Peregrine

ServiceCenter

System Tailoring, Volume 3

Release 5.1

PART NO: SCT-5.1-ENG-01036-00233

Peregrine SYSTEMS

This edition applies to version 5.1 of the licensed program.

Peregrine Systems, Inc.
Worldwide Corporate Headquarters
3611 Valley Centre Drive San Diego, CA 92130
Tel 800.638.5231 or 858.481.5000
Fax 858.481.1751
www.peregrine.com

Contents

Getting Started

The ServiceCenter System Tailoring Guides have supplemental information for system administrators who install and configure ServiceCenter. Tailoring is any change to standard functionality without changing actual code. For example, you can:

- Change the look and operation of forms.
- Change default values for objects on forms that ServiceCenter uses for field validation.
- Create macros, scripts, and stored queries.
- Changes to record definitions.

Use these guides to make further changes to support site-specific requirements, including special field validation, new or modified forms design, expanded or varied workflow, and automatic notifications.

Tailoring ServiceCenter

Most tailoring can be done using high-level ServiceCenter tools, without directly changing the RAD code that is the actual ServiceCenter development medium. There are several tailoring tools, including the Database Manager, Format Control Editor, Link Editor, and Revision Control that enable you to manipulate the common RAD code sets and algorithms in the document engine. Because these tools enable extensive changes to ServiceCenter, Peregrine recommends that you analyze your requirements carefully before you begin tailoring implementation. Balance the gains of tailoring against simplifying future upgrades to new releases.

Using the System Tailoring Guides

System tailoring information appears in three separate guides. The following table shows the focus of each guide and where you should look for more information.

System Tailoring Guide, Volume 1	System Tailoring Guide, Volume 2	System Tailoring Guide, Volume 3
■ Forms Designer	■ Document engine overview	■ Incident management
■ Format Control	■ Validity table processing	■ Stored queries
■ Array maintenance	■ The notification engine	■ Sequential numbers
■ Special processing considerations	■ Global lists and Global initer	■ Scripting
■ Sequential numbering for Format Control	■ Using and creating online help	■ Plug-in support
■ Format Control processes	■ The Cascade Update utility	■ Creating wizards
■ Format Control posting	■ The display application	■ Creating and editing macros
■ Format Control error messages	■ Advanced operations	■ Development audit utility
■ Format Control common applications	■ Adding and modifying fields	■ Revision control
■ Publishing ServiceCenter information	■ Calendar management	■ DDE support
■ Static messages		■ Data Policy
■ System management		■ Clocks
■ ServiceInfo forms		■ System language: data types, variables, operators, expressions
		■ ServiceCenter default variables
		■ Link management
		■ Virtual joins

You can use the Acrobat Search feature to locate more specific topics on the ServiceCenter 5.1 documentation CD-ROM.

Knowledge Requirements

The instructions in this guide assume a working knowledge of Peregrine Systems ServiceCenter and the installation platform. You can find more information in the following guides.

- For information about a particular platform, see the appropriate platform documentation.
- For information about customizing your environment using parameters, see the *ServiceCenter Technical Reference* guide.
- Before you run the ServiceCenter server, see the *ServiceCenter User's Guide*.
- For administration and configuration information, see the *ServiceCenter System Administrator's Guide* or the *ServiceCenter Application Administration Guide.*
- For database configuration information, see the *ServiceCenter Database Management and Administration Guide.*
- For copies of the guides, download PDF versions from the CenterPoint web site using the Adobe Acrobat Reader, which is also available on the CenterPoint Web Site. For more information, see *Peregrine's CenterPoint Web Site* on page 14. You can also order printed copies of the documentation through your Peregrine Systems sales representative.

Examples

The sample windows and the examples included in this guide are for illustration only, and may differ from those at your site.

Contacting Customer Support

For more information and help with this new release or with ServiceCenter in general, contact Peregrine Systems' Customer Support.

Peregrine's CenterPoint Web Site

You can also find information about version compatibility, hardware and software requirements, and other configuration issues at Peregrine's Centerpoint web site: *http://support.peregrine.com*

1 Log in with your login ID and password.

2 Select **Go** for **CenterPoint**.

3 Select **ServiceCenter** from **My Products** at the top of the page for configuration and compatibility information.

Note: For information about local support offices, select **Whom Do I Call?** from **Contents** on the left side of the page to display the **Peregrine Worldwide Contact Information**.

Corporate Headquarters

Address:	Peregrine Systems, Inc. Attn: Customer Support 3611 Valley Centre Drive San Diego, CA 92130
Telephone:	+(1) (858) 794-7428
Fax:	+(1) (858) 480-3928

North America and South America

Telephone:	+(1) (858) 794-7428 (Mexico, Central America, and South America)
Fax:	+(1) (858) 480-3928
E-mail:	support@peregrine.com

Europe, Asia/Pacific, Africa

For information about local offices, see *Peregrine's CenterPoint Web Site*. You can also contact *Corporate Headquarters*.

Contacting Education Services

Training services are available for the full spectrum of Peregrine Products including ServiceCenter.

Current details of our training services are available through the following main contacts or at:

http://www.peregrine.com/education

Address:	Peregrine Systems, Inc. Attn: Education Services 3611 Valley Centre Drive San Diego, CA 92130
Telephone:	+1 (858) 794-5009
Fax:	+1 (858) 480-3928

1 CHAPTER Incident Management Structure

Problem and Probsummary Records

The Incident Management system stores information about incidents in two separate files: problem and probsummary. Each incident has only one probsummary record. Every time an incident is updated, however, the system automatically adds a new record to the problem file. This assumes that paging is on. If it is not, then there will be only one problem page.

Before an update

problem	probsummary
incident 43, Page 3, Last page=true	incident 43
incident 43, Page 2, Last page=false	
incident 43, Page 1, Last page=false	

After an update

problem	probsummary
incident 43, Page 4, Last page=true	incident 43
incident 43, Page 3, Last page=false	
incident 43, Page 2, Last page=false	
incident 43, Page 1, Last page=false	

Note: Italics indicate a new record.

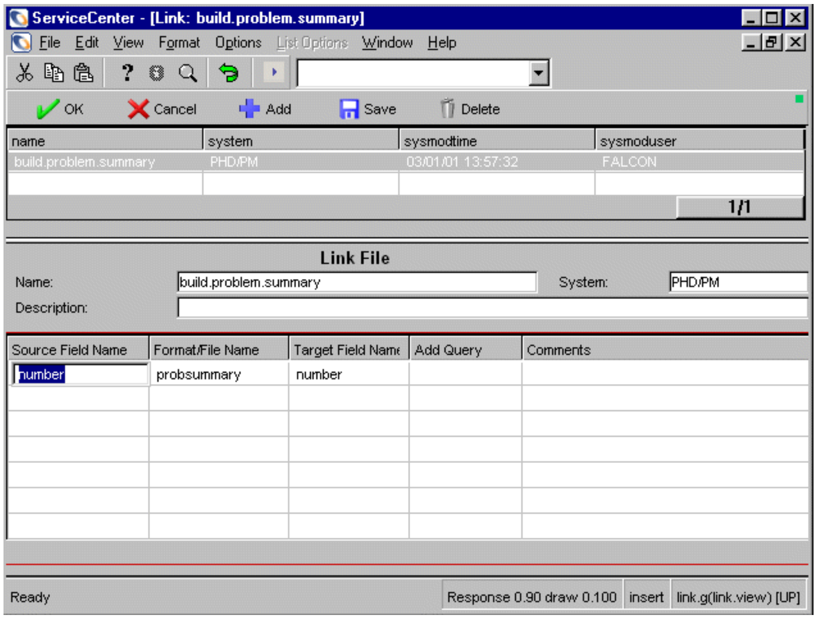
Linking the problem and probsummary files

The system uses a link record called build.problem.summary to determine which fields are copied from the problem file into the probsummary file. The link relation is based on the number field in the problem and probsummary files.

To display the problem and probsummary link definitions:

1. Select the **Services** tab of System Administrator's Home Menu.
2. Click **Incident Management**. The **Incident Management** menu is displayed.
3. Click **Security Files.** The Incident Management Security Administration Utility is displayed.

4 Click **Summary Link**. The build.problem.summary link file is displayed.


ServiceCenter - [Link: build.problem.summary]
File Edit View Format Options List Options Window Help
OK Cancel Add Save Delete
Link File
Ready

name	system	sysmodtime	sysmoduser
build.problem.summary	PHD/PM	03/01/01 13:57:32	FALCON

1/1

Name: build.problem.summary System: PHD/PM
Description:

Source Field Name	Format/File Name	Target Field Name	Add Query	Comments
number	probsummary	number		

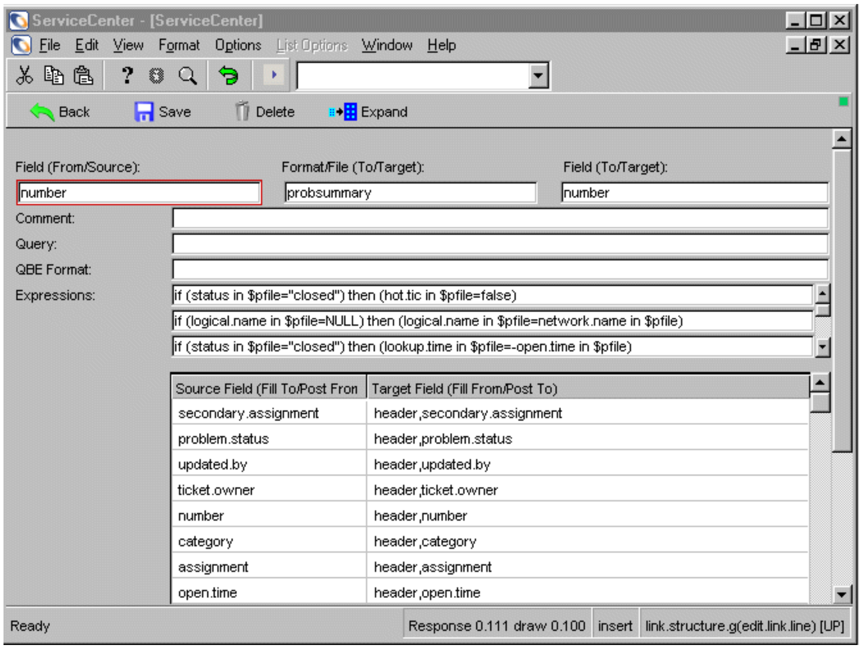
Response 0.90 draw 0.100 insert link.g(link.view) [UP]

Figure 1-1: Summary Link File for Incident Management

5 Place the cursor in any field of the Link record, for example, number.

6 Select **Select Line** from the **Options** menu.

The Link Line Definition form displays the links between the individual fields in the **problem** file and the fields available in the **probsummary** file.


ServiceCenter - [ServiceCenter]
File Edit View Format Options List Options Window Help
Back Save Delete Expand

Field (From/Source): number

Format/File (To/Target): probsummary

Field (To/Target): number

Comment:

Query:

QBE Format:

Expressions:

```
if (status in $pfile="closed") then (hot.tic in $pfile=false)
if (logical.name in $pfile=NULL) then (logical.name in $pfile=network.name in $pfile)
if (status in $pfile="closed") then (lookup.time in $pfile=-open.time in $pfile)
```

Source Field (Fill To/Post Fron	Target Field (Fill From/Post To)
secondary.assignment	header,secondary.assignment
problem.status	header,problem.status
updated.by	header,updated.by
ticket.owner	header,ticket.owner
number	header,number
category	header,category
assignment	header,assignment
open.time	header,open.time

Ready | Response 0.111 draw 0.100 | insert | link.structure.g(edit.link.line) [UP]

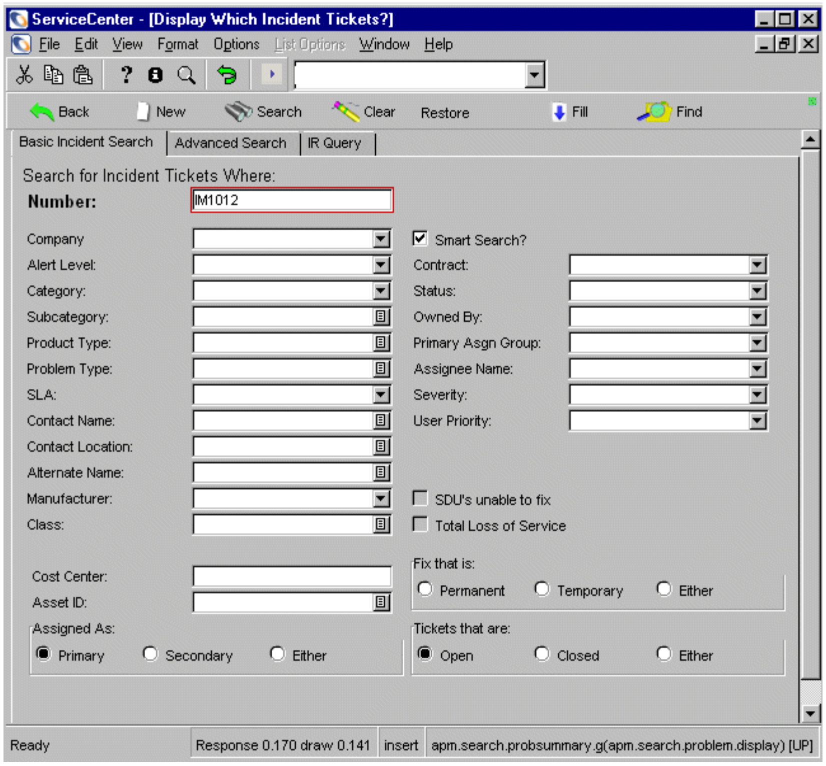
Figure 1-2: Incident Link Line Definition

Incident Management's built in search routines search against the **probsummary** file rather than the **problem** file. They then display a list of matching values from the **probsummary** file in either a table (GUI mode), or a QBE list (Text mode). Whenever a user selects an incident ticket, the system displays the last page of the incident record from the **problem** file.

Searching the probsummary file

To search the probsummary file:

1 Click the **Search** button in the Incident Management menu. The probsummary search form is displayed.


ServiceCenter - [Display Which Incident Tickets?]
File Edit View Format Options List Options Window Help
Back New Search Clear Restore Fill Find
Basic Incident Search | Advanced Search | IR Query

Search for Incident Tickets Where:

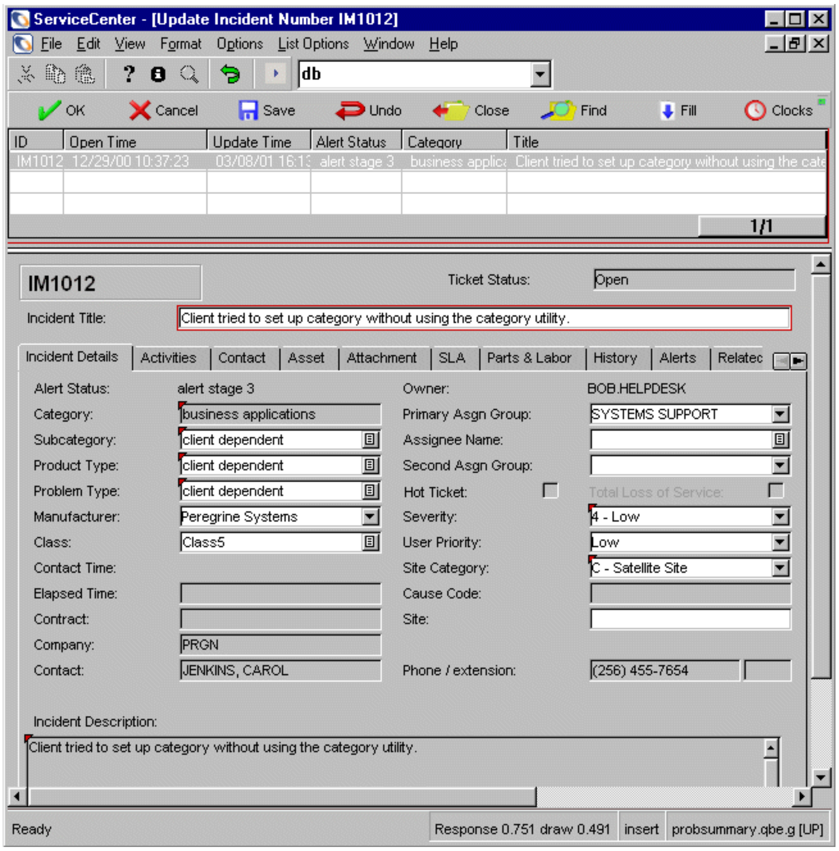
Number:	IM1012		
Company:		[✓] Smart Search?	
Alert Level:		Contract:	
Category:		Status:	
Subcategory:		Owned By:	
Product Type:		Primary Asgn Group:	
Problem Type:		Assignee Name:	
SLA:		Severity:	
Contact Name:		User Priority:	
Contact Location:			
Alternate Name:			
Manufacturer:		[ ] SDU's unable to fix	
Class:		[ ] Total Loss of Service	
Cost Center:		Fix that is:	Permanent Temporary Either
Asset ID:			
Assigned As:	[✓ Primary] Secondary Either	Tickets that are:	[✓ Open] Closed Either

Ready | Response 0.170 draw 0.141 | insert | apm.search.probsummary.g(apm.search.problem.display) [UP]

Figure 1-3: Probsummary Search Form

Important: There are two places to select ticket Status in this form: the Status combo box and the Open, Closed and Either radio buttons. If a query fails to produce existing records, make sure the Status field and the Status flag reflect the same ticket status.

2 Enter a search query (for example, IM1012) to display matching **probsummary** records or leave the Number field blank and click the **Search** button to display a list of all records.


ServiceCenter - [Update Incident Number IM1012]

ID	Open Time	Update Time	Alert Status	Category	Title
IM1012	12/29/00 10:37:23	03/08/01 16:13	alert stage 3	business applic	Client tried to set up category without using the cate

IM1012 Ticket Status: Open

Incident Title: Client tried to set up category without using the category utility.

Alert Status:	alert stage 3	Owner:	BOB.HELPDESK
Category:	business applications	Primary Asgn Group:	SYSTEMS SUPPORT
Subcategory:	client dependent	Assignee Name:	
Product Type:	client dependent	Second Asgn Group:	
Problem Type:	client dependent	Hot Ticket:	Total Loss of Service:
Manufacturer:	Peregrine Systems	Severity:	4 - Low
Class:	Class5	User Priority:	Low
Contact Time:		Site Category:	C - Satellite Site
Elapsed Time:		Cause Code:	
Contract:		Site:	
Company:	PRGN		
Contact:	JENKINS, CAROL	Phone / extension:	(256) 455-7654

Incident Description:
Client tried to set up category without using the category utility.

Ready | Response 0.751 draw 0.491 | insert | probsummary.qbe.g [UP]

Figure 1-4: Probsummary Record List

3 Select a specific record from the list.

The system retrieves the selected record from the **S** file.

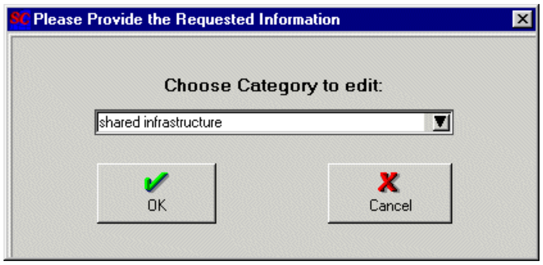
Categories and Forms

Each incident ticket in the system can have only one category. This category determines what type of information is stored with an incident ticket. For example, a PC hardware incident will track different information than one dealing with a network outage.

All incident records, regardless of category, store their data in the `problem` file. Categories merely provide different views of the `problem` file, despite the fact that records for all categories are kept in the same file.

To access a Category record:

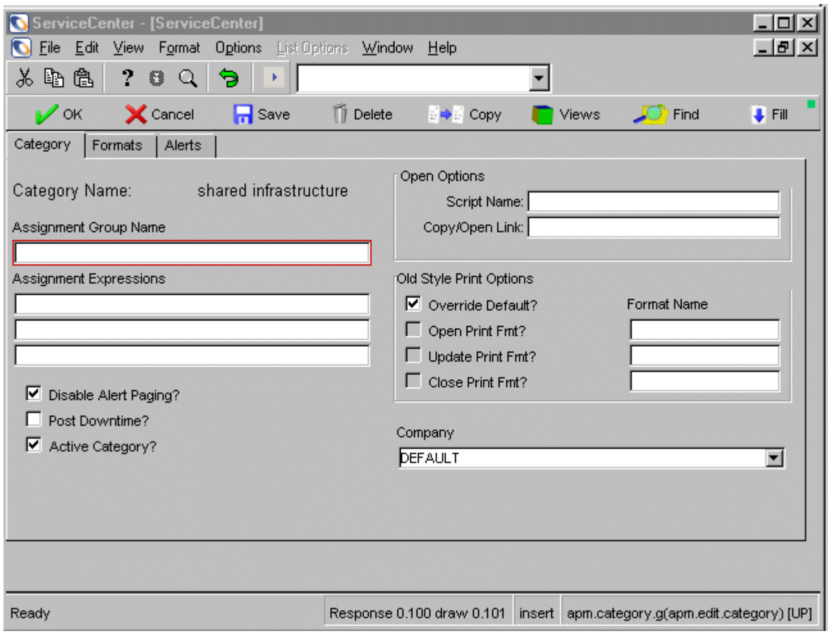
1. Select the **Services** tab of System Administrator's Home Menu.
2. Click **Incident Management.** The Incident Management menu is displayed.
3. Click **Security Files.** The Incident Management Security Administration Utility is displayed.
4. Click **Edit** in the Category structure. A dialog box is displayed asking you to select the category you want to edit.


Please Provide the Requested Information

Choose Category to edit:

shared infrastructure

5 Select a category from the drop-down list. The requested record is displayed.


ServiceCenter - [ServiceCenter]
File Edit View Format Options List Options Window Help
OK Cancel Save Delete Copy Views Find Fill
Category Formats Alerts

Category Name: shared infrastructure

Assignment Group Name

Assignment Expressions

☑ Disable Alert Paging?
☐ Post Downtime?
☑ Active Category?

Open Options
Script Name:
Copy/Open Link:

Old Style Print Options
☑ Override Default?
☐ Open Print Fmt?
☐ Update Print Fmt?
☐ Close Print Fmt?
Format Name

Company
DEFAULT

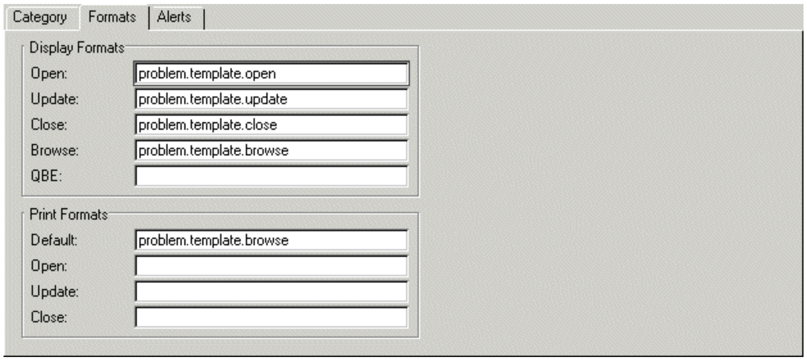
Ready | Response 0.100 draw 0.101 | insert | apm.category.g(apm.edit.category) [UP]

Figure 1-5: Category Record— Category Tab

Formats

Each category defines five different forms in its Category record.

To view the Forms in a Category record, click the **Formats** tab.



Display Formats

Open:	problem.template.open
Update:	problem.template.update
Close:	problem.template.close
Browse:	problem.template.browse
QBE:	

Print Formats

Default:	problem.template.browse
Open:	
Update:	
Close:	

Figure 1-6: Category Record — Formats Tab

These forms are described below:

Form Type	Usage
Open	When users open a new incident ticket in this category, they enter data in this form.
Update	When users update an existing incident ticket in this category, they enter data in this form.
Close	When users close an existing incident ticket, this form is displayed.
Browse	When text mode users first select an incident ticket from a Record list, they are shown this format. When they press **update**, the update format is displayed.
QBE	When users search for a form, they are shown this format.

Typically, forms used in this fashion are named problem.<category>.<action>. For example, problem.template.open.

The following are the default forms:

Format Type	Format Name
Open	problem.template.open
Update	problem.template.update
Close	problem.template.close
Browse	problem.template.browse (or problem.browse)
Qbe	problem.template.qbe

Workflow

There are several possible workflows for Incident Tickets. A common one is described by the following diagram.

