tables, files, 236
 - local functions, 226
 - macros, 223
 - program parameters, 228
 - scroll screens, 230
 - special check functions, 239
 - specifying tables for screens, 231
 - specifying update order, 234
 - variables, 224
- define statements
 - in ACE reports, 378
- definitions
 - SQL tables, 31
- Degree table, 62
- deleting
 - ID records, 186
- deleting records
 - in Schedule Entry program, 213
- Denomination table, 64
- Department table, 66
- DESC attribute, 13
- differences
 - in product, 1
- Division table, 66
- Division/Department table, 66
- documentation
 - standards for program documentation, 417
- dupid. *See* Duplicate ID Detection program
- duplicate ID
 - background mode, 177
 - database definitions, 176
 - database files, 175
 - interactive mode, 179
 - merging, 192
 - program arguments, 174
 - review mode, 183
 - test functions, 172
 - testing limitations, 178
- Duplicate ID Detection program, 172–85

E

- Education record, 34
- Employment record, 34
- enable macros
 - common, 139

- enrollment status macros, 143
- Entry Library program, 221–52
 - adding tables, 222
 - binding columns, 237
 - definition file macros, 223
 - linking tables, fields, 236
 - program parameters, 228
 - scroll tables, 230
 - special check functions, 239
 - special flags, 231
 - specifying local functions, 226
 - specifying update order, 234
 - table level functions, 232
 - tables for screens, 231
 - updating records, 237
 - variables, 224
- entry library programs
 - screen standards, 374
- Entry Library programs
 - ptp functionality, 241
 - transaction procedures, 252
- Entry Selection table, 70
- environment variables
 - dbmake, 8
- ESQL
 - programming standards, 396
- Ethnic table, 73
- Event record, 34
- events/scheduling macros, 143
- Exam table, 75
- Examination record, 34
- examples
 - ACE report sorting, 303
 - ACE reports, 275–77
 - audit script, 211
 - def.c add field array, 238
 - def.c check function array, 240
 - def.c common fields array, 237
 - def.c macros, 223
 - def.c update field array, 238
 - def.c update order array, 235
 - def.c variables, 225
 - Entry Library program parameters, 229
 - Entry Library ptp functionality, 242
 - includes, 153
 - local functions, 227
 - macro file, 130
 - menu, 320
 - menu options, 389
 - menudesc file, 320
 - menuopt file, 312
 - schemas, 29–30
 - section in program documentation, 419
 - sortpage macros ACE report, 216
 - special flags, 233
 - special functions, 251
 - transaction procedure in Entry Library programs, 252
 - triggers, 25
- executing
 - ACE reports, 381
- Expanded Merge Item window, 205

F

- Facility table, 77
- faculty macros, 144
- Faculty record, 34
- fatal error messages
 - programming standards, 412
- fatal errors
 - in program documentation, 418
- field descriptions
 - locating, 33
- fields
 - order in entry screens, 374
 - order in perform screens, 374
- Fields By File report, 33
- Fields By Track report, 33
- file format macros, 144
- file transfer macros, 144
- files
 - data dictionary, 333
 - menu definition
 - PERFORM screens, 372
 - menudesc, 320
 - menuopt, 312
 - menuparam, 323
 - naming conventions, 397
 - nomenu, 325
 - nomenu_\$CARSDb, 325
 - stored procedures, 26
- Files By Track report, 33
- footers
 - in ACE reports, 279, 381
- Form Entry program, 161
- Form Order History window, 166
- Form Order table, 80
- form/label macros, 144
- format
 - schema file, 10
- formatting
 - ACE reports, 278
 - in ACE reports
 - page headers, 278
 - page trailers, 279
 - PERFORM screens, 370
- forment. See Form Entry program
- Forms Order record, 35
- functions
 - _varaccum, 287
 - _vardef, 285

- _varget, 286
- _variaccum, 289
- _variget, 287
- _varistore, 286
- _varpctcor, 289
- _varpctold, 289
- _varstore, 286
- ACE reports, 273
- acearray, 283–91
- check, 248
- GET_PRIMARY_REC, 244
- IS_DISPLAY_ONLY, 247
- special, 250