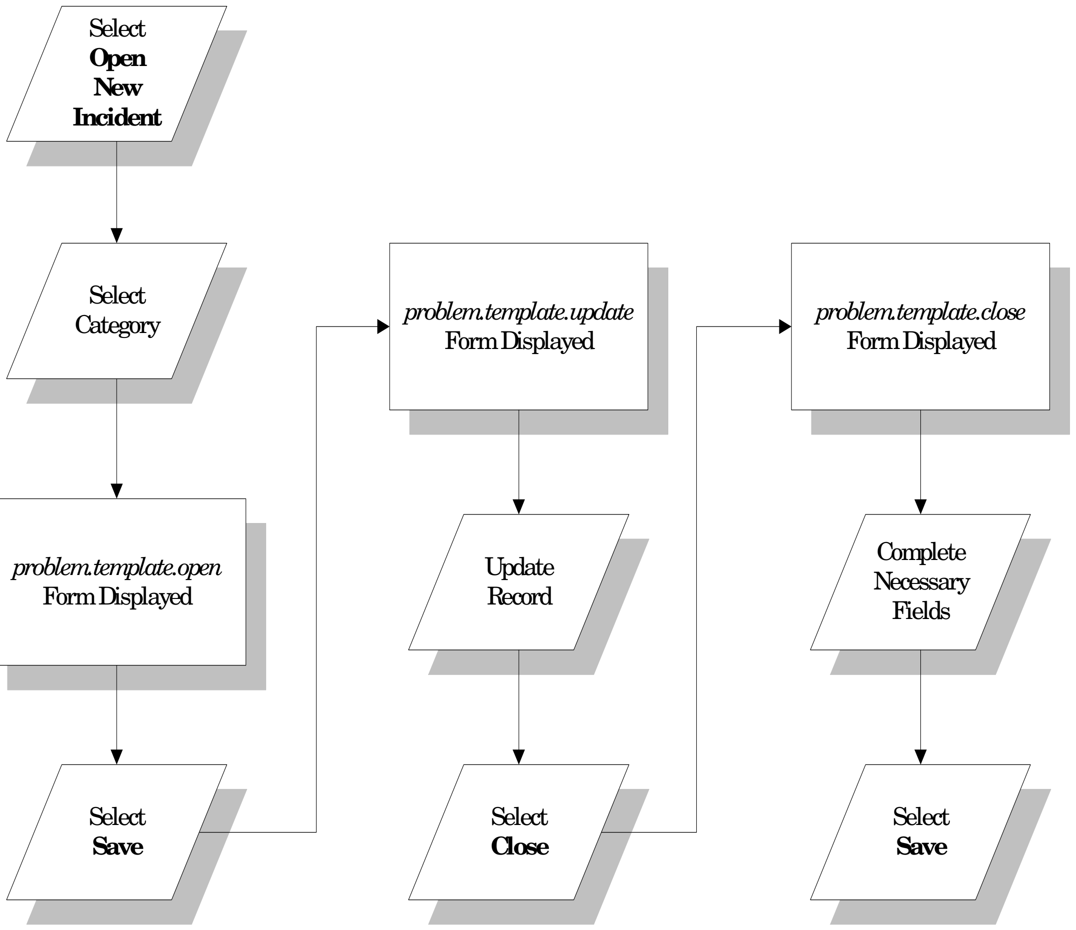

Select **Open New Incident**
Select Category
problem.template.open Form Displayed
Select **Save**
problem.template.update Form Displayed
Update Record
Select **Close**
problem.template.close Form Displayed
Complete Necessary Fields
Select **Save**

Figure 1-7: Life of an Incident ticket opened in Service Management.

Form naming conventions

ServiceCenter uses separate forms for Text mode, GUI mode, and Web mode and tracks them by assigning different one-letter extensions to the base form name.

- problem.template.close is the base form name and is used in Text mode.
- problem.template.close.g is the GUI/Java form and is used instead of problem.template.close in GUI and Java modes
- problem.template.close.w is the HTML form and is used instead of problem.template.close in the Web mode.

There are two important considerations regarding ServiceCenter naming conventions:

- Always use the base form name when entering a form into the system. The ServiceCenter binaries will automatically look for a form with the appropriate extension if running in Web or GUI mode.
- You do not need to define all three forms. If the system cannot locate a GUI specific, or Web specific form, it will use the Text mode format instead.

The following table describes the system display logic for a form named hardware:

Text	GUI/Java	Web/HTML
uses **hardware**	if **hardware.g** exists, uses **hardware.g** else uses **hardware**	if **hardware.w** exists, uses **hardware.w** else uses **hardware**

Assignment Groups

Incident Tickets are assigned to various assignment groups to be resolved. Each incident ticket can have only one primary assignment group, but may have as many secondary assignment groups as necessary.

Assignment groups are defined in two places:

- In the Category record
- On individual tickets

Note: Assignment group designations on individual incident tickets supersede those defined in the Category record.

Assignment groups have two main functions:

- Determine ticket visibility in user inboxes.
- Help determine what happens to incident tickets when their alert levels are escalated.

When a user opens a default inbox in Incident Management, a list of tickets is displayed for the assignment group(s) of which that user is a member. For example, if a user is a member of the systems and engineering assignment groups, all open tickets listing systems or engineering as the primary or secondary assignment groups will appear in that user's inbox.

Example

The table below shows the Assignment Group associations for seven sample incident tickets:

Incident #	Primary Assign.	Secondary Assign.	Secondary Assign.
1	systems		
2	management	engineering	
3	development	pc hardware	
4	systems		
5	engineering		
6	pc hardware	systems	
7	pc software	engineering	systems

The members of the two Assignment Groups—systems and engineering — are defined as shown below:

systems	engineering
Alan	Bob
Bob	Sam

Ed	Terry
Jim	
Rita	

The table below shows which tickets will be visible to each member of the two Assignment Groups:

User Name	Visible Tickets
Alan	1,4,6,7
Bob	1,2,4,5,6,7
Ed	1,4,6,7
Jim	1,4,6,7
Rita	1,4,6,7
Sam	2,5,7
Terry	2,5,7

Security Model

The Incident Management system has only one environment record. This record contains options that affect all users of Incident Management. Typical options stored here are:

- The default category for new incident tickets.
- Whether or not to use paging with Incident Management.
- Whether or not to enable distributed ticketing.

Users

Each person who logs onto ServiceCenter is a user. Each user must have a personal information record stored in the operator table. Information associated with a user includes personal data such as name, address, phone number(s), login name, and password for ServiceCenter. Operator records also store capability words for a given user. Without an operator record, no user can log onto ServiceCenter.

Profiles

Each user of Incident Management must have an Incident Management profile record or use the default profile. User profiles are stored in the pmenv table. They can be defined in one of two ways:

- A personal profile record which defines rights for a single user.
- A profile record defining members of a profile group.

Profiles store Incident Management rights and privilege information, such as whether or not a user can close incident tickets. Profiles also store information that may affect the way Incident Management looks and behaves. For example, a profile may list a personal search form for a specific user.

Profile groups

Profile groups store lists of users who share a common Incident Management profile. Each group has a distinct record in the group table as well as one profile record stored in the pmenv table. A member of a group will use the group's profile record unless that user also has a personal profile record.

Capability words

The following capability words are used to grant privileges in Incident Management:

- SysAdmin — grants the user System Administrator authority with the right to run administrative utilities for all ServiceCenter applications.
- ProbAdmin — grants the user administrative status within Incident Management only. For example, a user with ProbAdmin capabilities cannot alter Change Management profiles.
- problem management — grants the user access to Incident Management from the menu screens.

For a complete of ServiceCenter capability words, see the System Administrator's guide.

How the system selects a profile

When a user tries to launch Incident Management, the system follows these steps to determine which profile to use:

1. The system looks for a profile record in the pmenv table whose name matches the user's login name. For example, if John Doe logs into ServiceCenter as jdoe, Incident Management will look for a profile record named jdoe. If such a profile is found, the system defines the user's Incident Management rights based on this personal profile.
2. If the system cannot locate a personal profile record for a user, it next looks for a profile group which has the user as a member. If the user is a member of a group, the system locates the profile record for that group. The profile record carries the same name as the group itself. Once the system finds the group profile record, the user is granted the rights defined by that record. Therefore, if jdoe is a member of the Engineering group, the system will locate a profile record named Engineering, and use that profile record to determine John Doe's rights.
3. If the system finds neither an individual profile nor a group profile, it looks in the Incident Management Environment record to see if the Allow Access Without Operator Record? check box is selected. This option allows access through the system's DEFAULT profile record. For more information on Profiles, refer to the ServiceCenter System Administrator's Guide.

Alerts

Incident Tickets are automatically escalated to different alert levels as a function of the category of the incident ticket in question and its primary assignment group. The category record determines when alerts and escalations occur. The assignment record determines what happens when an alert occurs.

Two different clocks manage alerts and escalations:

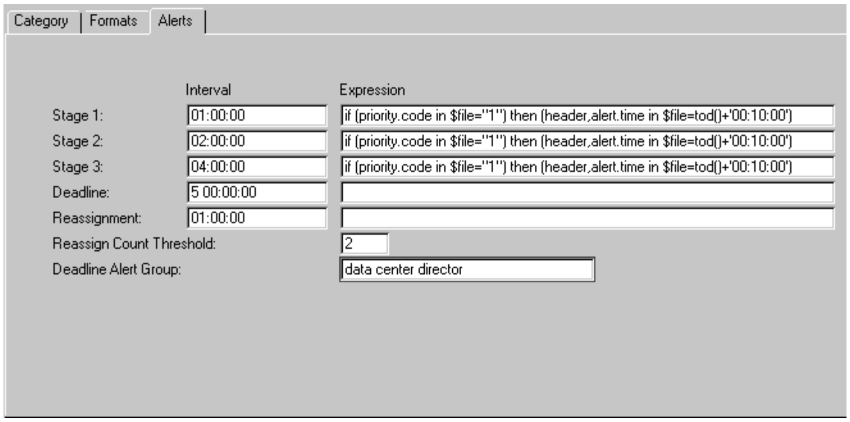
- The deadline alert clock starts the moment the incident ticket is opened. When a specified amount of time has passed on this clock, the incident ticket is escalated to DEADLINE ALERT, regardless of any recent update activity.

- The second escalation clock also begins ticking when an incident ticket is opened. When it reaches certain thresholds, the incident ticket is escalated to a higher alert stage — alert stage 1, alert stage 2, or alert stage 3. Unlike the deadline alert clock, this clock resets whenever the incident ticket is updated by a user.

To view the alerts for a Category:

1 Open the **Category** record as described *Categories and Forms* on page 23.

2 Click the **Alerts** tab to view the alerts.



Category | Formats | Alerts

	Interval	Expression
Stage 1:	01:00:00	if (priority.code in $file="1") then (header,alert.time in $file=tod()+'00:10:00')
Stage 2:	02:00:00	if (priority.code in $file="1") then (header,alert.time in $file=tod()+'00:10:00')
Stage 3:	04:00:00	if (priority.code in $file="1") then (header,alert.time in $file=tod()+'00:10:00')
Deadline:	5 00:00:00	
Reassignment:	01:00:00	
Reassign Count Threshold:		2
Deadline Alert Group:		data center director

Figure 1-8: Category Record— Alerts Tab

The numbers in the Interval fields dictate the amount of time that each of the clocks will run. The deadline interval of 5 00:00:00 indicates that the deadline alert clock will run for 5 days before it moves the ticket into DEADLINE ALERT. The stage 1 alert interval of 01:00:00 indicates that the incident ticket will move into alert stage 1 one hour after its last user update.

The alert clocks are processed as follows:

Time	Activity	Incident Alert Status
08/19/01 12:30:00	User Opens Incident	Opened
08/19/01 13:30:00	Incident Automatically Escalated to Alert Stage 1	Alert stage 1
08/19/01 14:30:00	Incident Automatically Escalated to Alert Stage 2	Alert stage 2

08/19/01 15:00:00	User Updates Incident	Updated
08/19/01 16:00:00	Incident Automatically Escalated to Alert Stage 1	Alert Stage 1
08/19/01 17:00:00	Incident Automatically Escalated to Alert Stage 2	Alert stage 2
08/19/01 19:00:00	Incident Automatically Escalated to Alert Stage 3	Alert stage 3
08/24/01 12:30:00	Incident Automatically Escalated to Deadline Alert	DEADLINE ALERT
08/24/01 12:45:00	User Updates Ticket	DEADLINE ALERT
08/24/01 12:45:00	User Updates Ticket	DEADLINE ALERT

Alerts & calendars

The clocks that manage alerts do not need to run on a 24 hour schedule. For example, if your employees work from 9 am to 5 pm, set the alert clocks to run only during these hours. By default, all alert clocks run on a twenty-four hour, seven-days-a-week schedule; however, if an availability calendar is selected for a particular incident ticket's primary assignment group, that ticket's alert clocks will run only during the duty hours defined by that calendar.

Alert expressions

You are not limited to driving alerts from category information only. You can also set escalation times based on any field or combination of fields in the incident ticket by using expressions.

The alert expressions should achieve the objective of establishing a value for the alert.time field (alert stages 1, 2 and 3), or the deadline.alert field (deadline alert). The resulting value calculated for this field will be used to determine a time interval between an existing reference point in time, and this value. ServiceCenter uses specific reference points in time depending on the action being taken against the ticket. When a ticket is open, the reference point is the open time of the ticket. When updating a ticket, the reference point becomes the update time of the ticket.

If the assignment group indicated on the ticket has an associated calendar, ServiceCenter will use the interval of time calculated above to determine the final alert time for the ticket. Because of this, it is important that the alert expressions are not built around reference points in time other than what ServiceCenter is designed to use (open time or update time). The most basic alert expressions will be of the following form:

Alert stages 1, 2, and 3:

alert.time in $file = tod() + <time interval>

Deadline alert:

deadline.alert in $file = tod() + <time interval>

Any valid field or combination of fields in the incident ticket can be used to establish the alert times. For example:

if priority.code in $file="1" then alert.time in $file = tod() + '01:00:00'

The following expression would yield undesirable results since it attempts to set the alert time based on a reference point outside of what ServiceCenter is designed to use (open or update time):

alert.time in $file = <custom.date.field> in $file + <time interval>

Alert notifications

Alert notifications are handled by macros. Incident Tickets are escalated according to the values in the Interval field. For further information on alerts, priorities, and escalation, refer to the ServiceCenter System Administrator's Guide.

2 Stored Queries

CHAPTER

The purpose of a *stored query* is to retrieve and display current information efficiently by using predefined search parameters. The Stored Query Maintenance utility allows designated users to define and store queries that can display lists of specific records or populate dynamic display objects such as charts and marquees. For example, stored queries can be created to search for incident tickets that have reached a certain status, to populate a chart that displays open tickets by category, or to display a list of change requests assigned to a particular approval group. Stored queries are commonly run from the following features in ServiceCenter:

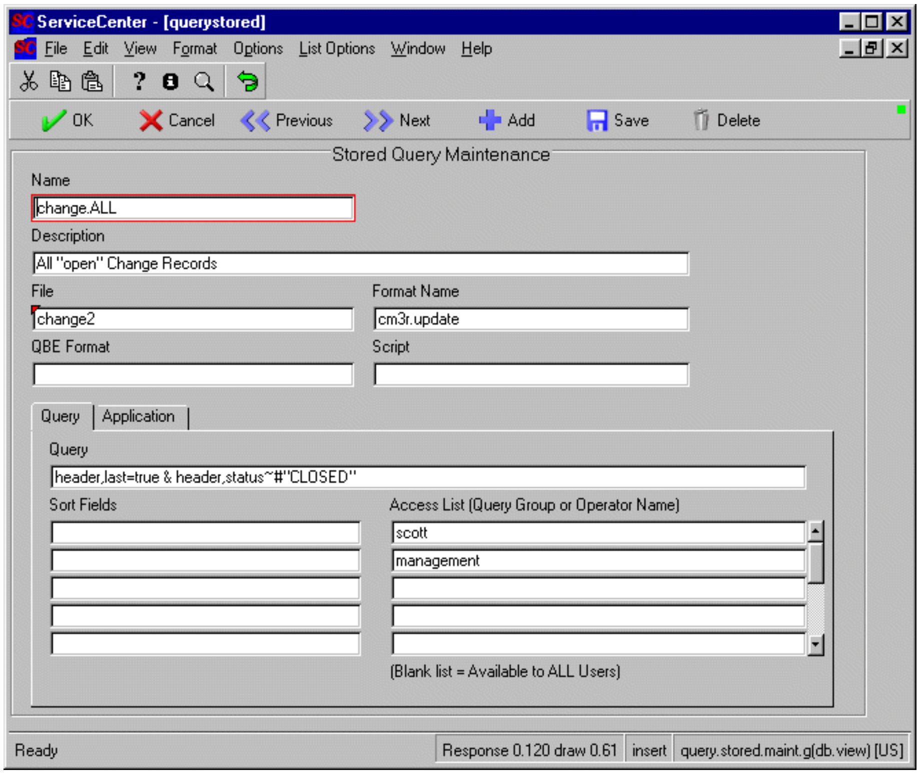
- Advanced/Expert Search menu option
- Display objects
- Buttons
- Scripts

Accessing Stored Queries

To open the Stored Queries maintenance form

1. From the System administrator's home menu, click the **Utilities** tab.
2. Click the **Tools** tab.
3. Click **Stored Queries**.

Stored Query Maintenance Form


ServiceCenter - [querystored]
File Edit View Format Options List Options Window Help
OK
Cancel
Previous
Next
Add
Save
Delete

Stored Query Maintenance

Name: change.ALL

Description: All "open" Change Records

File: change2

Format Name: cm3r.update

QBE Format:

Script:

Query | Application

Query: header,last=true & header,status~#"CLOSED"

Sort Fields:

Access List (Query Group or Operator Name): scott, management

(Blank list = Available to ALL Users)

Ready | Response 0.120 draw 0.61 | insert | query.stored.maint.g(db.view) [US]

Figure 2-1: The change.ALL Stored Query Maintenance form

Fields

Name—unique name of the query. This name can be anything, but should reflect the query's purpose and be easily recognizable. For example, queries concerning change requests might begin with *cm3r* and those searching for problem tickets might begin with *pm*.

Description—plain text description of the query's function. This is not required and is not used anywhere else in ServiceCenter.

File—name of the ServiceCenter file that this query should search.

Format Name—name of the form used to display the records retrieved by the stored query. This is a required field unless the following conditions exist:

- The **File** field contains certain values, including problem, probsummary, cm3r, cm3t, device, or incidents.
- An application is specified.
- A format exists with the same name as the **File** field.

QBE Format—name of a custom QBE form you have created for displaying the data returned from your search. This is an optional field. If you do not specify a form, the system displays the data you have requested in a default QBE form.

Script—name of a script to execute when this stored query is selected. This is an optional field. You can use scripting to retrieve data you want to incorporate into the stored query.

Query Tab

Query—query to be executed when this stored query is selected by the user. To ensure an efficient search with a stored query, all query syntax should be fully keyed. The following is an example of a query against the cm3r file that returns all of a user's open change requests:

```
header,requested.by=$lo.ufname & header,last=t & header,open=t
```

Queries can refer to $ variables which can then be defined by the user at run time with the execution of a script, as in the following example:

```
header,category=$category and header,last=true and
header,status^#"closed"
```

A script is executed that displays a form containing an input field for the variable *$category*. If the user enters **software** in the field, the following query is executed:

```
header,category="software" and header,last=true and
header,status^#"closed"
```

The $ variable is replaced with the value of the variable.

Important: If $ variables are used in a query, they must be initialized to match the data type of the field referenced in the query. Initialize variables by defining a Display condition in Format Control for the script form. Character data types do not need to be initialized.

Sort Fields—controls the key used for the query if more than one Database Dictionary key starts with the same field. This is an optional field.

Access List—defines users who can see this particular query. Only those operators listed by login name or those users who are members of a Query Group (from the operator record) named in the list can access this query. If the list is blank, this query is available to all users. This is an optional field. Use the Access List to:

- Build stored queries that are only available to certain users.
- Keep the number of stored queries a user sees at a manageable level. Define a Query Group in the operator record of each member of a user class (Tech Level 1, Tech Level 2, etc.) and give them access rights only to the queries they require to do their job.

Application Tab

Application—name of the application to call from this query. This is an optional field.

Parameter Names—name of the parameters to pass to the application called by the query. This is an optional field.

Parameter Values—values of the parameters to pass to the application called by the query. This is an optional field.

Creating Stored Queries

To create a stored query:

1 Select the **Utilities** tab in the system administrator's home menu.

2 Click **Tools.**

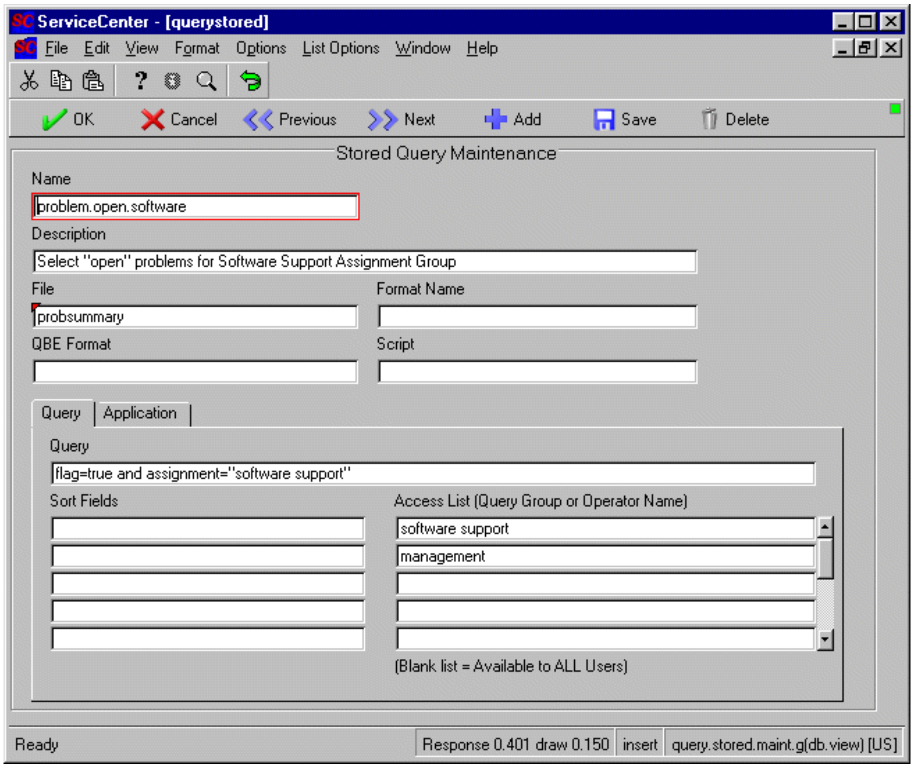
The Tools menu is displayed.

3 Click **Stored Queries.**

A blank Stored Query Maintenance form is displayed.

4 Click **Search** to display a list of existing queries to copy or create your own query from scratch.

5 Click **Add.**
The following message is displayed in the status bar: *Record added to querystored file.*


ServiceCenter - [querystored]
File Edit View Format Options List Options Window Help
OK
Cancel
Previous
Next
Add
Save
Delete

Stored Query Maintenance

Name: problem.open.software

Description: Select "open" problems for Software Support Assignment Group

File: probsummary

Format Name:

QBE Format:

Script:

Query | Application

Query: flag=true and assignment="software support"

Sort Fields:

Access List (Query Group or Operator Name): software support, management

(Blank list = Available to ALL Users)

Ready | Response 0.401 draw 0.150 | insert | query.stored.maint.g(db.view) [US]

The shown query displays all open incident tickets for the software support assignment group. This query is only available to users who are members of the query groups called *software support* and *management*.

For this query to be accessible to users who must use it, *software support* or *management* must be entered in the Query Groups array in the Startup tab of each designated user's operator record.

Figure 2-2: Stored query for category tickets

Running Stored Queries

You can use several ServiceCenter features to run stored queries. Access to the **querystored** file and the ability to edit existing stored queries or add new stored queries is granted through capability words in a user's operator file. Users with these capabilities can directly select or manipulate stored queries with the Advanced or Expert Search menu options. All other features use stored queries to present records only.

Display objects

Stored queries can be used by the system administrator to retrieve and display dynamic data in charts or marquees or to write a script that employs a stored query to locate specific records. Stored queries applied in this manner are not accessible to the user and operate in the background to retrieve records from the database. For example, you may want to place a dynamic chart on a supervisor's startup menu showing all open tickets by category. By placing buttons that run individual stored queries on the bottom of the chart, the supervisor can display lists of tickets by category.

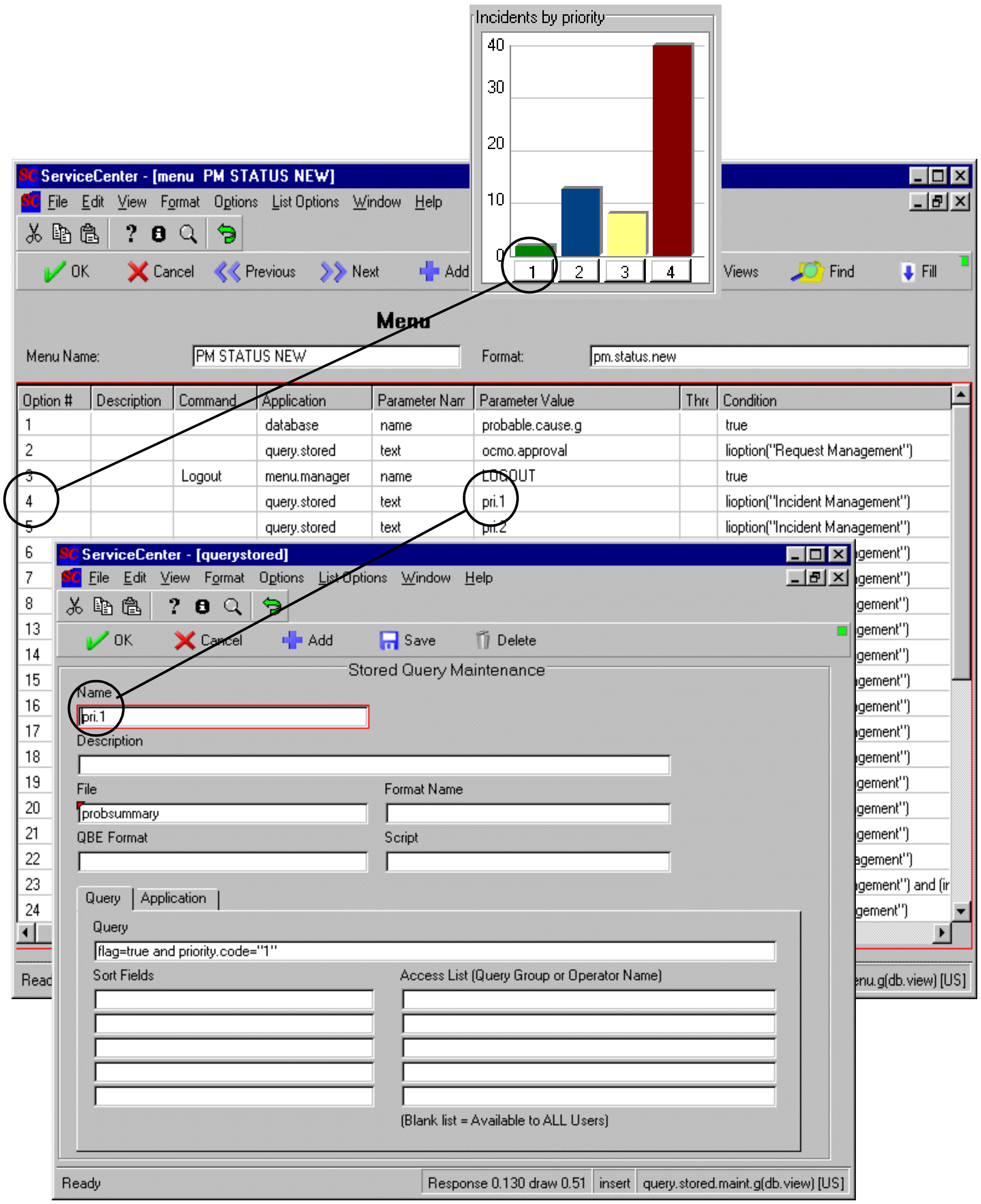
Menu buttons

Buttons that are configured to run stored queries from a menu must have a button ID defined in a menu record. Use the following values to run a stored query from a menu:

- Application: query.stored
- Parameter Name: text
- Parameter Value: <*stored query name*>
- Condition: true (or a condition statement that checks for a particular query group)

The following menu record is from MAX.MANAGER (IM STATUS NEW), whose startup menu contains two charts, one of which displays incident tickets by priority. The button bar at the bottom of this chart runs the queries that display tickets for each priority. The query in this example returns a list of priority 1 tickets.

Note: The number on the button in the chart has no relationship to the Button ID in the menu record.


Incidents by priority
40
30
20
10
0
1
2
3
4
ServiceCenter - [menu PM STATUS NEW]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Views Find Fill
Menu
ServiceCenter - [querystored]
File Edit View Format Options List Options Window Help
OK Cancel Add Save Delete
Stored Query Maintenance
Query Application

Menu Name: PM STATUS NEW
Format: pm.status.new

Option #	Description	Command	Application	Parameter Nam	Parameter Value	Thre	Condition
1			database	name	probable.cause.g		true
2			query.stored	text	ocmo.approval		lioption("Request Management")
3		Logout	menu.manager	name	LOGOUT		true
4			query.stored	text	pri.1		lioption("Incident Management")
5			query.stored	text	pri.2		lioption("Incident Management")
6							gement")
7							gement")
8							gement")
13							gement")
14							gement")
15							gement")
16							gement")
17							gement")
18							gement")
19							gement")
20							gement")
21							gement")
22							agement")
23							gement") and (ir
24							gement")

Read

nu.g(db.view) [US]

Name: pri.1
Description:
File: probsummary
Format Name:
QBE Format:
Script:

Query: flag=true and priority.code="1"
Sort Fields:
Access List (Query Group or Operator Name):
(Blank list = Available to ALL Users)

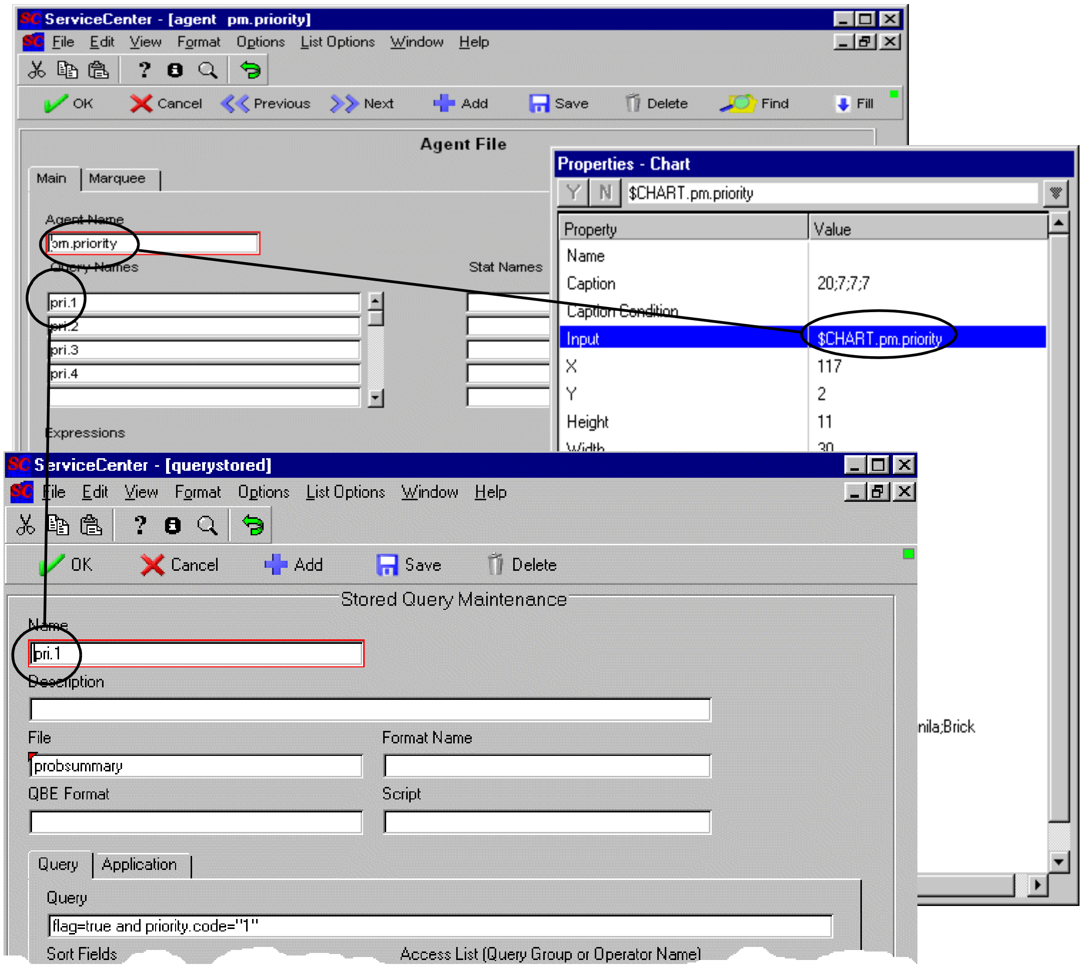
Ready | Response 0.130 draw 0.51 | insert | query.stored.maint.g(db.view) [US]

Figure 2-3: Stored query defined in a menu record

Charts and marquees

Stored queries can produce dynamic information that is gathered by the system and displayed in a bar chart. These queries are defined in agent records which are referenced by each display object in Forms Designer.

A marquee runs one stored query at a time and cannot provide access to actual records. Queries run from marquees are used to display messages *about* records. For example, the marquee in MAX.MANAGER's startup menu can be set to display a message describing the number of tickets of a particular priority that currently exist in the system.


ServiceCenter - [agent pm.priority]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete Find Fill
Main Marquee
Stat Names
Expressions
Properties - Chart
Y N
ServiceCenter - [querystored]
File Edit View Format Options List Options Window Help
OK Cancel Add Save Delete
Query Application
nila;Brick

Agent File

Agent Name: pm.priority

Query Names: pri.1, pri.2, pri.3, pri.4

$CHART.pm.priority

Property	Value
Name	
Caption	20;7;7;7
Caption Condition	
Input	$CHART.pm.priority
X	117
Y	2
Height	11
Width	30

Stored Query Maintenance

Name: pri.1

Description:

File: probsummary

Format Name:

QBE Format:

Script:

Query: flag=true and priority.code="1"

Sort Fields

Access List (Query Group or Operator Name)

Figure 2-4: Relationship of stored queries to an agent record

Scripts

Stored queries called from a script can populate forms with useful data. Use this capability to grant limited database access to certain users. An example of this might be to allow Level 1 technicians access to a user information form through which they can update selected elements of a caller’s contact record.

The stored query called by the script retrieves data from the **contacts** file, which is then displayed by the script. Once any updates are saved, the script continues walking the Level 1 technician through the normal call-taking workflow.

You may also execute a script from a stored query. For more information about scripting, refer to *Scripting* in this volume.

Menu option searches

Stored queries can be run from the Expert Search menu option in search forms for the principal ServiceCenter applications (Incident Management, Change Management, Inventory/Configuration Management, etc.) or from the Advanced Search menu option in the Database Manager. The appearance of these options and the features they control are dependent upon the capabilities defined for each user in the operator record. Refer to page 47 for a discussion of capability words associated with stored queries.

To run stored queries from a ServiceCenter application:

1 Open an application from the startup menu.

For this example, we are using MAX.MANAGER's menu with the capability word **query.stored** entered in his operator record.

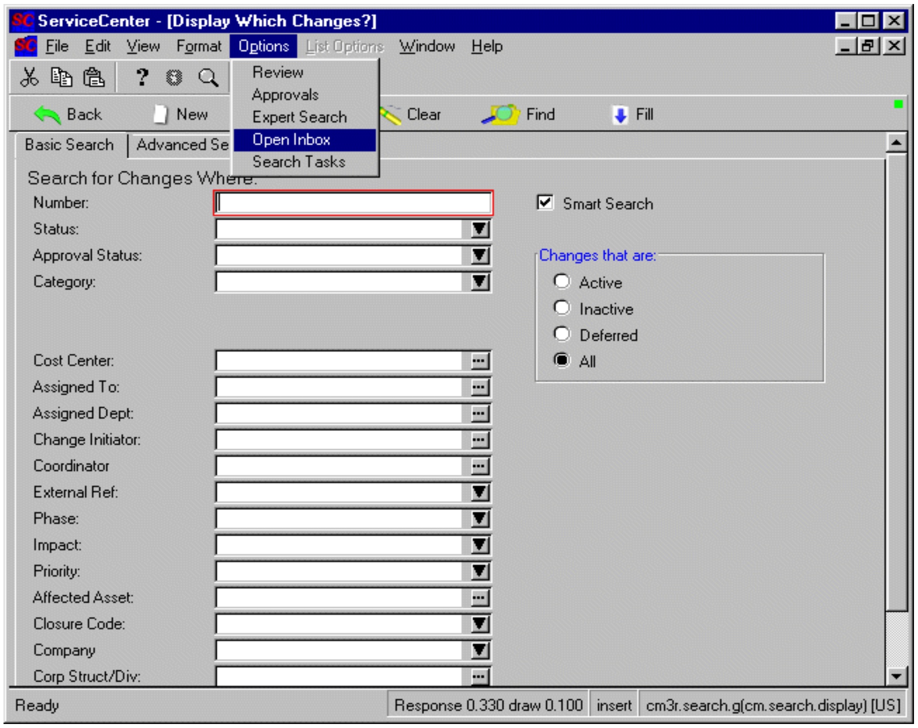
2 Click **Change Management**.

The Change Management menu is displayed.

3 Click **Search Changes**.

The Change Management search form is displayed.

4 Select **Options** > **Expert Search**.



Search for Changes Where:

Number:	
Status:	
Approval Status:	
Category:	
Cost Center:	
Assigned To:	
Assigned Dept:	
Change Initiator:	
Coordinator	
External Ref:	
Phase:	
Impact:	
Priority:	
Affected Asset:	
Closure Code:	
Company	
Corp Struct/Div:	

✓ Smart Search

Changes that are: Active / Inactive / Deferred / [✓ All]

Ready | Response 0.330 draw 0.100 | insert | cm3r.search.g(cm.search.display) [US]

Figure 2-5: Change Management search form

A QBE list of stored queries associated with Change Management and available to max.manager is displayed.

5 Double-click on the query you want to run or select it and press **Enter**.

Records matching the query are displayed in a QBE list. If only one record matches your search criteria, that record is displayed. If no change records match, the following message is displayed in the status bar: *No Changes found to satisfy search argument.*

6 Double-click on the record you want to view or select it and press **Enter**.

Capability Words

Access to certain stored query functions is controlled by capability words which appear in a user's operator record. The following capability words provide a full range of querying capabilities in the standard system:

- query.window
- query.stored
- query.stored.mod
- QueryAdmin

query.window

This capability allows a user to access the query window. Users with this capability cannot view stored queries, but may create their own query using the tools provided in the query window.

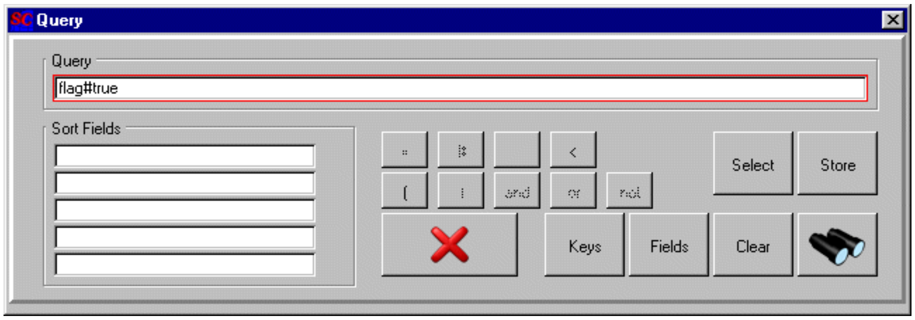
To perform an Expert Search using the query window:

1 Select **Options** > **Expert Search** in a search form (Incident Management, Change Management, or Inventory/Configuration Management).

The query window is displayed, showing a condition in the **Query** field that reflects the last search performed. For example, if you have searched for priority 1 tickets, either from the search form or with Expert Search, the following query appears in the window:

```
priority.code#"1" and flag#true
```

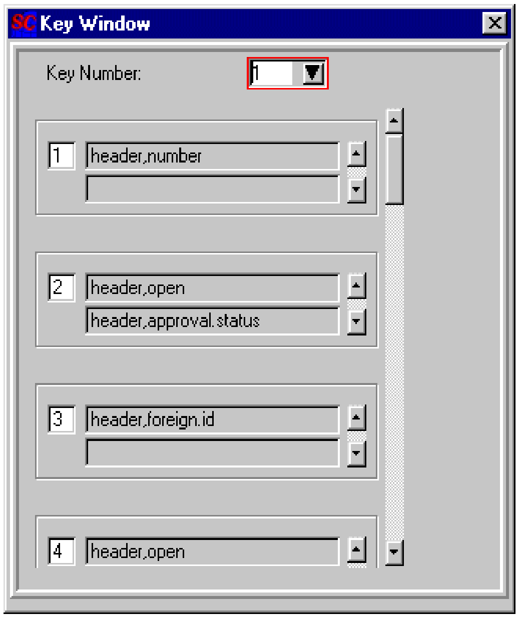
Note: The Select and Store buttons are not available to users with query.window as their only querying capability.



Query: flag#true

2 Click **Keys** to display the key structure as it appears in the Database Dictionary record for the host file.

For example, from the Incident Management search form, the key structure for the probsummary file is displayed.


Key Window

Key Number: 1

1 header.number

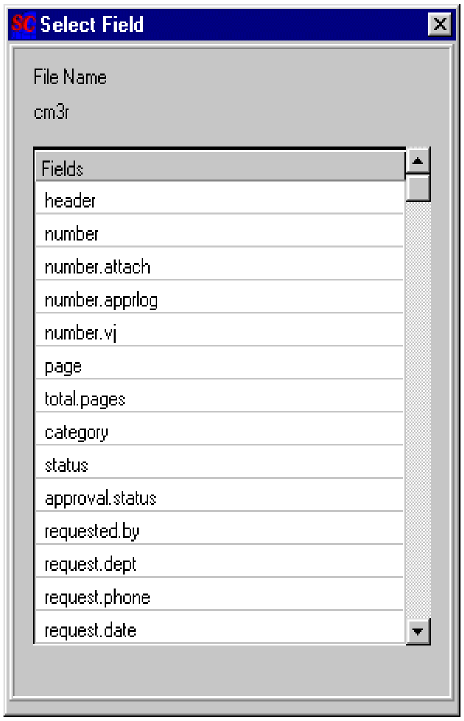
2 header.open
header.approval.status

3 header.foreign.id

4 header.open

3 Select a key to use in your search by entering the fields of the key in the Sort Fields array. Refer to page 38 for a definition of the Sort Fields array.

4 Click Fields to display a list of fields in the host file to use in constructing your query.



File Name

cm3r

Fields
header
number
number.attach
number.apprlog
number.vj
page
total.pages
category
status
approval.status
requested.by
request.dept
request.phone
request.date

5 Use the symbol buttons in the window to add operators such as &, >, =, or # to your query.

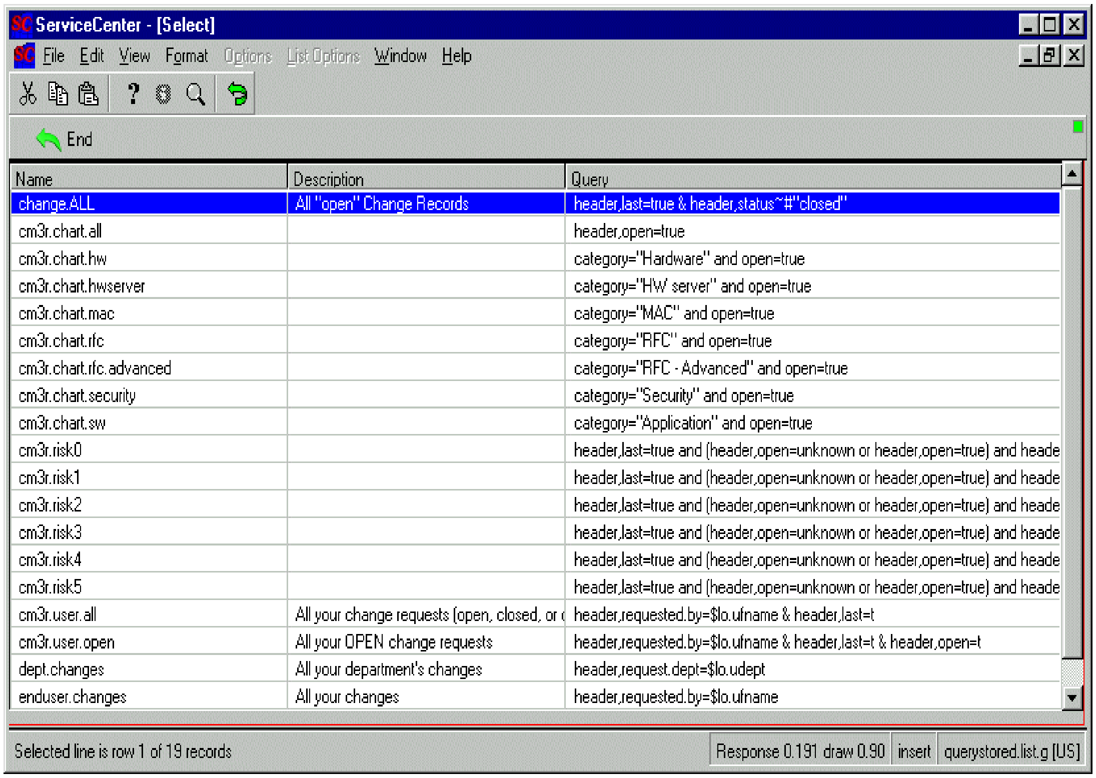
6 If you want to delete your work, click Clear to clear the **Query** field.

7 Click the Search button to search the database using your query.

Records matching the query are displayed in a QBE list. If only one record matches your search criteria, that record is displayed. If no records match, a prompt is displayed stating, *No records found.*

query.stored

This capability word allows a user to execute, but not modify, stored queries. Users with this capability word in their operator records see only a QBE list of appropriate stored queries when executing an Expert Search. Double-click on the query you want to run, or select it and press Enter.



Name	Description	Query
change.ALL	All "open" Change Records	header,last=true & header,status~#"closed"
cm3r.chart.all		header,open=true
cm3r.chart.hw		category="Hardware" and open=true
cm3r.chart.hwserver		category="HW server" and open=true
cm3r.chart.mac		category="MAC" and open=true
cm3r.chart.rfc		category="RFC" and open=true
cm3r.chart.rfc.advanced		category="RFC - Advanced" and open=true
cm3r.chart.security		category="Security" and open=true
cm3r.chart.sw		category="Application" and open=true
cm3r.risk0		header,last=true and (header,open=unknown or header,open=true) and heade
cm3r.risk1		header,last=true and (header,open=unknown or header,open=true) and heade
cm3r.risk2		header,last=true and (header,open=unknown or header,open=true) and heade
cm3r.risk3		header,last=true and (header,open=unknown or header,open=true) and heade
cm3r.risk4		header,last=true and (header,open=unknown or header,open=true) and heade
cm3r.risk5		header,last=true and (header,open=unknown or header,open=true) and heade
cm3r.user.all	All your change requests (open, closed, or c	header,requested.by=$lo.ufname & header,last=t
cm3r.user.open	All your OPEN change requests	header,requested.by=$lo.ufname & header,last=t & header,open=t
dept.changes	All your department's changes	header,request.dept=$lo.udept
enduser.changes	All your changes	header,requested.by=$lo.ufname

Selected line is row 1 of 19 records | Response 0.191 draw 0.90 | insert | querystored.list.g [US]

Figure 2-6: QBE list of stored queries for the probsummary file

query.stored.mod

Users with this capability word can view and select stored queries for a search or modify a stored query using the tools provided in the query window. Users cannot edit or delete existing stored queries, but can add new stored queries to the database.

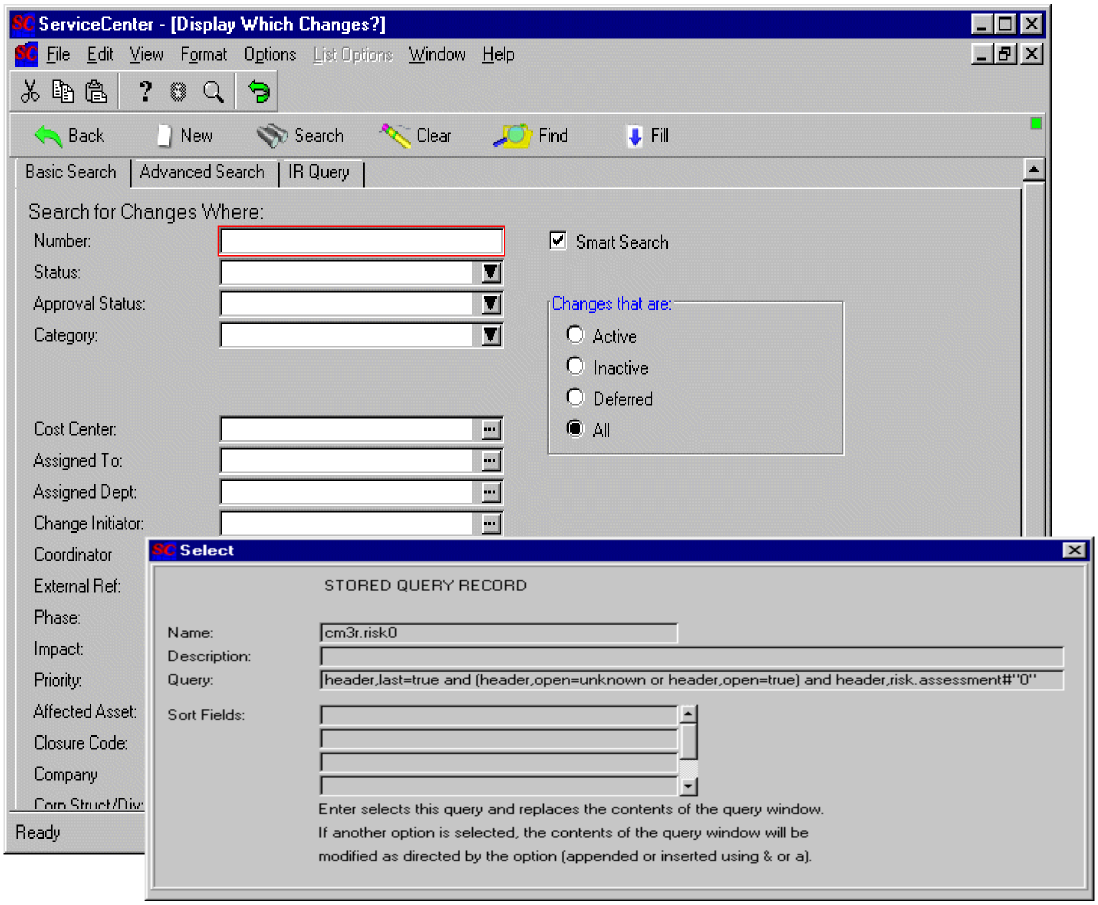
To modify a stored query:

1. Select **Option** > **Expert Search** in a search form (IM, CM, or ICM).

   The query window is displayed.