G

- GET_PRIMARY_REC function, 244
- Group Scheduling record, 35

H

- Handicap table, 83
- Hold Action table, 85
- hold permissions, 85
- Hold record, 35
- Hold table, 85

I

- ID Audit
 - permissions, 186
 - program arguments, 187, 207
- ID Audit program, 186–91
- ID column, for *mergeid*, 193
- ID Contact record, 35
- ID Entry program, 170–71
- ID Office Permissions table, 90
- ID record, 35
- ID records
 - auditing, 186
 - duplicate records, 186
 - maintained in detail windows, 170
- ID/Profile macros, 145
- idaudit. See ID Audit program
 - crash recovery, 191
- identry. See ID Entry program
- Image Document record, 35
- implementing
 - constraints, 20
 - triggers/audit trials, 23–24
- includes, 149–59
 - common, 157
 - custom, 156
 - customizable, 155
 - examples, 153
 - file contents, 151
 - file types, 154

- function, 150
- implementing, 152
- macro dependency, 149
- non-customizable, 155
- relationship to macros and C programs, 128
- setting up, 159
- indexes
 - in *mergeid*, 194
 - in schema, 12, 346, 347
- INFORMIX
 - data files, 328
 - DBMS, 328
- input parameters
 - stored procedures, 28
- interactive mode
 - mergeid* program, 200
- interactive mode, *mergeid* program, 194
- Interest record, 35
- Interest table, 92
- Interest table macros, 146
- introduction
 - section in program documentation, 417
- Involvement record, 35
- Involvement table, 94
- Involvement table macros, 146
- IS_DISPLAY_ONLY function, 247

K

- keys
 - composite, 346
 - primary, 346
 - schema standards, 346

L

- last page
 - in ACE reports, 381
- libentry. See Entry Library program
- linking
 - tables and fields in Entry Library programs, 236
- locating
 - field descriptions, 33
 - ID Maintenance feature, 170
 - spooler files, 309
 - tables, 33
- LOCATION attribute, 13
- LOCKMODE attribute, 13, 341
- logical index, in *mergeid*, 195

M

- macros, 127–48
 - benefits, 129
 - common, 139
 - custom, 135

- customizable, 132
- definition file macros, 223
- directory structure, 134
- file contents, 130
- file types, 132
- non-customizable, 133
- relationship to includes and C programs, 128
- sort break in ACE, 215
- standards
 - in menu options, 359
 - in menu source files, 356
 - in program screens, 364
 - in screens, 369
- user, 137
- Mail program, 324
- make
 - Makefiles, 400
- manual
 - intended audience, 1
 - purpose, 1
- margins
 - in ACE reports, 274
- Marital table, 96
- menu access, 325
- menu definition file
 - PERFORM screens, 372
- menu descriptions
 - attributes, 321
- menu option
 - runreports script, 307
- menu options
 - attributes, 314, 385
 - basic standards, 386
 - examples, 389
 - file location
 - for ACE reports, 382
 - for ACE reports, 382
 - Mail, 324
 - News, 324
 - run description scripts, 388
 - standards, 383
 - tags, 383
 - testing, 388
- menu parameter file, 323
- menu system, 311–419
- menudesc files, 320
 - creating, 322
- menuopt files, 312
 - creating, 318
 - modifying, 319
- menuparam file
 - setup, 323
- menus
 - in Database Administration, 208
 - option standards, 358
 - programming standards, 413

- source file standards, 355
- Merge ID Interactive screen, 201
- Merge ID List screen, 203
- Merge ID program
 - batch mode, 194, 206
 - criteria for columns, 193
 - description, 192
 - entering pairs, 200
 - features, 194
 - foreign key, 192
 - ID column, 193
 - interactive mode, 194, 200
 - Interactive screen, 201
 - list screen, 203
 - logical index, 195
 - merge index, 195
 - merge types, 198
 - permissions, 199
 - running audits, 194
 - tables and records, 197
 - unique index, 194
- merge index
 - in *mergeid*, 195
- Merge Table List window, 204
- Merge Type table, 197
- Military record, 35
- modifying
 - menuopt files, 319
- multi-column constraint, 19