2. Click **Select** to display a list of stored queries appropriate for this file.
3. Double-click the query you want or select it and press **Enter**.

   The selected query is displayed in a read-only window with help text. Notice that the buttons have changed in the system tray.



[✓] Smart Search

Changes that are: Active / Inactive / Deferred / [● All]

STORED QUERY RECORD

Name: cm3r.risk0

Description:

Query: header,last=true and (header,open=unknown or header,open=true) and header,risk.assessment#"0"

Enter selects this query and replaces the contents of the query window. If another option is selected, the contents of the query window will be modified as directed by the option (appended or inserted using & or a).

Figure 2-7: Query-building options

4 Press Enter to move the query as it appears to the query window.

5 Click Select to display the list of stored queries again.

6 Select another query from the list.

Your second selection appears by itself in the read-only window.

7 Click one of the option buttons in the system tray to create a new query using both elements you have selected.

- Click Append using & to append the stored query to the end of a query you are creating using an and (&) operator.
- Click Append using | to append the stored query to the end of a query you are creating using an or operator.

- Click Insert using & to insert the stored query at the beginning of a query you are creating using an and (&) operator.
- Click Insert using | to insert the stored query at the beginning of a query you are creating using an or operator.

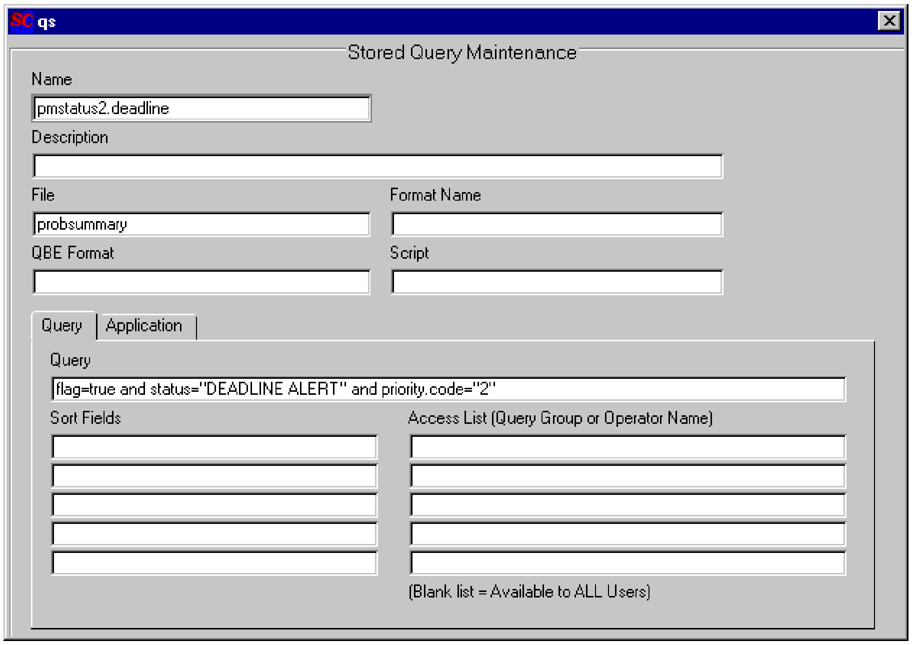
8 Click the **Search** button to run the query without saving it.

— or —

9 Click **Store** to add your modified query to the **querystored** file.

The value in the **Name** field of the Stored Query Maintenance form is that of the stored query last appended or inserted. The value in the **Query** field is the new query you have created.

10 Rename the query with a unique name descriptive of your new query, add a description, and any other controls you want.



Name: pmstatus2.deadline
Description:
File: probsummary
Format Name:
QBE Format:
Script:

Query: flag=true and status="DEADLINE ALERT" and priority.code="2"
Sort Fields:
Access List (Query Group or Operator Name)
(Blank list = Available to ALL Users)

Figure 2-8: New stored query record.

11 Click Add.

You are returned to the search form. The following message is displayed in the status bar: *Query added to querystored file.*

QueryAdmin

This capability gives this user access to all query capabilities listed above and also add/update access to the stored query database. A possible use of this capability word would be in a condition statement in a menu record granting database access for stored queries to specific users. To accomplish this, you would need to create an access point (button or menu option) in Forms Designer and an entry in the appropriate menu record. For example, you might use the following condition to establish user rights:

```
index("QueryAdmin", $lo.ucapex)>0
```

Note: Users with SysAdmin capabilities have access to all stored query functions. For more information on capability words, refer to the *System Administrator's Guide*.

3 CHAPTER

Sequential Number Setup

This chapter shows you:

- How to create a sequential number record.
- How to update a sequential number record.
- How to delete a sequential number record.

The Sequential Number file is used in conjunction with Format Control to generate sequence numbers for records in a database. The sequence number is automatically incremented or decremented when a new record is added. For instructions on creating a Format Control record with sequential numbering, refer to the Format Control documentation.

Accessing the Sequential Number File

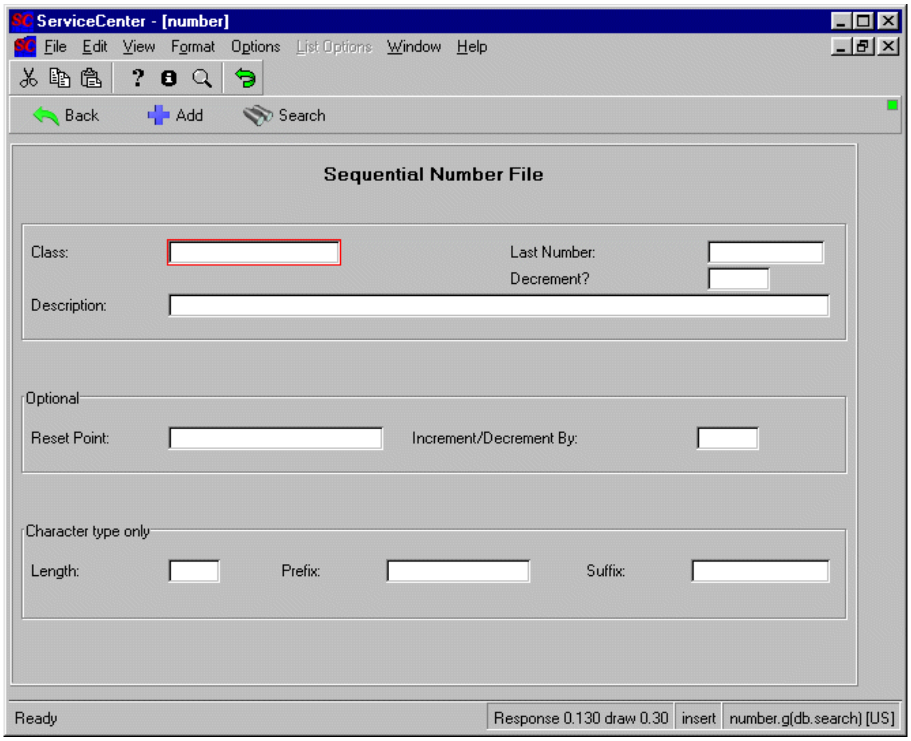
The sequential number records are accessed from the Tools menu under the Utility tab of the administrator's home menu.

To access the Sequential Number file:

1 Click the **Utilities** tab.

2 Click the **Tools** button.

3 Click **Sequential Numbers**.



Sequential Number File

Class:

Last Number:

Decrement?

Description:

Optional

Reset Point:

Increment/Decrement By:

Character type only

Length:

Prefix:

Suffix:

Ready | Response 0.130 draw 0.30 | insert | number.g(db.search) [US]

Figure 3-1: Blank Sequential Number Record

Sequential Number Fields

Class	A unique identifier for the sequential number record.
Last Number	The value from which the sequential numbering starts. For example, to start the numbering at one, this value is set to zero.
Decrement?	A boolean value controlling whether the sequential number increments or decrements. A value of **FALSE** or **NULL** (blank) means increment. A value of **TRUE** means decrement.
Description	A short explanation of the use of this sequential number.
Reset Point	The value at which the sequential number resets to its original starting value. If left blank, the sequential number never resets.
Increment/Decrement By	The value by which each number will increase or decrease.
Length	The total length of the number string for character type sequential numbers. Numbers are left-padded with zeros to reach this length. The length of the sequential number varies if this value is zero or blank.
Prefix	This string precedes the actual number for character-type sequential numbers. For example, if the desired format of a sequential number is **EMP99999** to number employee records, enter **EMP** in the prefix field.
Suffix	This string follows the actual number for character-type sequential numbers. For example, if the desired format of a sequential number is **99999EMP**, enter **EMP** in the suffix field.

Note: The number must be stored as a character type to use either the prefix or suffix.

Important: If you already have records in your database with employee number fields, be sure that your **last number used** contains a value greater than your existing employee number.

Creating a Sequential Number Record

Sequential numbers can be set up to work in several different ways. The examples that follow show you how to use each setup. The first example shows how to set up a normal number counter.

Setup a simple number counter

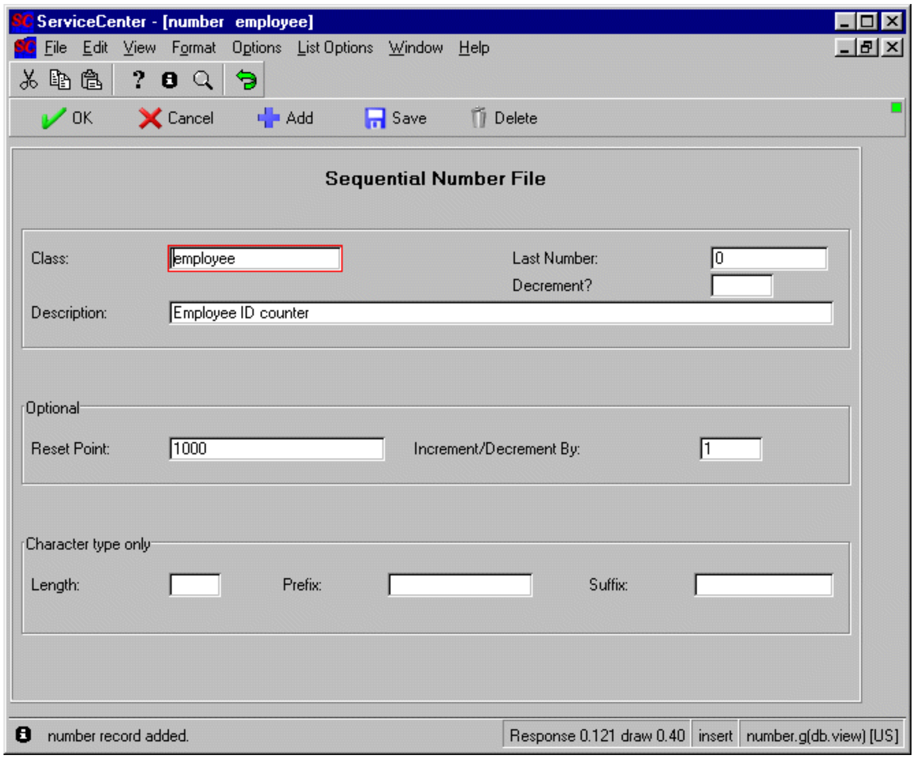
The following examples illustrates how to create a record called *employee* in the Sequential Number File to automatically increment employee numbers starting with 1.

1. Access the Sequential number file.
2. Create a new file. Enter `employee` in the Class field for this example.
3. Enter `0` in the Last Number field to begin incrementing from zero.
4. Enter a short description for the number class.

   For example: `Employee ID number counter`.
5. Enter `1000` in the Reset Point field.
6. Enter `1` for the Increment/Decrement field value.
   - Since the Decrement field was left blank, each number will increase by one.

7. Click the **Add** button.

The new sequential number record is added to the *sequential number* file (Figure 3-2 on page 59).


ServiceCenter - [number employee]
File Edit View Format Options List Options Window Help
OK
Cancel
Add
Save
Delete

Sequential Number File

Class: employee
Last Number: 0
Decrement?
Description: Employee ID counter

Optional
Reset Point: 1000
Increment/Decrement By: 1

Character type only
Length:
Prefix:
Suffix:

number record added.
Response 0.121 draw 0.40 | insert | number.g(db.view) [US]

Figure 3-2: New incrementing sequential number record.

Using decrement in sequential numbers

You can use sequential numbers to decrement a starting value. For example, you can decrement a quantity field when deleting stock from inventory. This example shows you how to decrement a value starting at 1000.

1 For this example, enter `active.devices` in the Class field.

2 Enter `1000` in the Last Number field.

3 Enter a short description for the number class.

For example, enter `Number of devices available`.

4 Enter `true` in the Decrement field.

5 Enter `0` in the Reset Point field.

6 Enter `1` for the Increment/Decrement field value.

Since the Decrement field was set to true, each number will decrease by one.

7 Click **Add**.

The new sequential number record is added to the *sequential number* file.

Using Prefix and Suffix in sequential numbers

This example uses prefixes and suffixes to assign character type ID numbers to workstation devices. The format of the ID is: DEV<number>T where DEV is a fixed character prefix, <*number*> is a sequential number starting with 1, and T is a fixed character suffix.

1 For this example, enter `devices` in the Class field.

2 Enter `0` in the Last Number field.

3 Enter a short description for the number class.

For example, enter `Workstation device ID counter`.

4 Leave the Decrement field blank.

5 Enter `1000` in the Reset Point field.

6 Enter `1` for the Increment/Decrement field value.

7 Enter `5` in the Length field.

8 Enter `DEV` in the Prefix field.

9 Enter `T` in the Suffix field.

10 Click **Add**.

The new sequential number record is added to the *sequential number* file. The first sequential number for the *devices* class will be **DEV00001T**.

Updating a Sequential Number Record

To update an existing sequential number record:

1 Access the sequential number record. Use the search function or select the record from a record list.

2 Enter any changes to the fields you want to update.

3 Click **Save**.

Deleting a Sequential Number Record

To delete an existing sequential number record:

1 Access the sequential number record that you want to delete. Use the search function or select the record from a record list.

For example, use the `employee` record created earlier in this chapter.

2 Click **Delete**.

You are prompted to confirm that you want to delete the record.

3 Click **Yes**.

The previous form is displayed with the message: *Record deleted from the number file*.

4 Scripting

CHAPTER

Under normal circumstances, the screen flow within ServiceCenter applications is controlled by the Rapid Application Development (RAD) code. Scripting allows you to interrupt the normal screen flow to display a series of forms, or execute decision-tree processing without modifying the original RAD code. Scripting does not affect the RAD screen flow.

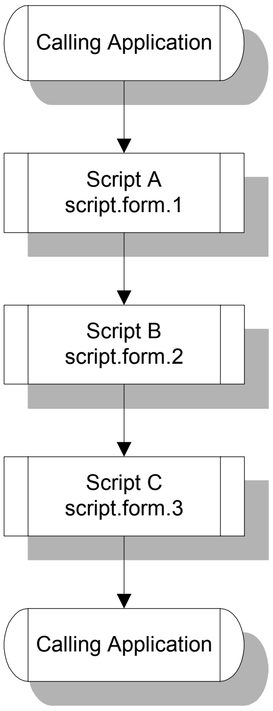
Scripting is useful for any process that requires an operator to supply prerequisite information. For example, during the incident determination cycle, you can create a script flow for operator-entered data. Based on how a caller replies to questions regarding the incident, your script determines which screen the operator sees next. While the script is executing, the operator-entered data is accumulated in a file variable which is returned to the calling application when the script is complete.

During execution, when a script displays a form, it has the look and feel of a customized RAD application and can be used in place of most RAD routines that are designed to gather data from a caller. Each script can display a standard ServiceCenter form and execute its Format Control record. The Format Control *display* processing is executed before the script form is displayed, and the *add* processing is executed after the operator selects the OK option. If a Format Control definition fails (an error condition is detected), the user is returned to the last script form displayed.

Scripting is also beneficial when multiple complex decisions must be made in order to reach a conclusion. For example, Change Management approval requirement conditions are normally based on the data contents of one field in the change record. Such a condition might be expressed as **header,risk.assessment in $cm3r**>4. However, there may be circumstances where the approval requirement condition is based on the values of several different fields. For instance, there may be three fields that affect the approval requirements: division, area, and department. Hard coding all the possible combinations of these field values into condition statements in RAD would involve a great amount of work and would be nearly impossible to maintain. You can define these conditions in scripting records which do not display forms, but which allow you to call a RAD subroutine or execute standard ServiceCenter processing statements (similar to Format Control calculation statements). These options allow for the manipulation of record data. At run time, the script becomes a decision-tree which results in significant processing reductions over the original method of RAD coding.

Script Flow

The *script flow* defines the order in which the script panels are executed. Scripts can move in a straight line from start to finish or branch into several possible processing flows. A simple flow is shown below:



A more complex script flow is controlled by condition statements that must evaluate to *true* before certain scripts can be executed. If none of the specified conditions evaluates to *true*, the script flow is considered complete, and control is passed back to the calling application.

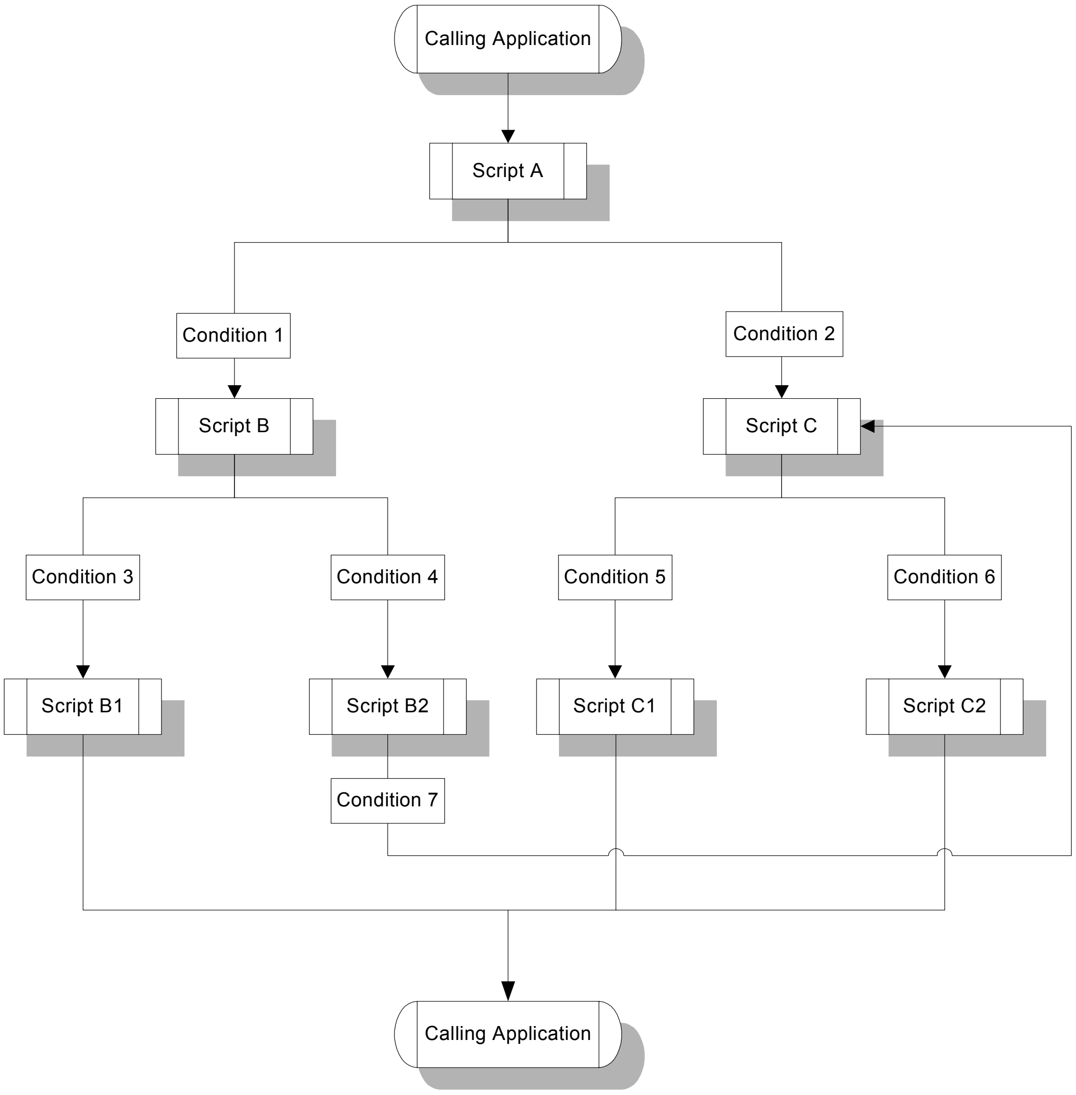


Figure 4-1: Complex Script Flow

Accessing Scripting

You can access the scripts file using any of three methods:

- Menu button
- Database Manager

Menu button

To access a script record from a menu:

1. Select the **Utilities** tab in the system administrator's home menu.
2. Click **Tools.**
3. Click **Scripts.**
4. Click **Search.**
5. Double-click a record, or select it and press **Enter.**

Database Manager

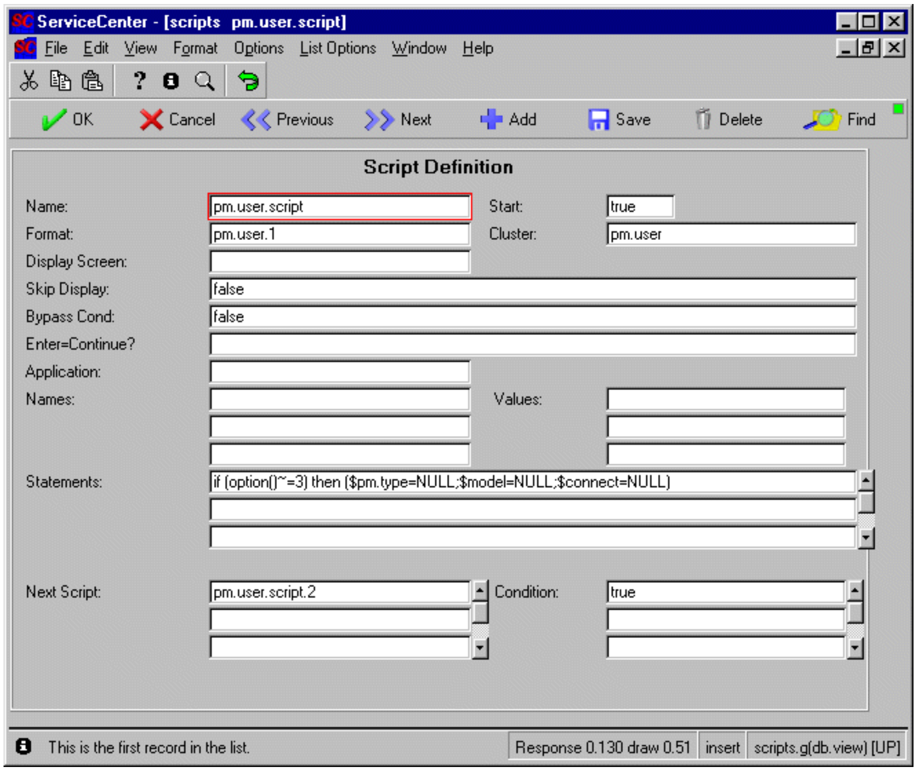
Accessing scripting from the Database Manager gives you additional options for manipulating the data. For example, you can perform a mass update to change the cluster name for each panel in your script flow or you can unload selected records to an external file.

To access a script record from the Database Manager:

1. Select the **Toolkit** tab in the system administrator's home menu.
2. Click **Database Manager.**
3. Enter scripts in the Form field.
4. Click the **search** button or press **Enter.**
5. Select *scripts* from the QBE list.
6. Click **Search.**

   A QBE list of existing script records is displayed.
7. Double-click a record, or select it and press **Enter.**

   The selected record is displayed.


ServiceCenter - [scripts pm.user.script]
File Edit View Format Options List Options Window Help
OK
Cancel
Previous
Next
Add
Save
Delete
Find

Script Definition

Name:	pm.user.script	Start:	true
Format:	pm.user.1	Cluster:	pm.user
Display Screen:			
Skip Display:	false		
Bypass Cond:	false		
Enter=Continue?			
Application:			
Names:		Values:	
Statements:	if (option()~=3) then ($pm.type=NULL;$model=NULL;$connect=NULL)		
Next Script:	pm.user.script.2	Condition:	true

This is the first record in the list. Response 0.130 draw 0.51 insert scripts.g(db.view) [UP]

Figure 4-2: Script record form

Script definition fields

Name-unique name of the script. This is a required field. Select a name that is descriptive of the script's function. A possible naming standard is:

<file name>.<function>.<sequence>

In this form, *<file name>* is the name of the ServiceCenter file, such as problem or cm3r, in which the script will be executed; *<function>* is a brief description of the purpose of the script, such as *priority, category,* or *resolution*; and *<sequence>* is a number within a cluster or within a certain branch of a script flow.

Start-optional logical field that indicates if this script is the first one displayed in a script flow. The value in this field has no effect on processing, but indicates which scripts are used as starting points when querying.

Note: Script Flow execution does not have to start with a script defined as a starting point.

Format-name of a standard ServiceCenter form displayed to the user when this script is executed. This is an optional field. Forms used by a script may be used elsewhere within ServiceCenter. The input fields displayed on all forms within a script flow are related to one dbdict (*problem, cm3r, device*, etc.). You can use $ variables in any script form; however, if you wish to retain their values in the *$script* file variable, you must create the necessary Format Control statements to copy the variables to input fields.

Cluster-optional field used to identify related scripts by grouping them under one common name. The value in this field has no effect on processing but is useful when querying.

Display Screen-optional field naming the unique screen ID of a `displayscreen` file record from the Display application. The options defined in this record named are available when the script form is displayed. If this field is blank, the system uses the default Database Manager options.

Skip Display-optional logical field that controls whether or not to skip displaying the script's form. When this condition evaluates to *true*, the form is not displayed, and processing determines which script should be executed next. The default is *false*.

Bypass Cond-optional logical field that specifies the condition that must exist for a user to bypass the script. If this condition evaluates to *true*, a Skip button is displayed in the system tray. The default is *false*. When Skip is selected, the script flow currently executing is terminated, and processing control returns to the calling application. The system administrator always has the authority to bypass the script.

Enter=Continue?-optional logical field that specifies the condition that must exist for the Enter key to have the same function as the OK button. If this condition evaluates to *true,* users can press either the Enter key or click OK to continue script processing. If the condition evaluates to *false* and the user presses Enter, the current script is re-displayed. The default is *false.*

Application-optional field naming a RAD application to call when this script is executed.

Names-parameter names for the RAD application named in the **Application** field.

Values-parameter values for the RAD application named in the **Application** field.

Statements-optional arrayed statements field that defines the processing statements to be executed when the script is processed. For example, specific values can be assigned to fields when this script is executed.

Next Script-name of the next script to be executed when the associated condition in the **Condition** field evaluates to *true*.

Condition-logical field that specifies the condition that must exist for the next script to be executed. At processing time, these conditions are evaluated from the top down. The script associated with the first condition to evaluate to *true* is the next script that is executed. If no exits are specified, or if all conditions evaluate to *false*, the script flow is considered complete and processing control returns to the calling application. Blanks in the condition field are treated the same as an evaluation of *false*. To base an exit condition on the value of an input field, use the *$script* file variable. For example, if the input field **name** is to be tested for a value of joe, the condition statement would be written as follows as:

name in $script="joe"

Due to processing considerations, the **Next Script** and **Condition** fields are independent arrays. You must scroll each array separately to keep the script names synchronized with their corresponding conditions.

Note: The Script maintenance applications check for Format Control *add,* and *display* options on the *scripts* form. You can define Format Control statements, if desired.

Processing Flow

Each script can execute a ServiceCenter form, displayoptions, a RAD subroutine, and condition statements. Note that each script can use all of these options. The sequence of this processing is:

1. Execute any *display* Format Control processing associated with the script's form.
2. Display the script form.
3. Execute any *add* Format Control processing associated with the script's form.
4. Execute statements defined in the script.
5. Execute the RAD application defined in the script.

Creating a Script

There are four steps to building a script:

1. Design and diagram the script flow, showing the names of all script panels and forms.
2. Design and create the script forms using Forms Designer. Create any support links and Format Control that might be necessary for your forms.
3. Create the script definition records.
4. Add your script to the ServiceCenter application, Format Control record, or stored query record from which it will be executed.

Diagraming the script flow

Develop a map of the entire flow, showing the name of the form (if any) displayed by each panel, and the conditions controlling the flow from panel to panel. Follow the map throughout the scripting process to avoid simple errors that will prevent your script from executing properly. Figure 4-3 on page 73 shows part of a conditional flow designed to gather the necessary information to open an incident ticket directly from a script. For the purposes of this example, only two of the conditional exits for category-specific information have been diagramed.

The first panel of this script provides an incident ticket number for your incident. The second panel displays a form requesting category and asset information. Depending upon the category selected, the third panel displays a category-specific form before exiting back into the common flow.

Successive forms gather the remainder of the information needed to open the ticket: assignments, details, service agreements, and contacts. When the script exits, the new ticket, containing all the data you have accumulated, is displayed and can be added to the database.

Notice that the two conditional script panels for the **hw.desktop** and **hw.mainframe** categories display the same form. You may write a single condition statement to include all the categories using that form.

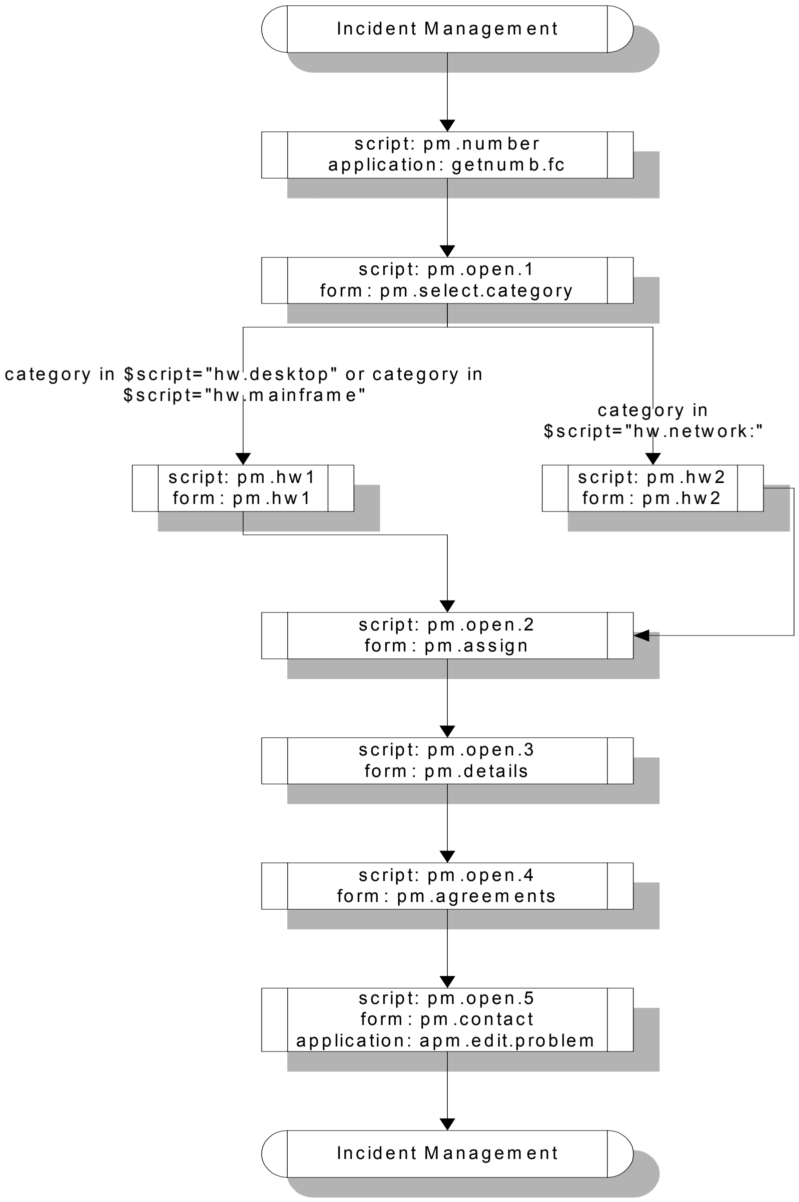


Figure 4-3: Scripting Flow Chart

Creating the forms

If your script requires special forms, create them now using the Forms Designer utility. Be sure to follow the naming conventions defined in your map. The examples in Figure 4-5 on page 76 show the script forms used to gather information about incidents. In this example, the second script panel (*pm.open.1*) displays the *pm.select.category* form, allowing the technician taking the call to select category and asset information. The category-specific forms required can be created by copying all the related fields directly from the incident ticket update form (e.g., **problem.hw.desktop.update.g**) for that category. The next panel in the common flow (*pm.open.2*) displays the *pm.assign* form for assignment group information and so on, until all the necessary data has been recorded about the incident. The final script panel, *pm.open.5*, creates a new incident record using all the accumulated data.

Using Fill boxes in script forms

For a Fill box to function properly on a script form, you must create the following additional records:

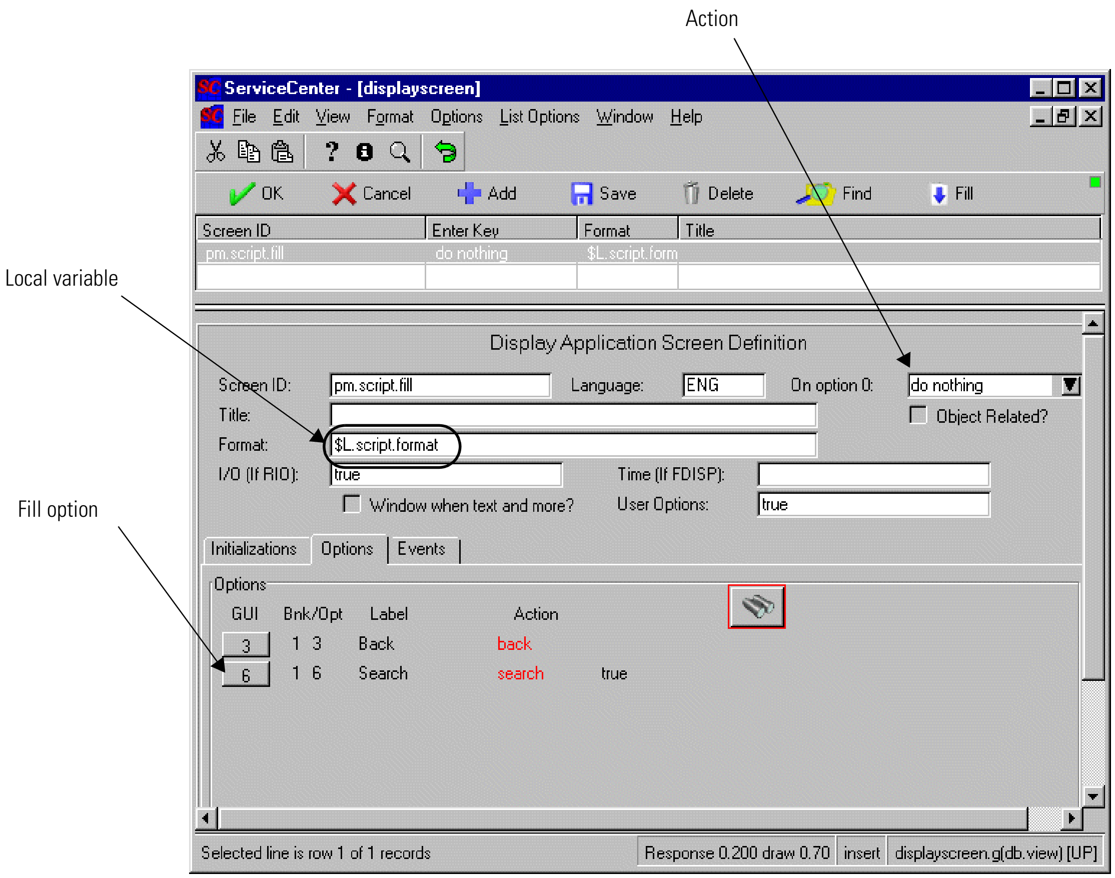
- Link record: Select Options > Link from Forms Designer and establish a link between the field in your script form and the field in the source file.
- Format Control record: Select Options > Format Control from Forms Designer and add a Format Control record granting Fill privileges.

  – or –

- Displayscreen record: The displayscreen controls the options that are displayed with a form. Create a displayscreen record in which you define displayoptions for your script forms, including a Fill button. Enter the Screen ID of your new record in the **Display Screen** field of each script definition that displays a form in which a Fill box appears. You may create unique displayscreen records for each form in the script flow if necessary, or associate a single record with all the forms.

Note: You can use a combination of displayscreen records and Format Control to activate Fill buttons in your script.

- If you create a displayscreen record and wish to bind its options to more than one script form, you must enter the local variable `$L.script.format` in the **Format** field of the displayscreen record. This variable is hardcoded into the **script.execute** application.


Action
Local variable
Fill option
ServiceCenter - [displayscreen]
File Edit View Format Options List Options Window Help
OK Cancel Add Save Delete Find Fill

Screen ID	Enter Key	Format	Title
pm.script.fill	do nothing	$L.script.form	

Display Application Screen Definition

Screen ID: pm.script.fill
Language: ENG
On option 0: do nothing
Object Related?
Title:
Format: $L.script.format
I/O (If RIO): true
Time (If FDISP):
Window when text and more?
User Options: true

Initializations | Options | Events

Options

GUI	Bnk/Opt	Label	Action	
3	1 3	Back	back	
6	1 6	Search	search	true

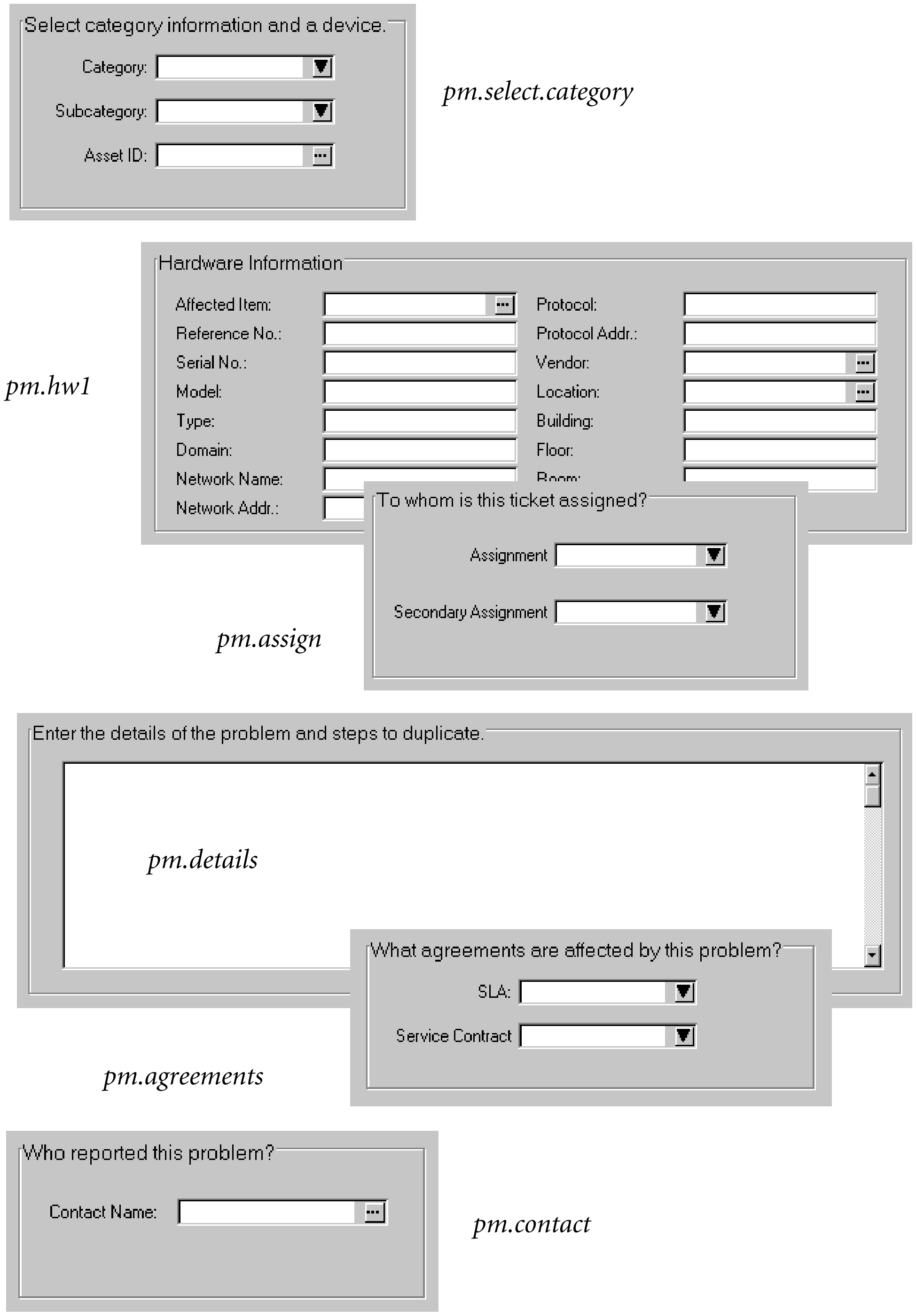
Selected line is row 1 of 1 records | Response 0.200 draw 0.70 | insert | displayscreen.g(db.view) [UP]

Figure 4-4: Displayscreen record using the scripting local variable

- When defining displayoptions for your script forms, refer to the following list of possible actions that are hardcoded into the **script.execute** application. Your choice of options in the **Action** field is limited to the values in this list:

ok	back	cancel	skip	views
extend	find	fill	validatefield	useroptions
redraw	re draw	closeapplication		

Note: Use the **ok** action when defining a Continue option.


Select category information and a device.
Category:
Subcategory:
Asset ID:
pm.select.category
Hardware Information
Affected Item:
Reference No.:
Serial No.:
Model:
Type:
Domain:
Network Name:
Network Addr.:
Protocol:
Protocol Addr.:
Vendor:
Location:
Building:
Floor:
Room:
pm.hw1
To whom is this ticket assigned?
Assignment
Secondary Assignment
pm.assign
Enter the details of the problem and steps to duplicate.
pm.details
What agreements are affected by this problem?
SLA:
Service Contract
pm.agreements
Who reported this problem?
Contact Name:
pm.contact

Defining the scripts

After diagramming the flow, the next step is to create each panel in the script flow. Make sure to follow the naming conventions for scripts and forms you have established in your script map.

To create a script definition:

1 Access a blank script definition record. Use one of the procedures described in the *Accessing Scripting* section beginning on page 67.

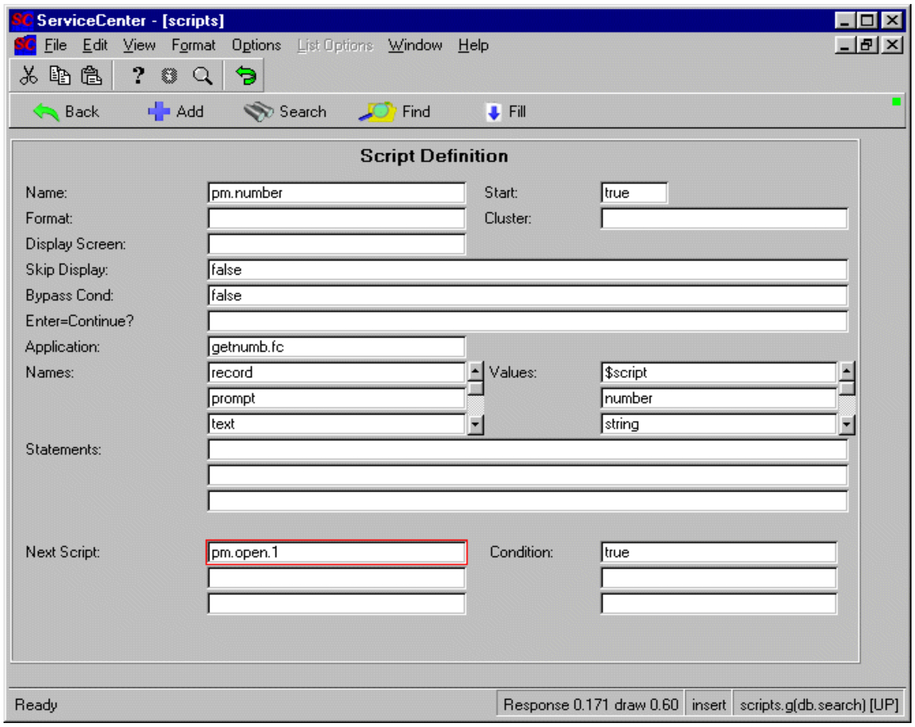
2 Complete the definition for the first script in your flow.

For example purposes, enter the following values:

Field	Value	Parameter Values/Conditions
Name	pm.number	
Start	true	
Application	getnumb.fc	
Name	record	$script
	prompt	number
	text	string
	name	Incident Management
	string1	PM
Next Script	pm.open.1	true

Note: The value for **Start** is true, indicating that this is the first panel in the script flow.

This panel associates a properly incremented incident ticket number with the data you are going to accumulate from the caller. This panel calls a RAD application only and does not display a form.


ServiceCenter - [scripts]
File Edit View Format Options List Options Window Help
Back Add Search Find Fill
Script Definition

Name:	pm.number	Start:	true
Format:		Cluster:	
Display Screen:			
Skip Display:	false		
Bypass Cond:	false		
Enter=Continue?			
Application:	getnumb.fc		
Names:	record	Values:	$script
	prompt		number
	text		string
Statements:			
Next Script:	pm.open.1	Condition:	true

Ready | Response 0.171 draw 0.60 | insert | scripts.g(db.search) [UP]

Figure 4-6: Sample script definition

3 Click New to add your panel to the **scripts** database.

Script record added appears in the message bar.

4 Click **Back** to return to a blank script definition form.

5 Complete the definition for the second script in your flow.

In this example, we would enter the following values:

Field	Value	Condition
Name	pm.open.1	
Format	pm.select.category	
Skip Display	false	

Field	Value	Condition
Bypass Cond	false	
Next Script	pm.hw1	category in $script=“hw.desktop” or category in $script=“hw.mainframe”
	pm.hw2	category in $script=“hw.network”

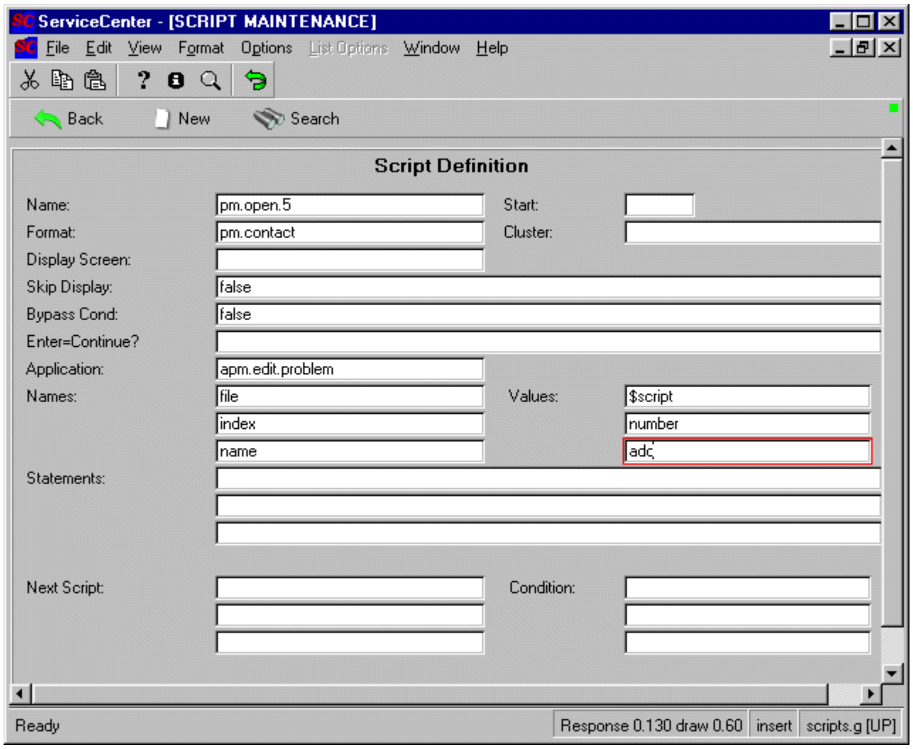
Note: This script lists conditional exits for only three categories. An actual script would require enough panels and conditions to handle all the categories in your system.

6 Click New to add your panel to the `scripts` database.

7 Create the remainder of the script definitions in your flow.

In this example, we must create specific definitions for each category (*pm.hw1* and *pm.hw2*), as well as the remainder of the scripts in the common flow (*pm.open.2* through *pm.open.5*). Make sure that each panel except the last panel in the flow has the name of the next panel to execute in the **Next Script** field.

In this example, panel *pm.open.5* displays a form and calls the RAD application **apm.edit.problem**. This application displays any information you have gathered in an incident ticket and allows you to abandon the incident or add it to the database.


ServiceCenter - [SCRIPT MAINTENANCE]
File Edit View Format Options List Options Window Help
Back New Search

Script Definition

Name:	pm.open.5	Start:	
Format:	pm.contact	Cluster:	
Display Screen:			
Skip Display:	false		
Bypass Cond:	false		
Enter=Continue?			
Application:	apm.edit.problem		
Names:	file	Values:	$script
	index		number
	name		adc
Statements:			
Next Script:		Condition:	

Ready | Response 0.130 draw 0.60 | insert | scripts.g [UP]

Figure 4-7: Last script panel in the flow

Leave the **Next Script** field blank in *pm.open.5*. This is the last panel in the script flow and must exit back to the calling application (Incident Management).