N

- naming conventions
 - C source files, 398
 - file names, 397
 - functions, 398
 - header files, 398
 - object files, 398
 - program names, 397
 - standards, 397
 - variable standards, 398
- News program, 324
- nomenu file, 325
- nomenu_\$CARSDb file, 325
- non-fatal error messages
 - programming standards, 412
- non-fatal errors
 - in dbmake, 20
 - in program documentation, 418
 - running sortpage, 219
- numeric fields
 - in ACE reports, 380

O

- Occupation table, 98
- Office Permission table, 85

- office permissions, 85
- Office table, 100
- on every record
 - in ACE reports, 380
- on last record
 - in ACE reports, 380
- Operator Form Request screen, 166
- operators only, 81
- options
 - Database Administration menu, 208
 - in Database Administration, 210
- Order Quantity screen, 167
- Organization record, 35
- output
 - ACE reports, 381
 - definition
 - in ACE reports, 379
- output parameters
 - stored procedures, 28

P

- page headers
 - in ACE reports, 278
- page specifications
 - in ACE reports, 274
- page trailers
 - in ACE reports, 279
- parameter labels
 - in Entry Library programs, 229
- parameter types
 - in Entry Library programs, 228
- parameters
 - special functions, 251
- parameters
 - Form Entry, 164
 - section in program documentation, 417
- PERFORM
 - in database dictionary, 329
 - screen standards, 367, 370
- PERFORM screen commands, 254
- periodic macros
 - common, 141
- Permission table, 102
- permissions, 85
 - audit trails, 23
 - merge ID, 199
- Phone Call record, 35
- PREFIX attribute, 14, 342
- primary ID, for merging, 192
- print commands
 - ACE reports, 274
- print spooler
- printer macros, 146
- printers
 - spooling software, 308

- Privacy Field table, 104
- Privacy table, 104
- private
 - style in entry screens, 104
- procedures
 - section in program documentation, 417
- process
 - spooling, 308
- process to process functionality
 - in Entry Library programs, 241
- processes
 - Form Entry program, 162
- processing
 - constraints in dbmake, 20
- product differences, 1
- Profile record, 35
- program screen
 - standards, 361
- programming
 - arguments in entry programs, 397
 - arguments standards, 396
 - coding structure, 401
 - conventions, 395
 - declaration files, 399
 - definition files, 399
 - design, 395
 - ESQL standards, 396
 - function definition, 407
 - include files, 399
 - macro files, 398
 - make files, 399
 - Makefiles, 400
 - naming conventions, 397
 - portable code, 408
 - project dependent standards, 410
 - SMO standards, 327, 416
 - source files, 400
 - standards, 395
 - user interface standards, 411
 - variable definitions, 405
- programs
 - Duplicate ID Detection, 172–85
 - Entry Library, 221–52
 - Form Entry
 - parameters, 164
 - process flow, 162
 - program screens and windows, 166
 - ID Entry, 170–71
 - Schedule Entry, 213
 - sortpage, 214
- Programs
 - Form Entry, 161
- prompts
 - in dbadmin audits, 211
- punctuation
 - standards

- in menu options, 359
- in menu source files, 356
- in program screens, 363

Q

- query statements
 - in Database Administration audits, 211

R

- read statements
 - in ACE reports, 379
- records
 - Accomplishment, 34
 - Addressee, 34
 - Addressing, 34
 - Alternate Address, 34
 - Business, 34
 - Church, 34
 - common, 34–35
 - Contact, 34
 - Contact BLOB, 34
 - Contact Detail, 34
 - Contact Image, 34
 - Database Field, 34
 - Database File, 34
 - Education, 34
 - Employment, 34
 - Event, 34
 - Examination, 34
 - Faculty, 34
 - Forms Order, 35
 - Group Scheduling, 35
 - Hold, 35
 - ID, 35
 - ID Contact, 35
 - Image Document, 35
 - Interest, 35
 - Involvement, 35
 - Military, 35
 - Organization, 35
 - Phone Call, 35
 - Profile, 35
 - Relationship, 35
 - standards definition, 340
 - Step Objective, 35
 - Step Requirement, 35
 - Temporary ID Data, 35
 - Tickler, 35
- Relationship record, 35
- Relationship table, 106
- removing
 - ID records, 186
- repair statements
 - in dbadmin audits, 211
- report format