Executing the script

Now that you have created your forms and the script definitions to display them, you must decide how you want to execute the script. You have several choices:

- Incident Management (using profile records)
- Display option (from within an application)
- Format Control
- Stored query

Incident Management

Profile records within Incident Management allow you to specify the script to execute when a user opens a new incident ticket. You can design a script that displays forms containing only those fields necessary for recording the particulars of an incident. For example, if the technician taking the call selects **hw.network** as the incident category, a condition statement in the script exits to a panel that displays a network hardware related form. When the script has finished, the record carrying all the necessary information is created, and the technician is ready to work on another issue.

To define an initial script in a Incident Management profile record:

1 Click **Incident** Management in the system administrator's home menu.

2 Click **Security Files.**

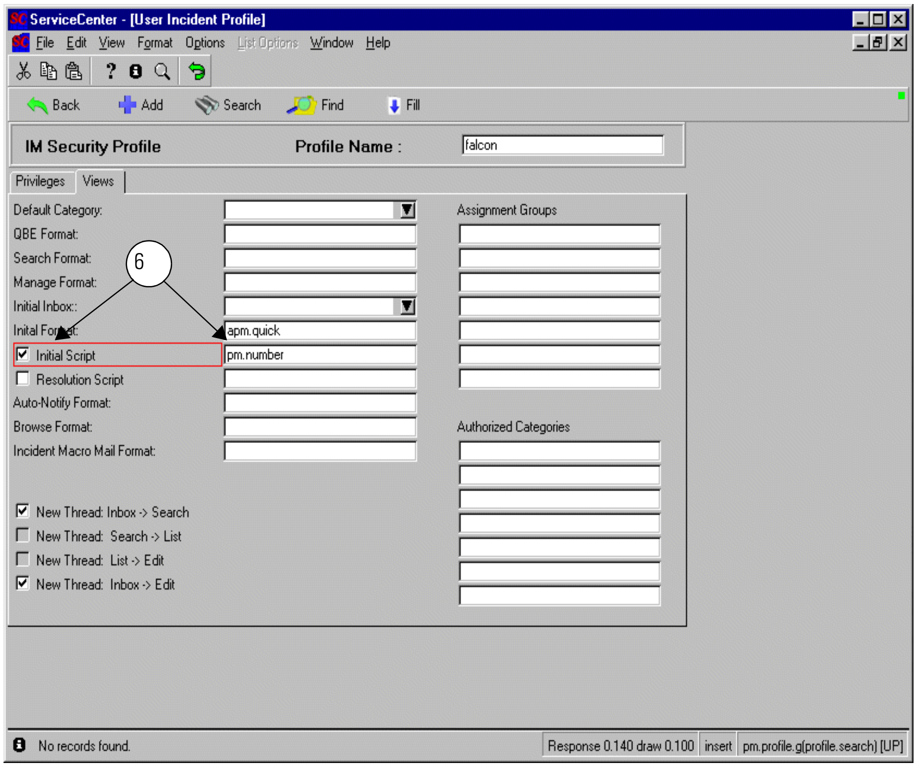
3 Click **Search/Add** in the incident Profiles structure of the Security Files tab.

4 Enter a profile in the Profile field and press **Enter**.

5 Select the **Views** tab in the profile record.

6 Check the **Initial Script** check box and enter the name of the first panel of your script in the adjacent field.

In this example, pm.number has been entered.



ServiceCenter - [User Incident Profile]

File Edit View Format Options List Options Window Help

Back Add Search Find Fill

IM Security Profile Profile Name : falcon

Privileges | Views

Default Category:		Assignment Groups
QBE Format:		
Search Format:		
Manage Format:		
Initial Inbox::		
Inital Format:	apm.quick	
☑ Initial Script	pm.number	
☐ Resolution Script		
Auto-Notify Format:		
Browse Format:		Authorized Categories
Incident Macro Mail Format:		

☑ New Thread: Inbox -> Search
☐ New Thread: Search -> List
☐ New Thread: List -> Edit
☑ New Thread: Inbox -> Edit

No records found. | Response 0.140 draw 0.100 | insert | pm.profile.g(profile.search) [UP]

Figure 4-8:

7 Click Update to save your changes.

The following message appears in the status bar: *The record has been updated.*

Displayoption

Use a displayoption to create a button or an Options menu selection on a ServiceCenter form from which to execute your script. For information on the Display application, refer to *Display Application* in this volume.

To execute a script from a displayoption:

1 Determine the screen ID of the form for which you want to define the option.

You may use the RAD debugger for this purpose or look up the ID in the appendix of *Display Application*.

2 Open a blank displayoption record using one of these methods:

- Enter do in the command line and press Enter.

- Enter displayoption in the **Form** field of the Database Manager dialog box and click Search or press Enter.
- Click Display Options in the Tools menu (accessed from the Utilities tab in the system administrator's home menu).

3 Enter the screen ID for the form you want in the **Screen ID** field.

4 Click Search or press Enter.

The displayoption records for that form are listed.

5 Scan the list of options to determine an available number for your new option.

Remember: option numbers < 200 appear as buttons in the system tray, and option numbers > 200 appear as Options menu items.

6 Create the record from scratch or modify an existing record.

7 Enter script.execute in the **RAD Application** field and pass in the following parameters:

Name	Value
file	$L.filed
name	Name of the first script panel in the flow

8 Click Add.

Format Control

Scripting can be called from the Subroutines process of Format Control. When executing a script from Format Control, it is important to place the script call as the *last* item on the Subroutine call list. Scripting, itself, calls Format Control, and it is conceivable that the Format Control records associated with the forms displayed by the script flow could manipulate data associated with the initial Format Control.

To execute a script with Format Control:

1 Select the Utilities tab in the system administrator's home menu.

2 Click Tools.

3 Click Format Control.

4. Enter the name of the Format Control record you want to edit in the **Name** field.

   For the previous example, enter `apm.quick`.
5. Click Search or press Enter.
6. Click Subroutines or select Options > Subroutines.
7. Enter `script.execute` in the **Application Name** field and pass in the following parameters:

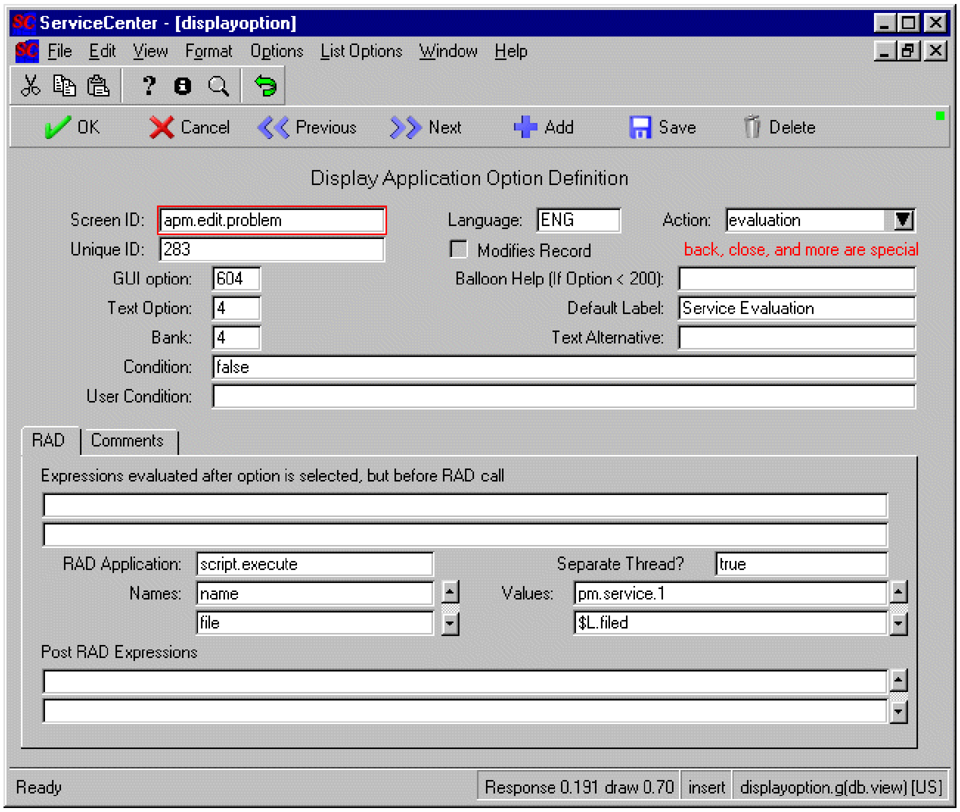
Name	Value
file	$file
name	Name of the first script panel in the flow

8. Set the **Before** field to *true*, or to a condition that evaluates to true, to execute the script before any other processing takes place.
9. Set the **Display** field to *true*, or to a condition that evaluates to true, to execute the script before *apm.quick* is displayed.
10. Click Back twice.
11. A prompt is displayed asking if you want to save the changes you have made to the record.
12. Click the OK button.

Stored query

A script executed from a stored query can facilitate a search for records by allowing the user to select precise search parameters. The ServiceCenter standard system contains an inactive script that can be executed from a stored query to display a list of key words relating to previous incidents. The stored query then uses the key word selected to search for incidents of a similar nature.

This stored query can be run from an open incident ticket using either Format Control or a displayoption (Screen ID: *apm.edit.problem*). In this example, a displayoption has been created that adds an option called Probable Cause to the Options menu of an open incident ticket. This option runs the stored query called *probcause.user*. For instructions on using stored queries, refer to *Stored Queries* in this volume.


ServiceCenter - [displayoption]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete

Display Application Option Definition

Screen ID:	apm.edit.problem	Language:	ENG
Action:	evaluation		
Unique ID:	283	Modifies Record	☐
			back, close, and more are special
GUI option:	604	Balloon Help (If Option < 200):	
Text Option:	4	Default Label:	Service Evaluation
Bank:	4	Text Alternative:	
Condition:	false		
User Condition:			

RAD | Comments

Expressions evaluated after option is selected, but before RAD call

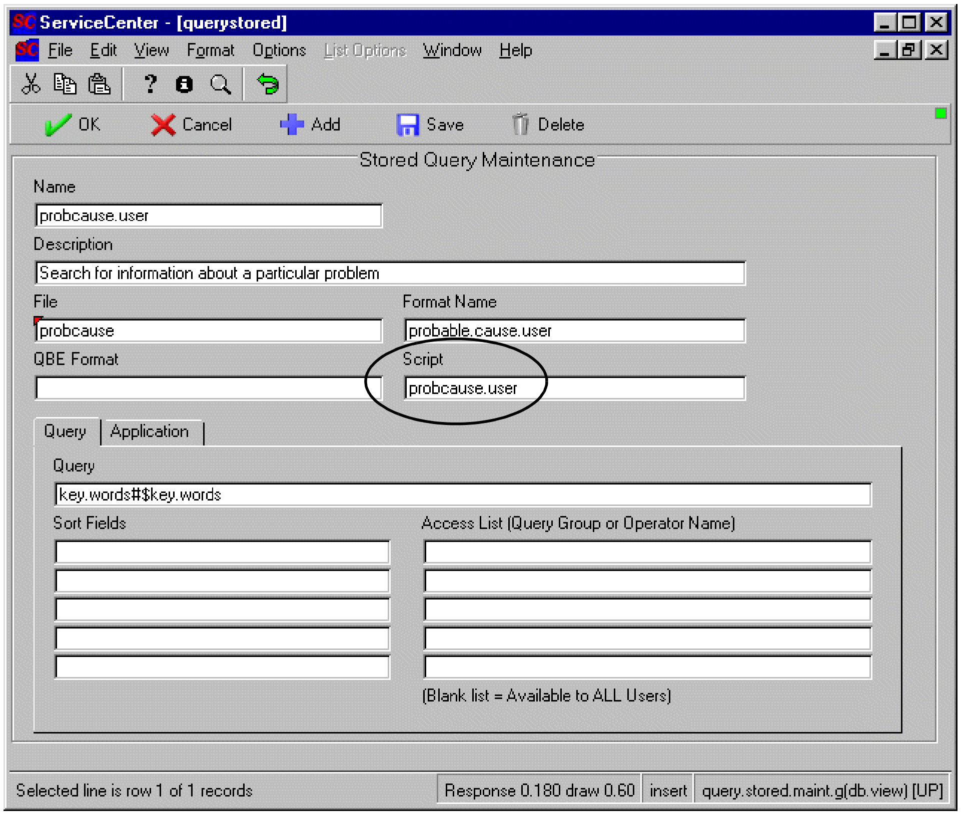
RAD Application:	script.execute	Separate Thread?	true
Names:	name	Values:	pm.service.1
	file		$L.filed

Post RAD Expressions

Ready | Response 0.191 draw 0.70 | insert | displayoption.g(db.view) [US]

Figure 4-9: Displayoption record that calls a stored query

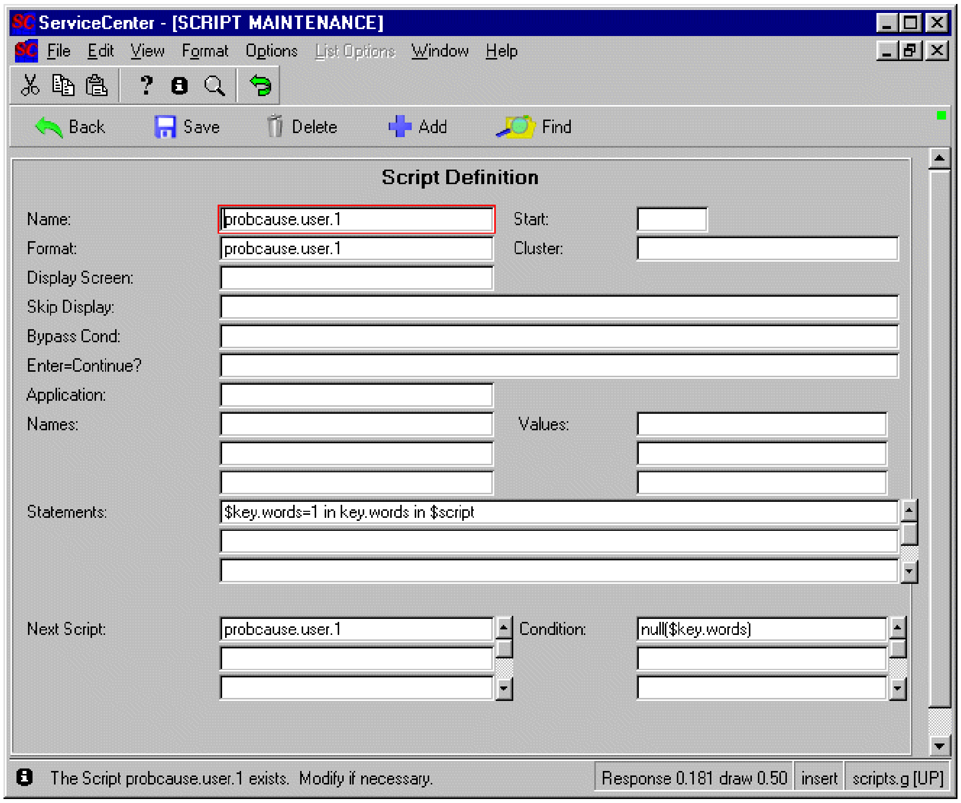
The stored query, *probcause.user*, executes the script called *probcause.user.1*.


ServiceCenter - [querystored]
File Edit View Format Options List Options Window Help
OK Cancel Add Save Delete
Stored Query Maintenance
Query Application

Name: probcause.user
Description: Search for information about a particular problem
File: probcause
Format Name: probable.cause.user
QBE Format:
Script: probcause.user
Query: key.words#$key.words
Sort Fields
Access List (Query Group or Operator Name)
(Blank list = Available to ALL Users)
Selected line is row 1 of 1 records | Response 0.180 draw 0.60 | insert | query.stored.maint.g(db.view) [UP]

Figure 4-10: Stored query maintenance record that executes a script

The script, *probcause.user.1*, has a single panel which displays the *probcause.user.1* form. The condition for exit (**null**(**$key.words**)) requires the user to choose a key word before allowing the script to exit.


ServiceCenter - [SCRIPT MAINTENANCE]
File Edit View Format Options List Options Window Help
Back Save Delete Add Find

Script Definition

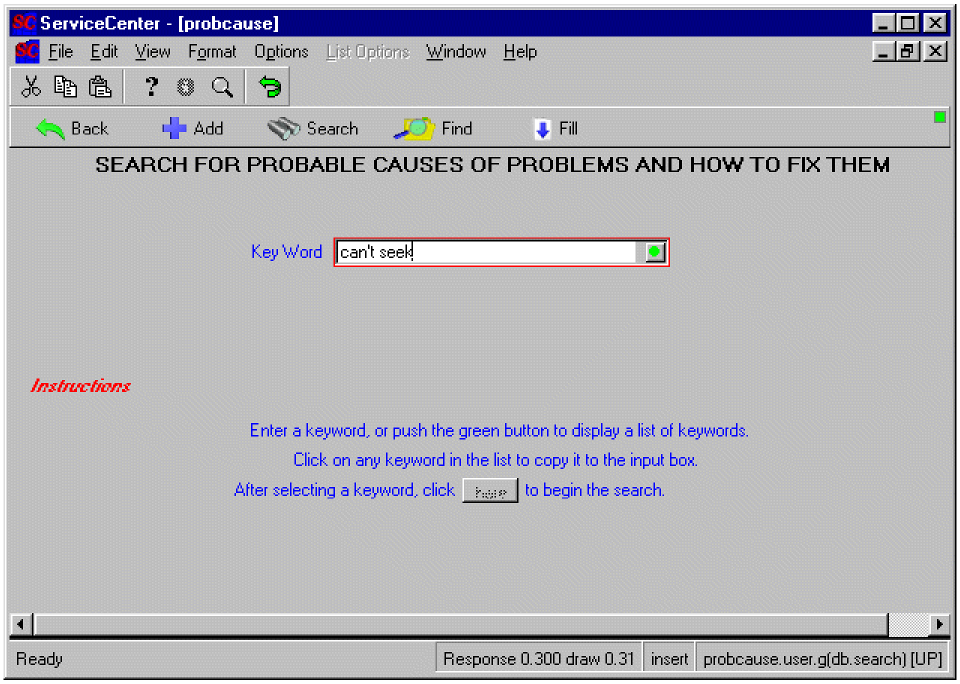
Name:	probcause.user.1	Start:	
Format:	probcause.user.1	Cluster:	
Display Screen:			
Skip Display:			
Bypass Cond:			
Enter=Continue?			
Application:			
Names:		Values:	
Statements:	$key.words=1 in key.words in $script		
Next Script:	probcause.user.1	Condition:	null($key.words)

The Script probcause.user.1 exists. Modify if necessary. Response 0.181 draw 0.50 insert scripts.g [UP]

Figure 4-11: Script definition executed from a stored query.

Important: Scripts executed from stored queries typically set values into the same fields used by the query to retrieve records. In this example, the common field is **key.words**.

The script is executed before the form from the query (*probable.cause.user*) is displayed. The form displayed by the script is called *probcause.user.1* and supplies the query form with a key word for searching.


ServiceCenter - [probcause]
File Edit View Format Options List Options Window Help
Back
Add
Search
Find
Fill

SEARCH FOR PROBABLE CAUSES OF PROBLEMS AND HOW TO FIX THEM

Key Word can't seek

Instructions

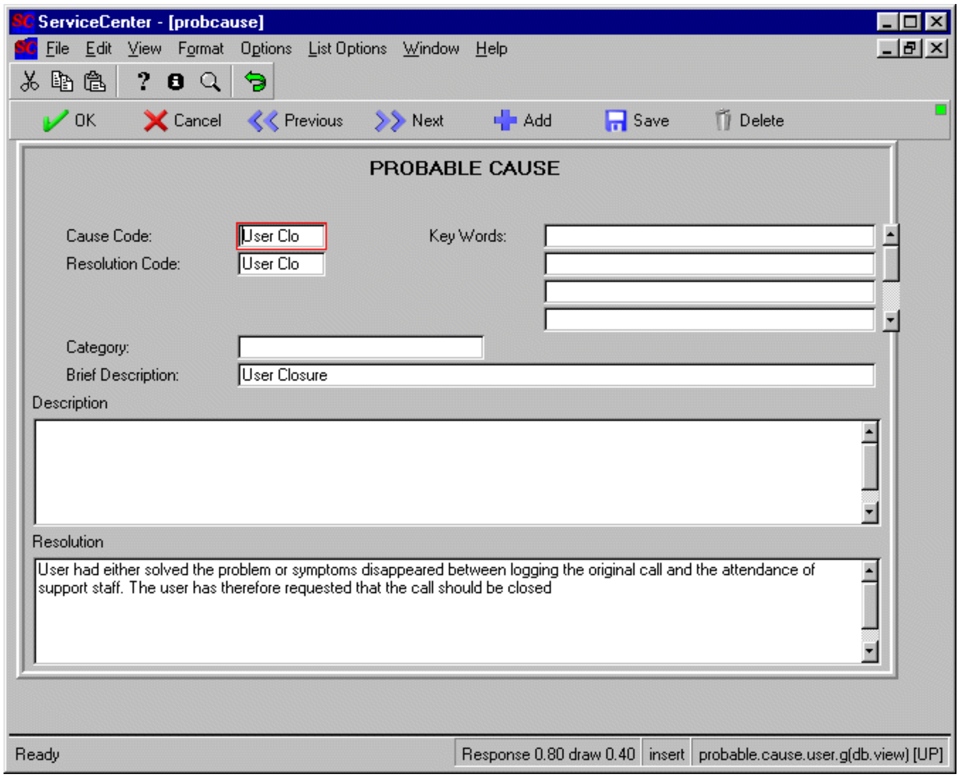
Enter a keyword, or push the green button to display a list of keywords.
Click on any keyword in the list to copy it to the input box.
After selecting a keyword, click [illegible] to begin the search.

Ready | Response 0.300 draw 0.31 | insert | probcause.user.g(db.search) [UP]

Figure 4-12: Probable cause key word script form

When the user selects a key word and clicks **OK**, the script exits. The stored query then uses the key word to search the **probcause** file for matching entries. If more than one match is found, the system displays a QBE list of possible causes for your incident.

Double-click a selection to view the details with the *probable.cause.user* form. When you have finished viewing all the choices, click Cancel to return to the incident ticket.


ServiceCenter - [probcause]
File Edit View Format Options List Options Window Help
OK
Cancel
Previous
Next
Add
Save
Delete

PROBABLE CAUSE

Cause Code:	User Clo	Key Words:	
Resolution Code:	User Clo		
Category:			
Brief Description:	User Closure		

Description

Resolution

User had either solved the problem or symptoms disappeared between logging the original call and the attendance of support staff. The user has therefore requested that the call should be closed

Ready | Response 0.80 draw 0.40 | insert | probable.cause.user.g(db.view) [UP]

Figure 4-13: Probable cause record accessed from a stored query

Deleting a Script

You can delete script panels and their related forms manually or allow the system to delete all the elements of a script flow automatically.

To delete a script flow automatically:

1 Select the first script in the flow using the procedures described in *Accessing Scripting* on page 67.

Note: If you select any other script panel in the flow, you may be given a partial listing only. Partial lists are stopped at any script panel in the flow whose display is controlled by a conditional statement. This feature allows you to isolate a single conditional branch of a script flow for deletion.

2 Click **Delete**.

A confirmation prompt is displayed, asking if you want to delete the related forms as well.

3 Select the **Delete associated forms** checkbox if you want to have the system delete all related forms automatically.

4 Click the **Delete** button.

   All the script panels in the flow and all the associated forms are listed. Scripts or forms used elsewhere in the system are not displayed.

5 Click the **Delete** button to delete all the items listed in the form.

   If the deletion procedure has been successful, the following message is displayed in the status bar: *The Script/Format delete process is complete. Check for error messages.*

6 Click the **Cancel** button to exit the delete routine and return to the last form displayed.

Script Reports

Script reports allow you to display your entire flow with different views. You may view the entire flow or just a part of it. You can display and print the details of each panel for comparison or troubleshooting. You have three choices of script reports you can print:

- **Script Flow**—generates a summary report of all possible script flow paths based on the assumption that the currently displayed script panel is the starting point of the flow. This report allows you to isolate each separate branch within your script.
- **Script Detail**—lists all the fields in the script definition record and the values you have entered for the currently displayed script panel.
- **Script Tree**—generates a summary report showing the logical flow of the script panels in an outline form. The currently displayed script panel is used as the starting point. This report allows you to see the relationships of panels to one another at a glance.