- in ACE reports, 379
 - report sorting, 302
 - reports, 269–93
 - Fields By File, 33
 - Fields By Track, 33
 - Files By Track, 33
 - idaudit, 190
 - restricting access of menu, 325
 - Right To Know macros, 146
 - ROWLIMITS attribute, 14, 342
 - rows
 - table, 31
 - running
 - ACE reports, 270
 - audits on id records, 186
 - dupid, 177
 - idaudit, 189
 - idaudit with options, 190
 - runreport script, 302, 304
 - runreports script
 - menu file, 307
- ## S
- schd_rec, 213
 - schdentry. *See* Schedule Entry program
 - Schedule Entry program, 213
 - adding records, 213
 - setup, 213
 - windows, 213
 - schema
 - composite keys, 346
 - data field section, 11, 343
 - database section, 11, 341
 - header section, 341
 - indexes, 347
 - location, 340
 - naming conventions, 340
 - primary key, 346
 - standard abbreviations, 347
 - standards, 327, 340
 - suffixes, 347
 - table names, 11, 341
 - table section, 13
 - type and length standards, 347
 - schemas
 - audit trails, 21
 - constraint analysis, 18
 - examples, 29–30
 - file structure, 10
 - specifying columns, 16
 - stored procedures, 26–28
 - table attributes, 13
 - template, 10
 - triggers, 21
 - school/community college macros, 146

SCR. See screens

screens

- attributes, 258, 260, 261, 264
- creating in entry library programs, 377
- definition files, 253, 256
- field types, 256, 257
- Form Entry, 166
- GUI attributes, 262
- instructions, 267
- Merge ID entry, 201
- Operator Form Request, 166
- Order Quantity, 167
- private style in entry screens, 104
- programming standards, 411
- standards for attribute section, 376
- standards for detail windows, 376
- standards in entry library programs, 374
- Student Form Request, 166

scripts

- audit
 - in Database Administration, 212
- runreport, 302, 304
- used in creating menu option run descriptions, 388

search fields

- in Database Administration, 208

secondary ID, for merging, 192

sections

- in ACE reports, 270
- mandatory
 - in ACE reports, 270

security

- data integrity, 333
- setup for SQL functions, 301

session/academic year macros, 147

setting

- includes, 159

setting up

- ID Entry screen, 171
- menu processing, 323

single-column constraint, 18

site macros, 147

sort clauses

- in ACE reports, 379

Sort Criteria table, 70

sorting

- capabilities in entry program detail windows, 70
- reports, 302

sortpage

- ACE output, 214
- header macro, 215
- macros in ACE reports, 214
- using UNIX sort utility, 214

sortpage program, 214

- bulk mail, 218
- commands, 218
- crash recovery, 219
- data flow diagram, 217
- define macro, 214
- non-fatal errors, 219

special flags

- in Entry Library programs, 231

special function, 250

special functions, 226

- events, 250
- example, 251

spool queues

- creating, 308

spooler files

- locations, 309

SQL

- table definition, 31

SQL functions

- security setup, 301
- troubleshooting, 301

standards

- arguments in entry programs, 397
- audit on summary fields, 396
- C source file names, 398
- coding structure, 401
- data abbreviations, 333
- data dictionary, 328
- data records, 332
- data structure, 332
- data tables, 332
- declaration files, 399
- definition files, 399
- ESQL, 396
- for ACE reports, 378
- for detail windows, 376
- for screen attribute section, 376
- function definition, 407
- function names, 398
- header files, 398
- include files, 399
- macro files, 398
- macros
 - in menu options, 359
 - in menu source files, 356
 - in program screens, 364
 - in screens, 369
- Makefiles, 400
- menu options, 358, 383, 386
- menu source files, 355
- menus, 413
- naming conventions, 397
- object file names, 398
- PERFORM screens, 367, 370
- portable code, 408
- product advisories, 416
- product issues, 416