Each report has the same header, printed in the default font for your printer. The following is a sample header:

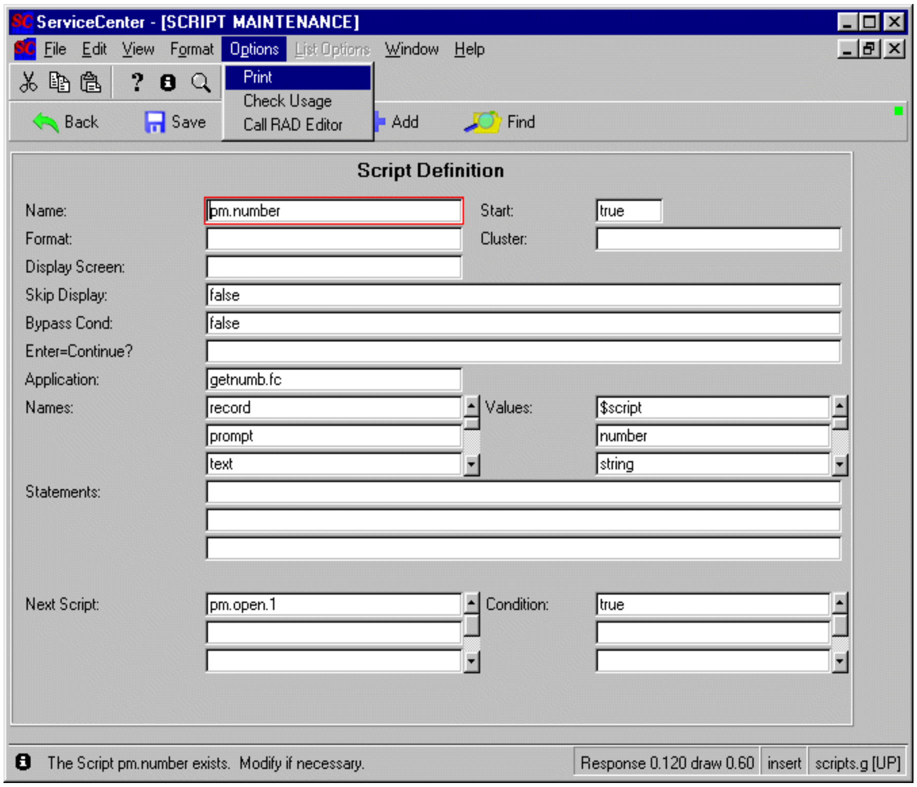
```
        ServiceCenter
Date:Print of spool record

07/31/2000 11:51

Operator: falcon  Selection:

Sequence:



Name: Script Path PrintNumber:843Page: 1
```

To print a report on a script:

1. Access a script definition record, using one of the procedures described in *Accessing Scripting* on page 67.

   You may report on any script panel in the flow. If you want to view the entire flow, run the report from the first panel.

2. Select **Options** > **Print.**


ServiceCenter - [SCRIPT MAINTENANCE]
File Edit View Format Options List Options Window Help
Print
Check Usage
Call RAD Editor
Back
Save
Add
Find

Script Definition

Name:	pm.number	Start:	true
Format:		Cluster:	
Display Screen:			
Skip Display:	false		
Bypass Cond:	false		
Enter=Continue?			
Application:	getnumb.fc		
Names:	record	Values:	$script
	prompt		number
	text		string
Statements:			
Next Script:	pm.open.1	Condition:	true

The Script pm.number exists. Modify if necessary. Response 0.120 draw 0.60 insert scripts.g [UP]

Figure 4-14: Print option for script reports

You are prompted to select a report type.

3 Select one of the radio buttons.

4 Click the **Print** button.

Script flow

Script flow reports may have more than one page depending upon the complexity of the script, particularly if conditional exits are defined. The following is an example of the first page of a script flow report, without the header:

```
NameFormat

ClusterBypass

  Next Panel Condition

...............................................



Script Path Summary List

             ........................

pm.number

pm.open.1

pm.hw2

pm.open.2

pm.open.3

pm.open.4

pm.open.5


pm.number
```

Script detail

The following is an example of a script detail report for the first panel in a flow, without the header. You may print a detail report on any panel in the flow.

```
NameContents

Name: pm.number

             Format:

              Start: true

            Cluster:

       Skip Display:

        Bypass Cond:

     Enter=Continue?
```

Script tree

In the example below, the report shows the hierarchy of each panel in the flow and the conditional statements that control the exits. This report can span several pages, depending upon the complexity of the conditional branches.

```
Script Flow

  pm.number.cond:

  applic: getnumb.fc

    pm.open.1 cond: true


      pm.hw2 cond: category in $script="hw.network"


        pm.open.2 cond: true


          pm.open.3 cond: true


            pm.open.4 cond: true
```

5 Plug-ins

CHAPTER

ServiceCenter plug-ins permit tight data and process integration between ServiceCenter applications and external automation or data sources. While the ServiceCenter system has a number of outstanding integration and workflow capabilities, there are times when a low level extension of the ServiceCenter platform is required. Plug-ins are designed expressly for this purpose.

There are a number of technologies that can be integrated directly into ServiceCenter, for example:

- User authentication can occur using a third-party scheme
- ServiceCenter data records can be populated from external sources
- External data sinks can be populated from ServiceCenter data records
- Client integration can be created directly with services on or connected to the client machine
- CORBA services can be invoked to provide inbound or create outbound data
- Entirely new scripting languages, such as ECMA script or PERL can be invoked
- Java classes can be instantiated and invoked

The plug-in model follows that used by many popular technologies, including Web browsers, the Java Virtual Machine (through JNI), scripting engines such as Java script or PERL.

Plug-ins can be called from anywhere in ServiceCenter where RAD is executed. Some examples of points where plug-ins can be called:

- Through a trigger
- Through Format Control
- Through a link record
- Through a script
- Through Display Options
- Through the Document Engine

Plug-ins are supplied by Peregrine, its partners, or written by customers.

Plug-In Platform Support

ServiceCenter plug-ins are supported on the following platforms:

- AIX
- HP-UX
- Solaris
- Windows NT 4+
- Linux

Plug-In Functions

There are only three functions that are called by the ServiceCenter executables. The functions are called to initiate and terminate a plug-in, and to execute a function called from the RAD program

SCPluginInitialize

```
SC_EXPORT int SCPluginInitialize(PSCPLUGIN_ENV penv)
```

SCPluginTerminate

```
SC_EXPORT int SCPluginTerminate(PSCPLUGIN_ENV penv)
```

SCPluginExecute

```
SC_EXPORT int SCPluginExecute(PSCPLUGIN_ENV penv, int iArgCount,
PDATUMARRAY pdArray);
```

Installing a Plug-In

The Windows DLL or UNIX library that contains the plug-in code, and the plug-in's dependent libraries (if any) must be available in the path of the ServiceCenter executable that calls the plug-in.

If your plug-in is called only from RAD and is used in an express client environment (recommended), then the plug-in can reside only on the server.

Plug-ins are configured in the sc.ini file, which must reside in the current working directory of the ServiceCenter server or client executable.

The format of the plugin-related entries in sc.ini is:

```
pluginN:<libraryname>
```

where *N* is the sequence number assigned to the plugin starting with "0" for the first plugin, and <**libraryname**> is the name of the DLL or shared library.

For example, a sc.ini file might contain:

```
plugin0:sample.dll
plugin1:anotherPlugin.dll
plugin2:yetAnotherPlugin.dll
```

You can have any number of plugin definitions in your sc.ini file, but the sequential numbering must start with 0 and continue without interruption. Any gaps in the numbering will cause higher-numbered plugins not to be loaded.

For example, the following sc.ini file is badly formed. Plugin1 will not be loaded because it is the first one and it is not assigned the number zero:

```
plugin1:neverLoaded.dll
```

Calling a Plug-In from RAD

RAD is the expression language of ServiceCenter and is used throughout the system for tailoring. Calling the plug-in from RAD involves a simple expression:

```
$return = plugin( name, [var1, … varN])
```

Each operand, including the return value, is a RAD variable. The variable can be either a part of a ServiceCenter record (`name in $file` for example) or a simple variable such as `$user`. The variables are passed to the plug-in by reference, meaning that any changes made to a variable by the plug-in are reflected to the RAD calling routine.

Operands

$return

Specifies a RAD variable that will indicate the success or failure of the plug-in function call. By convention, a zero return indicates success; a non-zero return indicates some sort of failure. If the return value is negative, it indicates an error locating or calling the plug-in. If the return value is positive, the plug-in has indicated some sort of condition. It is up to the plug-in author to create and document the various return codes.

name

Indicates the name of the plug-in. As each plug-in is loaded, it identifies itself with a character name "java", or "perl" for example. Consult the plug-in documentation for the proper name.

var1, … varN

Zero or more variables that are passed to the plug-in. The variables are passed by reference, meaning that they may be input or output variables as determined by the plug-in code. The proper number of arguments should be passed to the plug-in.

Supported Types

The following types of RAD variables are currently supported for use in plug-ins:

- NUMBER
- CHARACTER
- DATETIME
- BOOLEAN
- FILE
- ARRAY
- STRUCTURE

The NUMBER type is implemented as a the C/C++ type **double** and can be changed to **int** or **long** by casting.

Creating a Plug-In

Creating a plug-in requires that you write a DLL for use on Windows systems or a shared library for use on UNIX systems. The library requires methods to initialize and terminate the plug-in environment and to execute a plug-in function when called from the RAD language.

Peregrine provides a sample plug-in that simply dumps the input variables to the ServiceCenter log file. Peregrine also provides a template to help start writing a plug-in.

Include file

The include file is located in the "include" directory of the ServiceCenter "plugins" distribution directory.

scplugin.h

Defines the plug-in interface, which consists of:

1. A set of C/C++ functions to be implemented by your DLL or shared library
   - SCPluginInitialize
   - SCPluginExecute
   - SCPluginTerminate
2. A set of API macros for manipulating DATUM types
3. A structure called SCPLUGIN_ENV which is used as a communication area between ServiceCenter and your plug-in

DATUMs

Like most loosely typed environments (JavaScript, Visual Basic, ActiveX, etc.), ServiceCenter provides variables to the scripting language through a coercible data type. In ServiceCenter, the type is called a DATUM. DATUMs are similar to other scripting language "variants" both in structure and function. Inspecting and changing DATUMS is simplified through a number of macros which are described below.

To give you some idea of how ServiceCenter implements DATUMs, a very simplified view of a DATUM is given below. However, plug-ins do **not** map DATUMs directly using a structure such as the one given below. Instead, the macros defined in scplugin.h describe the DATUM as a void pointer, and provide a set of macros which call functions to manipulate DATUMs.

This is done to ensure that plug-ins do not have to be recompiled to work with future versions of ServiceCenter, and to make sure that if the internal form of the DATUM changes within ServiceCenter in the future, an older plug-in will not inadvertently corrupt data.

```
typedef struct datum
{
 union
 {      /* fixed-length value of datum */
  FLOAT           da_float;       /* NUMBER */
  SCALAR_TIME     da_time;        /* TIME */
  INT16           da_int16;       /* BOOLEAN,LOCAL,OPERATOR */
  struct rel_blk *  da_rel;       /* RELATION */
 }  da_fixed;
 STRING       da_varying;         /* STRING */
 char         da_type;            /* type of datum */
 char         da_state;           /* state of datum */
} DATUM;
typedef DATUM * PDATUM;
```

Note that the DATUM contains a structure called a STRING. In ServiceCenter, the STRING is not zero or null-terminated. Internally, ServiceCenter maintains the current and maximum length of the STRING for efficiency sake, similar to a Java StringBuffer.

Note: All string values exchanged between ServiceCenter and the plug-in are null-terminated C strings. This is done for the convenience of the plug-in programmer.

Macros

Macros provide a simple way for the plug-in developer to manipulate ServiceCenter DATUMS and their contained STRINGs. To use the macros, the pointer to the environment variable must be named “**penv**”.

Note: Macros are all upper case.

Macro Definitions

int DA_TYPE(PDATUM pd)

Returns the integer value of the type code for the DATUM pointed to by pd. To convert this value to a string, call DA_TYPENAME using the type code.

char * DA_GETTYPENAME(int nTypeCode)

Returns a pointer to a null-terminated string containing the name of the type of DATUM indicated by nTypeCode.

Note: You must free any pointer returned by this macro by calling DA_FREESTRING. All character pointers returned by routines beginning with "DA_GET" (DA_GETTYPENAME, DA_GETSTRING, DA_GETTIMESTRING, etc.) must be freed using DA_FREESTRING.

int DA_ISNULL(PDATUM pd)

Returns an integer truth value indicating whether the DATUM pointed to by pd is null or not. The C++ version of the DA_ISNULL macro returns bool.

int DA_ISAGGREGATE(PDATUM pd)

Returns an integer truth value indicating whether the DATUM pointed to by pd represents an aggregate type, i.e. a STRUCTURE or an ARRAY. The C++ version of the DA_ISAGGREGATE macro returns bool.

int DA_LENGTH(PDATUM pdAggregate)

Returns an integer indicating how many entries there are in the aggregate DATUM pointed to by pd. Only a STRUCTURE or ARRAY DATUM can be passed to DA_LENGTH.

PDATUM DA_FIRST(PDATUM pdAggregate)

Returns a pointer to the first contained DATUM of an aggregate DATUM, or NULL if pdAggregate is not an aggregate DATUM, or if the aggregate DATUM is NULL, i.e. the DA_ISNULL macro would return true for each contained DATUM.

PDATUM DA_NEXT(PDATUM pdAggregate, PDATUM pdPrev)

Returns a pointer to the next contained DATUM of aggregate DATUM `pdAggregate` after the previously retrieved DATUM `pdPrev`, or NULL if there is no next DATUM following `pdPrev`. NULL is also returned if `pdAggregate` is not an aggregate DATUM, or if `pdPrev` is not contained within pdAggregate.

PDATUM DA_LAST(PDATUM pdAggregate)

Returns a pointer to the first contained DATUM of an aggregate DATUM, or NULL if `pdAggregate` is not an aggregate DATUM, or if the aggregate DATUM is NULL, i.e. the DA_ISNULL macro would return true for each contained DATUM.

PDATUM DA_ITEM(PDATUM pdAggregate, int index)

Returns a pointer to the contained DATUM within aggregate DATUM `pdAggregate` represented by the integer value `index`.

double DA_GETNUMBER(PDATUM pd)

Returns the double value of the DATUM pointed to by `pd`. Your program can cast this value to a different numeric data type such as `int` or `long`.

DA_SETNUMBER(PDATUM pd, double lValue)

Sets a DATUM pointed to by `pd` to the type "NUMBER" and gives it the value `lValue`. Any numeric type (such as int or long) may be passed to the DA_SETNUMBER macro. The value is coerced to a double by the macro.

int DA_GETBOOLEAN(PDATUM pd)

Returns the integer truth value of the BOOLEAN DATUM pointed to by `pd`. Returns an integer truth value indicating whether the DATUM pointed to by `pd` is null or not. The C++ version of the `DA_GETBOOLEAN` macro returns `bool`.

DA_SETBOOLEAN(PDATUM pd, int bTrue)

Sets a DATUM pointed to by `pd` to the type "BOOLEAN" and gives it the value bTrue. A `bool` value should be passed to the C++ version of `DA_SETBOOLEAN`.

char * DA_GETSTRING(PDATUM pd)

Returns a pointer to a null-terminated character string containing the string value of the DATUM pointed to by pd. Your program is responsible for passing this pointer to DA_FREESTRING when the character string is no longer needed. Failure to do this will cause a memory leak in the ServiceCenter server.

Warning: Do not call "free" or "delete" in your plugin code with pointer values returned by DA_GET routines such as DA_GETSTRING. Doing so will cause the ServiceCenter processes executing your plugin code to crash. You must call DA_FREESTRING instead. Also, do not call DA_FREESTRING with pointer values allocated by your plugin code using "malloc", "new", or other storage allocation routines. Doing so will cause heap corruption and an eventual crash of all ServiceCenter processes running your plugin code.

DA_FREESTRING(char* pszString)

Frees the storage pointed to by pszString. Must only be called with values returned by one of the DA_GET routines, such as DA_GETTYPENAME, DA_GETSTRING, DA_GETTIMESTRING, DA_GETFILENAME, or DA_GETFIELDNAME

DA_SETSTRING(PDATUM pd, char * pszValue)

Sets a DATUM pointed to by pd to the type "STRING" and gives it the value contained in the null-terminated string pszValue. The null-terminatednull-terminated string pszValue is allocated by your plug-in and is not modified or freed by ServiceCenter. You are responsible for managing any storage represented by pszValue.

char * DA_GETTIMESTRING(PDATUM pd)

Returns a pointer to a null-terminated character string containing the date/time value for the DATUM pointed to by pd, as a GMT time. For absolute date/time values, the returned string is formatted as "YYYY/MM/DD hh:mm:ss", for example: "2002/04/22 14:00:00".

For elapsed time values a string formatted as “ddd hh:mm:ss:” where “ddd” represents the number of days and is subject to zero suppression. Only as many digits as needed to represent the number of days will appear. The “ddd” part will not appear if the elapsed time is less than one day. Your program is responsible for freeing the storage represented by the returned pointer by calling DA_FREESTRING. For example, “60 13:00:00”.

Note: The string format used by this function will change in a future release of ServiceCenter to be ISO-8601 compliant.

DA_SETTIMESTRING(PDATUM pd, char * pszTimeString)

Sets a DATUM pointed to by pd to the type “TIME” and gives it the value represented by pszTimeString. The string pointed to by pszTimeString must be formatted exactly the way that DA_GETTIMESTRING formats them, and must be expressed in GMT time. The null-terminated string pszTimeString is allocated by your plug-in and is not modified or freed by ServiceCenter. You are responsible for managing any storage represented by pszTimeString.

Note: The string format used by this function will change in a future release of ServiceCenter to be ISO-8601 compliant.

DA_SETCURRENTTIME(PDATUM pdFile)

Sets a DATUM pointed to by pd to the type “TIME” and gives it the current date and time at the ServiceCenter server.

char * DA_GETFILENAME(PDATUM pd)

Returns a pointer to a null-terminated character string containing the string value of the filename for the FILE DATUM pointed to by pdFile. Your program is responsible for passing this pointer to DA_FREESTRING when the character string is no longer needed. Failure to do this will cause a memory leak in the ServiceCenter server. See additional cautions described above under DA_GETSTRING.

char * DA_GETFIELDNAME(PDATUM pd, char* pszFieldRef)

Returns a pointer to a null-terminated character string containing the fieldname for the field pszFieldRef in the FILE represented by the FILE DATUM pointed to by pdFile. A field reference is a null-terminated string. The null-terminated string pszFieldRef is allocated by your plug-in and is not modified or freed by ServiceCenter. You are responsible for managing any storage represented by pszFieldRef.

Field references are traditional ServiceCenter symbolic and/or relative field names such as “header,number” or “header,1” or simply “1”. For files not containing structures or arrays, column names can be retrieved using either the simple column name, such as “number”, “description”, etc. or the relative field number in string form, such as “1”, “2”, “3”. Expressions such as "header,1" are only needed when the file contains arrays or structures such as "header" or "trailer" like the ServiceCenter problem file.

Your program is responsible for passing the returned pointer value containing the field name to DA_FREESTRING when the string is no longer needed. Failure to do this will cause a memory leak in the ServiceCenter server. See additional cautions described above under DA_GETSTRING.

PDATUM DA_GETFIELD(PDATUM pd, char* pszFieldRef)

Returns a pointer to the DATUM for the field represented by pszFieldRef in the FILE represented by the FILE DATUM pdFile. A field reference is a null-terminated string formatted as described above under DA_GETFIELDNAME. The null-terminated string pszFieldRef is allocated by your plug-in and is not modified or freed by ServiceCenter. You are responsible for managing any storage represented by pszFieldRef.

SC_LOGPRINT((printf style arguments))

Prints a line to the ServiceCenter log file. You use SC_LOGPRINT exactly as you would “printf”, **except** that you must supply an **extra set of parentheses** around everything passed to SC_LOGPRINT. This is necessary so that you can pass an arbitrary number of arguments to the macro. Macros do not support a variable number of arguments very well in C/C++. Also, a newline is automatically supplied.

Example of the use of SC_LOGPRINT:

SC_LOGPRINT(("Successful initialization for plugin %s", pszPluginName));

6 CHAPTER The Wizard Creation Tool

The Wizard Creation Tool lets implementers and administrators add wizards to modules within ServiceCenter that assist users with certain tasks. For example, using the Wizards Creation Tool you can define a wizard that assists users in adding contacts to a database. The Wizard Creation Tool was used to create several wizards that ship with this release of ServiceCenter.

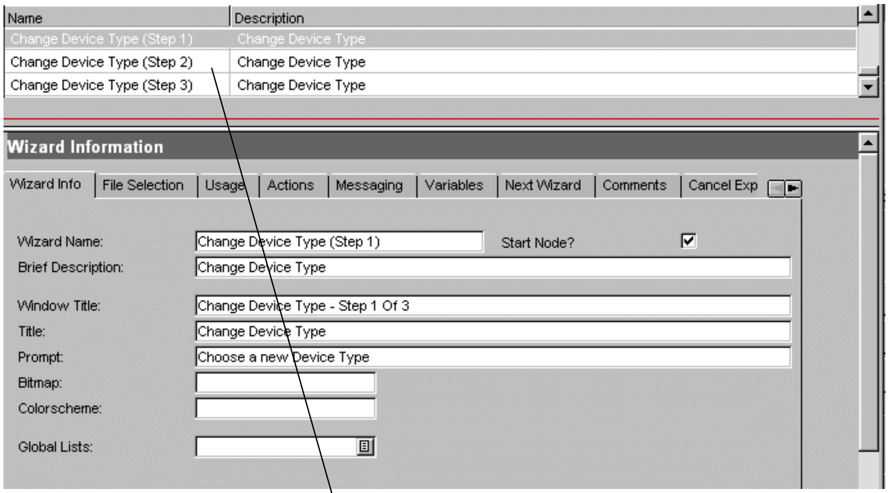
Accessing the Wizard Creation Tool

The Wizard Creation Tool is found under the Toolkit tab on the system administrator's main menu.

Creating a Wizard

To create a wizard, open the Wizard Creation Tool and fill in the fields that will define the wizard. Once you have finished defining the wizard, you can call the wizard from any existing format using display options or format control. For more information on the fields, see *Field Definitions* on page 112.

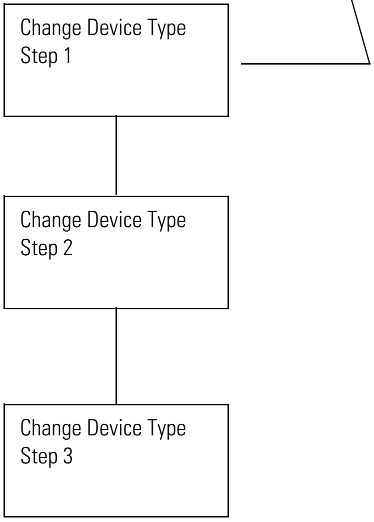
Each record within a wizard file represents a whole wizard. Define a series of wizards and string them together to present the user with a fully realized, end-to-end wizard.



Name	Description
Change Device Type (Step 1)	Change Device Type
Change Device Type (Step 2)	Change Device Type
Change Device Type (Step 3)	Change Device Type

Wizard Information

Wizard Name: Change Device Type (Step 1)
Start Node? ✓
Brief Description: Change Device Type
Window Title: Change Device Type - Step 1 Of 3
Title: Change Device Type
Prompt: Choose a new Device Type
Bitmap:
Colorscheme:
Global Lists:


Change Device Type Step 1
Change Device Type Step 2
Change Device Type Step 3

The Change device type wizard consists of three separate wizard definitions. End users see only one wizard that walks them through changing a device type.

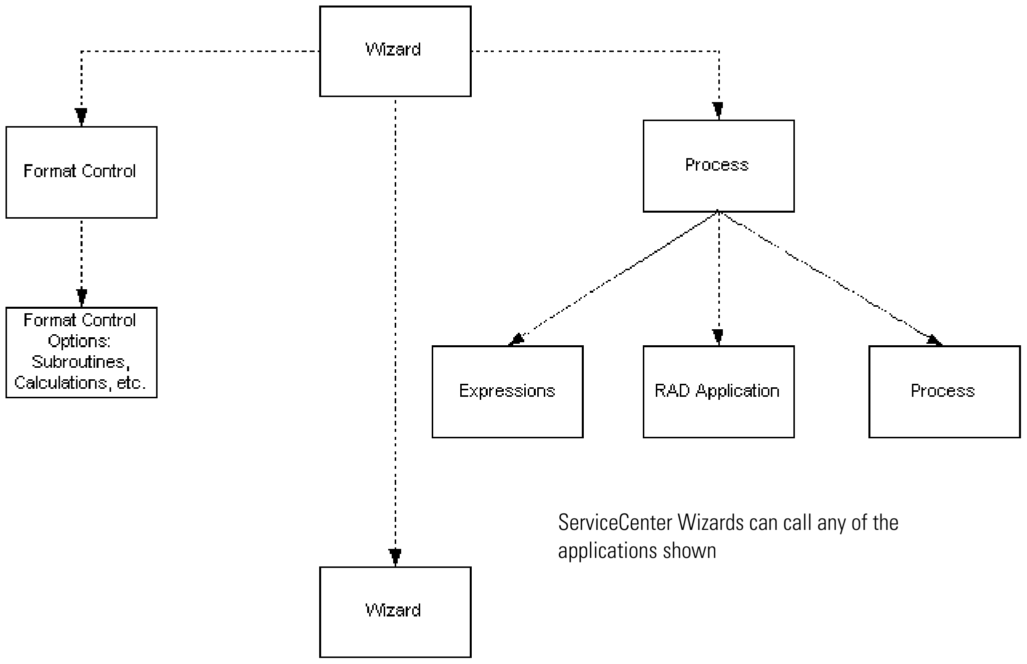
Calling a Wizard

The Wizard RAD application is wizard.run, and can be called from:

- Menus
- Display Options
- Format Control
- States
- Process Records

Wizards themselves can call Format Control records, Processes or other wizards, as shown below. Wizards do not allow direct calls to RAD applications.