- program arguments, 396
- program coding, 395
- program documentation, 417
- program name abbreviations, 335
- program screens, 361
- programming, 395
- project dependent, 410
- punctuation
 - in menu options, 359
 - in menu source files, 356
 - in program screens, 363
- schema, 340
- SMOs, 327, 416
- source files, 400
- status and error messages, 412
- user interface, 355, 411
- using transaction processing, 396
- variable definitions, 405
- variable naming conventions, 398
- State table, 108
- STATUS attribute, 14
- status messages
 - programming standards, 412
- Step Objective record, 35
- Step Requirement record, 35
- stopping menu access, 325
- stored procedures
 - accessing, 26
 - compile process, 28
 - input parameters, 28
 - output parameters, 28
 - privilege definitions, 27
 - schemas, 26–28
- Student Form Request screen, 166
- Subscription table, 110
- Suffix table, 112
- suffixes
 - in schema, 347
- support
 - for ACE reports, 382
- supporting
 - PERFORM screens, 372
- syntax
 - schema names, 7
- system
 - includes, 149–59
 - macros, 127–48
 - menus, 311–419

T

- table attributes
 - in schemas, 13
- table level functions
 - in Entry Library programs, 232
- table lookup
 - standards in screens, 374
- table names
 - database tables, 11, 341
 - in schema, 11, 341
- tables
 - Accomplishment, 36
 - ADR, 38
 - Alter, 9
 - Alternate Address, 40
 - Building, 44
 - Citizen, 46
 - columns, 31
 - Communication, 48
 - Configuration, 50
 - Contact, 53
 - Country, 56
 - County, 58
 - Day, 60
 - Degree, 62
 - Denomination, 64
 - Division/Department, 66
 - Entry Selection, 70
 - Ethnic, 73
 - Exam, 75
 - Facility, 77
 - Form Order, 80
 - Handicap, 83
 - Hold, 85
 - Hold Action, 85
 - ID Office Permissions, 90
 - Interest, 92
 - Involvement, 94
 - locating, 33
 - Marital, 96
 - Occupation, 98
 - Office, 100
 - Office Permission (holds), 85
 - Permission, 102
 - Privacy, 104
 - Privacy Field, 104
 - Relationship, 106
 - rows, 31
 - Sort Criteria, 70
 - specifying columns, 16
 - standards definition, 340
 - State, 108
 - Subscription, 110
 - Suffix, 112
 - Tickler, 114
 - Title, 116
 - Veteran Chapter, 120
 - Zip Code, 124
- Tables
 - Merge ID, 197
- telephone number macros, 147
- Temporary ID Data record, 35

- testing
 - ACE reports, 382
 - menu options, 388
 - PERFORM screens, 372
 - program standards, 395
- TEXT attribute, 15
- Tickler record, 35
- Tickler table, 114
- Title table, 116
- TRACK attribute, 15
- track macros, 148
- transaction procedures
 - in Entry Library programs, 252
- translating
 - ACE reports, 381
- triggers, 21
 - categories of, 22
 - examples, 25
 - syntax in dbmake, 22
- triggers/audit trials
 - implementing, 23–24
- troubleshooting
 - acearray functions, 291
 - SQL functions, 301

U

- unavailable features. See product differences
- unique indexes
 - in dbmake, 20
 - in *mergeid*, 194

UNIX

- table names, 7
- UNIX sort utility, 214
- user interface
 - standards, 355
- user macros, 137
- using
 - acearray functions, 284
 - dbadmin options, 207
 - dupid in interactive mode, 179
 - dupid in review mode, 183
 - EdVanta System print spooler, 269, 308
 - idaudit options, 187
 - programming standards, 395
 - transaction processing, 396

V

- variables
 - ACE reports, 273
 - acearray functions, 284
- Veteran Chapter table, 120

W

- windows
 - Alternate Recipient window, 166
 - Form Order History, 166
- word processing macros, 148

Z

- Zip Code table, 124