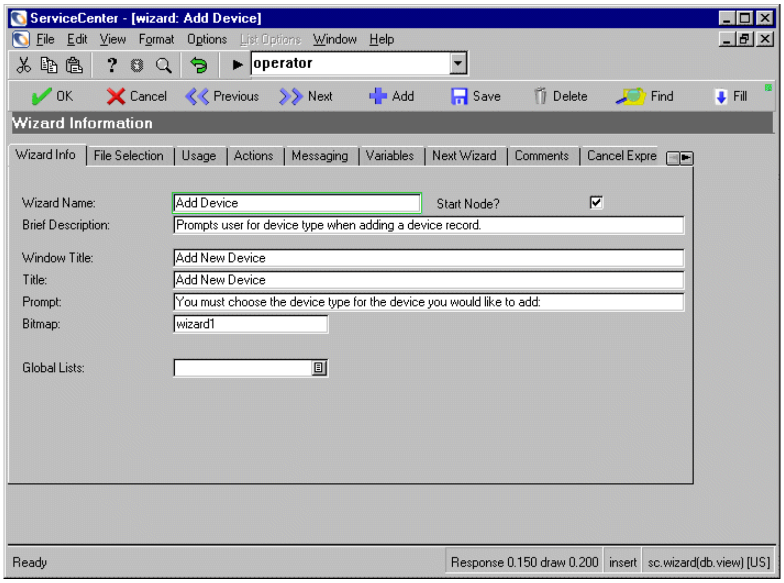
Note: It is possible to send a Wizard into an infinite loop by calling a Process that calls a Wizard that calls the original Process, for example.


Wizard
Format Control
Format Control Options: Subroutines, Calculations, etc.
Process
Expressions
RAD Application
Process
Wizard
ServiceCenter Wizards can call any of the applications shown

Field Definitions

Define wizards using the tabs on the Wizards panel. Each tab controls a different aspect of a Wizard. Not all fields on each tab are required.

Wizard Info Tab



Wizard Name:	Add Device
Start Node?	✓
Brief Description:	Prompts user for device type when adding a device record.
Window Title:	Add New Device
Title:	Add New Device
Prompt:	You must choose the device type for the device you would like to add:
Bitmap:	wizard1
Global Lists:	

Figure 6-1: The Wizard Info Tab

Wizard Name - Enter a brief, descriptive name for the wizard. This field forms the unique key for the wizard in the wizard's dbdict.

Start Node? - Select this check box if the wizard is the first in a series of wizards.

Brief Description- Enter a brief, meaningful description.

Window Title - Enter text that will display in the title bar of the wizard. This field will also take a message number from the message database. For example, scmsg(1,"am").

Note: If the wizard you are creating will be localized (translated to another language or languages) then using the scmsg database is the preferred method.

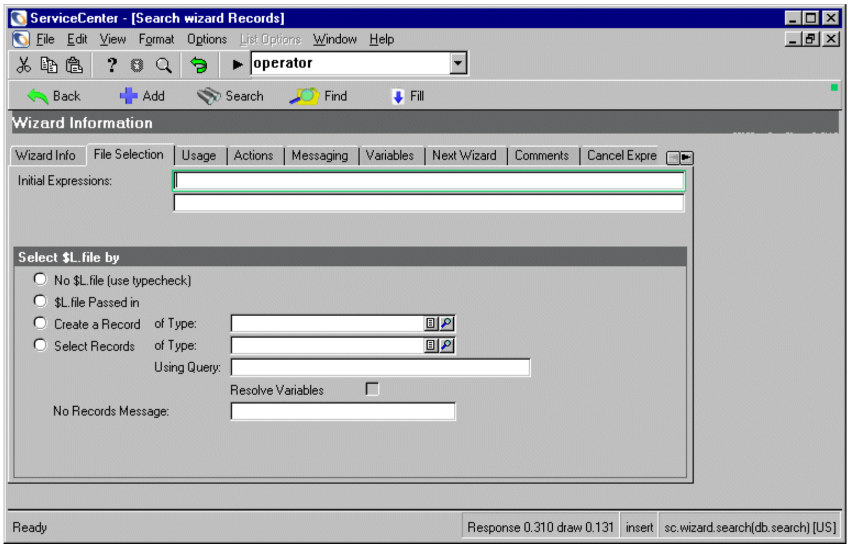
Title - Enter the title of the wizard in this field. This field will also take a message number from the message database. For example, scmsg(1,”am”). The Title appears on the upper left of the first page of the wizard, in bold.

Prompt - The Prompt field provides instructions for the user and appears in the center of the panel in large type. This field will also take a message number from the message database. For example, scmsg(1,”am”).

Bitmap - Add a bitmap to the left panel of the wizard. The file wizard1.bmp (located in the BITMAPS folder) defaults.

Global Lists - The lists you need to have loaded into memory when the wizard starts.

File Selection Tab


ServiceCenter - [Search wizard Records]
File Edit View Format Options List Options Window Help
operator
Back Add Search Find Fill
Wizard Information
Wizard Info File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expre

Initial Expressions:

Select $L.file by
- No $L.file (use typecheck)
- $L.file Passed in
- Create a Record of Type:
- Select Records of Type:

Using Query:

Resolve Variables

No Records Message:

Ready | Response 0.310 draw 0.131 | insert | sc.wizard.search(db.search) [US]

Figure 6-2: The File Selection Tab

Initial expressions - Enter expressions that will run before the wizard starts.

No $L.file (use typecheck) - If selected, this field initializes a typecheck file which acts as a holding file.

$L.file passed in - Selecting this option indicates the $L.file variable will be passed to the wizard from a previous wizard. You must initialize the variable in the Variables tab of the first wizard in the series.

Create a record - Selecting this option indicates the wizard will create a record. You must enter or select the record type in the **of type** field. This becomes $L.file.

- **of type** - Enter the type of record to be created. Click the fill button to select a record from a QBE list. Click the View button to display the dbdict for the record you have selected.

Select records - Select this option to have the user select a record from a list.

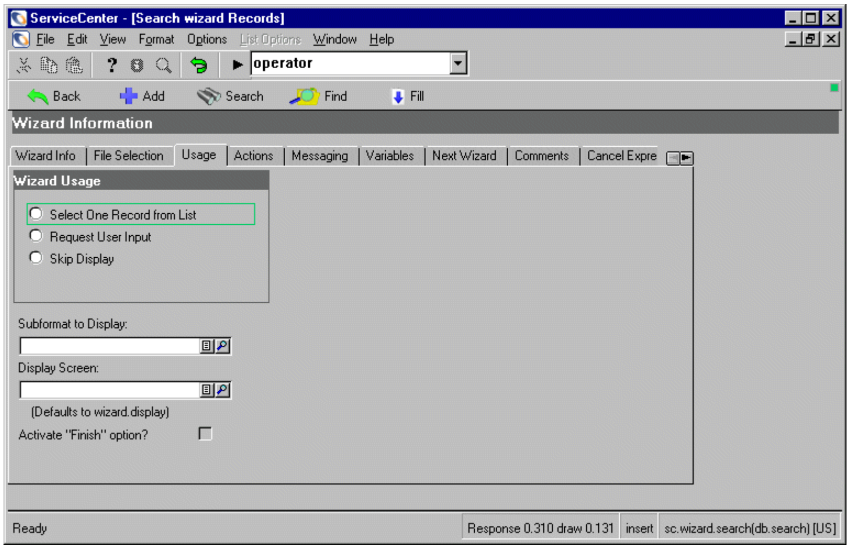
- **of type** - This field determines the record type for a query, for example Location. Enter the type of record or click the fill button to select a record from a QBE list of all records.

using query - Enter a query, using RAD syntax, to search for records.

Resolve variables - Selecting this box ensures that any variables entered in the Select Records field are evaluated before running the query.

No Records Message - Enter the text to display to the user if no records are returned. This field will also take a message number from the message database. For example, scmsg(42,”am”).

Usage Tab



Wizard Usage

- ○ Select One Record from List
- ○ Request User Input
- ○ Skip Display

Subformat to Display:

Display Screen:

(Defaults to wizard.display)

Activate "Finish" option? ☐

Ready | Response 0.310 draw 0.131 | insert | sc.wizard.search(db.search) [US]

Figure 6-3: The Usage Tab

Select one record from list - Selecting this option makes the **Selection Criteria** options available.

Files under Selection Criteria

Use $L.file as $L.selection - Selecting this options uses the $L.file variable as the selection criteria for a search. The $L.file variable must be initialized on the **File Selection** tab.

Query for Records - The Query for Records fields can be used in conjunction with one another.

- **of types** - This field determines the record type for a query, for example **Location**. Enter the type of record or click the fill button to select a record from a QBE list of all records.
- **using query** - Enter a query, using RAD syntax, to search for records.
- **sort by** - Use this field to determine what fields to sort returns by.

If No Records - What to display to the user if no records are available. Select an option from the list.

- **No Records Message** - Enter the text to display to the user if no records are returned. This field will also take a message number from the message database. For example, scmsg(42,"am").

If One Record - Determines what will happen when only one record is returned. Select the desired behavior from the list.

Allow "Skip" option? - Select this option to allow users to skip a panel.

Request user input - This field indicates that the user will be prompted for information.

Skip Display - Wizard runs without GUI component. This is used when the wizard simply does file manipulation or chaining wizards together.

Sub Format to Display - Enter a subformat name. `wizard.subformat` defaults.

Display Screen - Enter a display screen, if you want to customize the display options. The default display screen is `wizard.display`.

Activate "Finish" option? - Selecting this checkbox makes a **Finish** button appear on the wizard panel. Use this on the final wizard in the series.

Actions Tab

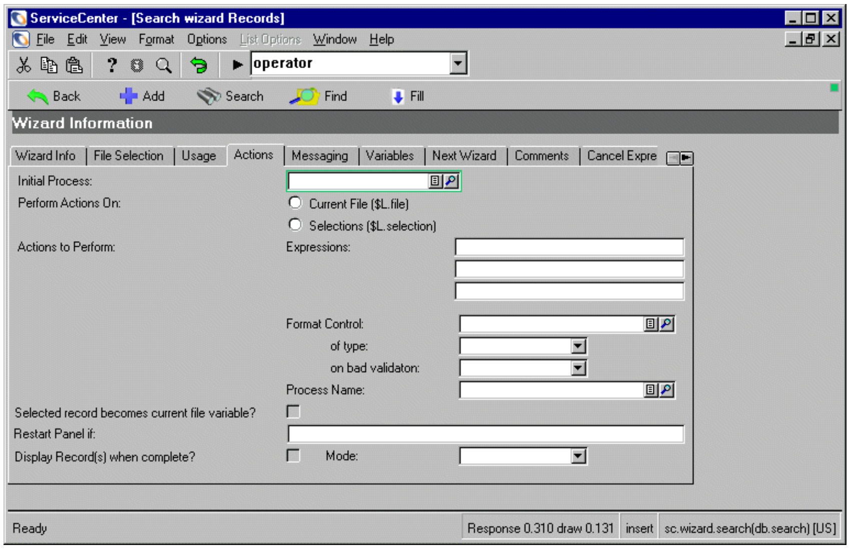

ServiceCenter - [Search wizard Records]
File Edit View Format Options List Options Window Help
operator
Back Add Search Find Fill
Wizard Information
Wizard Info File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expre
Initial Process:
Perform Actions On:
Current File ($L.file)
Selections ($L.selection)
Actions to Perform:
Expressions:
Format Control:
of type:
on bad validaton:
Process Name:
Selected record becomes current file variable?
Restart Panel if:
Display Record(s) when complete?
Mode:
Ready
Response 0.310 draw 0.131
insert
sc.wizard.search(db.search) [US]

Figure 6-4: The Actions Tab

Actions define what will happen when a user clicks **Next, Cancel or Previous** within a wizard, and are defined on the **wizard.display** Display Options file. Fields on this tab are executed in order.

Initial Process - The Initial Process is the first process to run when a user performs an action, such as clicking **Next**.

Perform actions On - The actions defined in the Actions to Perform block will apply to the file or files you select here. Choose either,

- **Current File** (**$L.file**) - performs the defined action on the current file only.
- **Selections** (**$L. selection**) - performs the defined actions on a selected group of records.

Actions to Perform:

- **Expressions** - Enter an action using ServiceCenter RAD syntax. The action defined here is applied to the files selected in the Perform Actions On fields.

Format Control - Enter a format control record or click either the Fill or View button to select a record from a QBE list

- **of type** - Select an action from the list that will trigger the format control record.
- **on bad validation** - Prompt for an action on bad validation. Return sends the user back to the panel, Continue lets the user continue.

Process name - The wizard you create can run a process after completing. Enter the process name here, click the fill button to select from a QBE list of available processes.

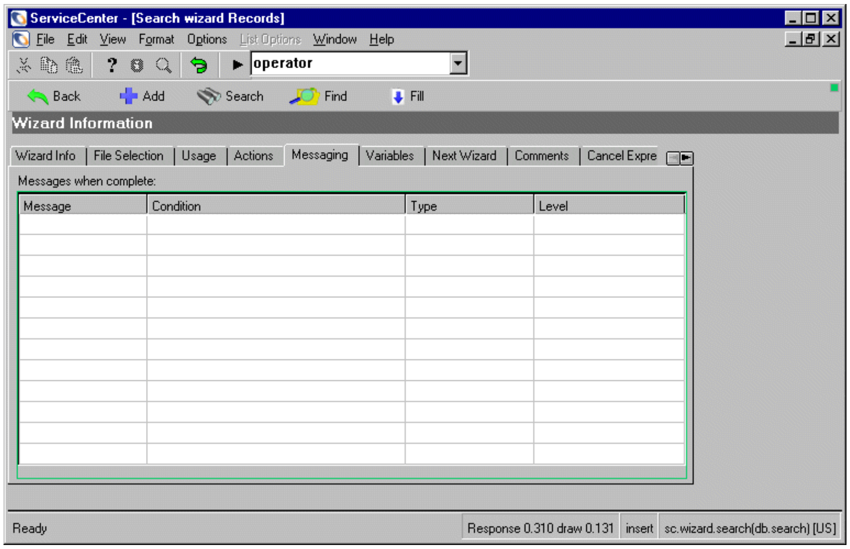
Replace Current File with Selections? - Checking this replaces the current $L.file variable with the selection list used by the wizard.

Restart Panel if: - Define the $L.return.action: yes/no/cancel. The parameter chosen indicates whether the user should go on or restart. For example, $L.return.action="no"

Display Records when complete? - If the user will be creating a record, entering True or an expression that evaluates to True will display the record to the user when the wizard completes.

Mode - This field determines how a record will display to the user if the Display Records when complete? field is selected. Browse defaults.

Messaging Tab


ServiceCenter - [Search wizard Records]
File Edit View Format Options List Options Window Help
operator
Back Add Search Find Fill
Wizard Information
Wizard Info | File Selection | Usage | Actions | Messaging | Variables | Next Wizard | Comments | Cancel Expre

Messages when complete:

Message	Condition	Type	Level

Ready | Response 0.310 draw 0.131 | insert | sc.wizard.search(db.search) [US]

Figure 6-5: The Messaging Tab

Message - Enter a message or message number from the scmsg database that will display when the wizard exits.

Condition - Enter a condition for the message, if desired. For example, not nullsub($L.finish, false)

Type - Select from the list how the message will display to users. The default is **On Screen**.

Level - Select from the list the level of activity that will be presented to users. The default level is **Info**.

Variables Tab

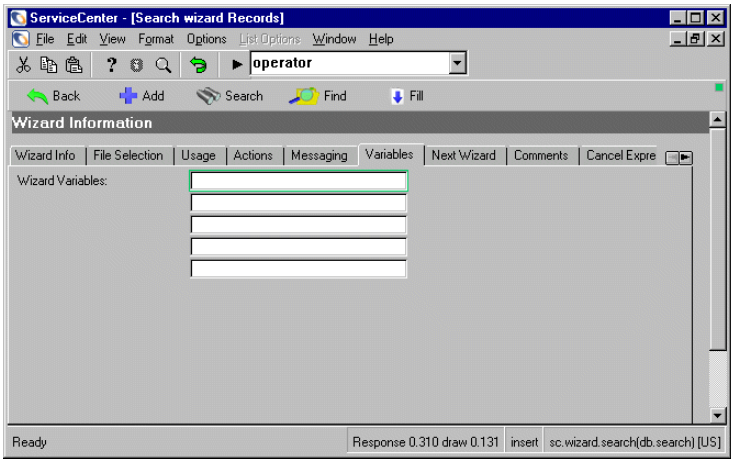

ServiceCenter - [Search wizard Records]
File Edit View Format Options List Options Window Help
operator
Back Add Search Find Fill
Wizard Information
Wizard Info File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expre
Wizard Variables:
Ready
Response 0.310 draw 0.131 insert sc.wizard.search(db.search) [US]

Figure 6-6: The Variables Tab

Wizard Variables - Use this tab to define variables for use within the wizard. Variables can be passed to applications or formats being created by the wizard, but must be assigned. For example, a variable such as `$L.return.action` must be defined here if it will be used within the wizard.

Next Wizard Tab

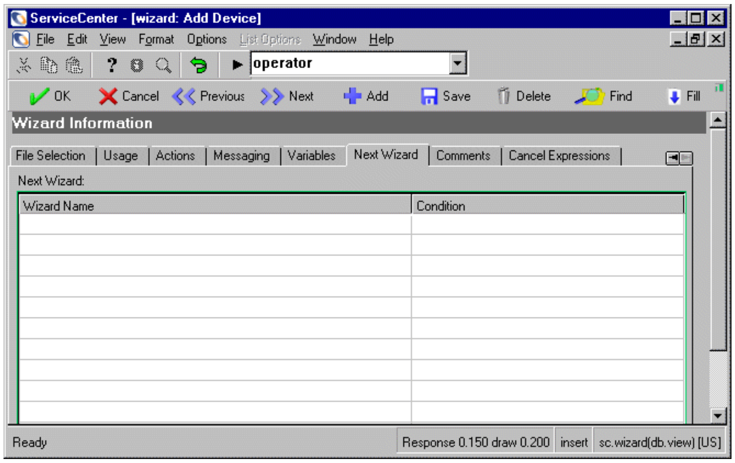

ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
operator
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expressions
Next Wizard:
Wizard Name
Condition
Ready
Response 0.150 draw 0.200 insert sc.wizard(db.view) [US]

Figure 6-7: The Next Wizard Tab

Wizard Name - Enter the name of the next wizard in the sequence.

Condition - Enter a condition that determines what happens next based on user input. For example, you could enter the next wizard in sequence and an expression that evaluates to true. Alternately, you could specify different wizards based on user actions or selections.

Comments Tab

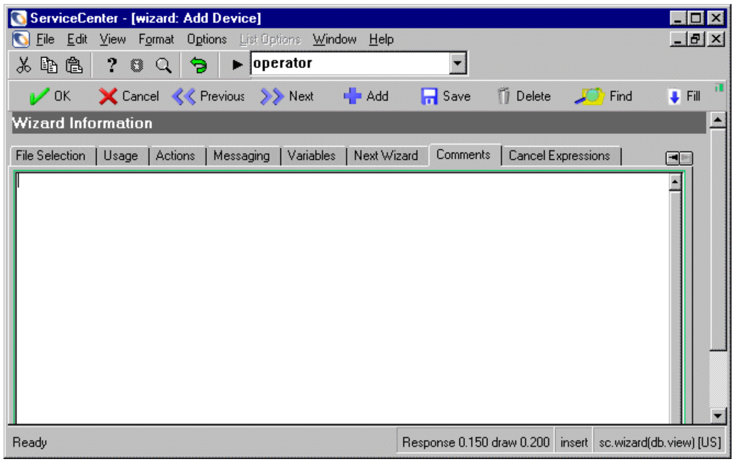

ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
operator
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expressions
Ready
Response 0.150 draw 0.200 insert sc.wizard(db.view) [US]

Figure 6-8: The Comments Tab

Enter any developer comments concerning the wizard here.

Cancel Expressions Tab

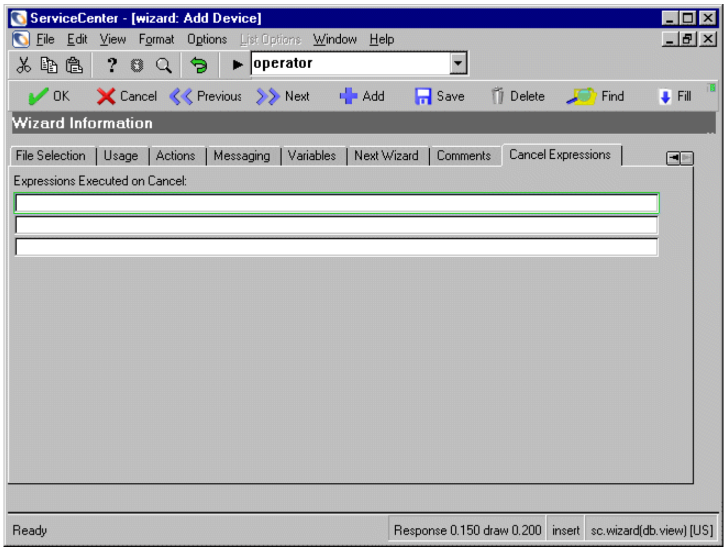

ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
operator
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
File Selection | Usage | Actions | Messaging | Variables | Next Wizard | Comments | Cancel Expressions
Expressions Executed on Cancel:
Ready
Response 0.150 draw 0.200 | insert | sc.wizard(db.view) [US]

Figure 6-9: The Cancel Expressions Tab

Enter any expressions that will execute on cancel here. This provides the wizard creator an opportunity to clean up variables initialized in the wizard, if desired.

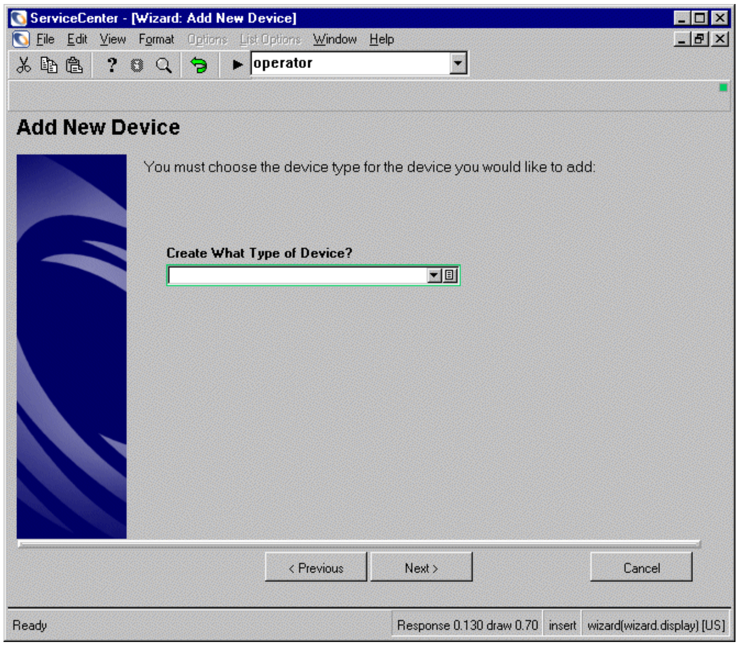
Sample Wizard: Add New Device

The Add New Device wizard is an out-of-box wizard. This is a simple one-panel wizard, which will be used as a sample to explain how a wizard is created.

To view the Add New Device Wizard:

1 From the home menu, select **Inventory Management** > **Assets.**

2 Click **New**.


ServiceCenter - [Wizard: Add New Device]
File Edit View Format Options List Options Window Help
operator
< Previous
Next >
Cancel

Add New Device

You must choose the device type for the device you would like to add:

Create What Type of Device?

Ready | Response 0.130 draw 0.70 | insert | wizard(wizard.display) [US]

Figure 6-10: The Add New Device wizard

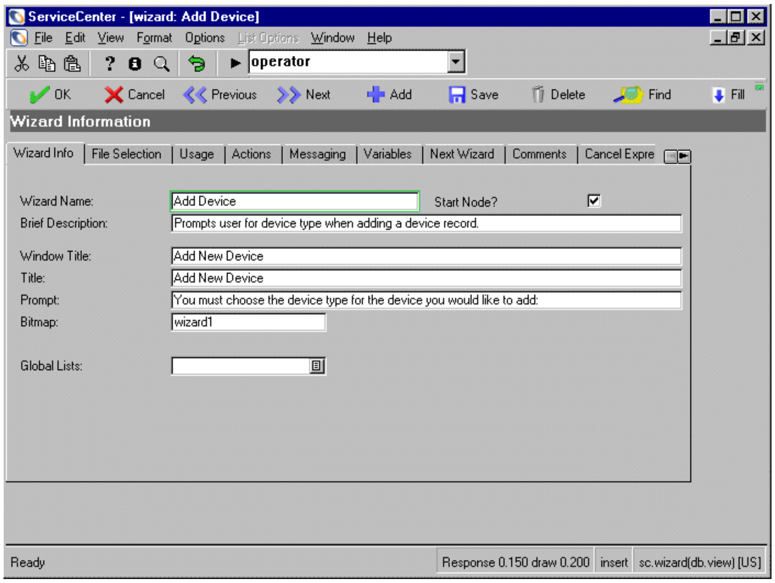
To view the Add New Wizard information file:

1 From the home menu, select the **Toolkit** tab > **Wizards**.

2 In the **Wizard Name** field, type `Add Device`.

3 Click **Search**.

4 Select **Add Device** from the record list.


ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
operator
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
Wizard Info File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expre

Wizard Name: Add Device
Start Node? ☑
Brief Description: Prompts user for device type when adding a device record.
Window Title: Add New Device
Title: Add New Device
Prompt: You must choose the device type for the device you would like to add:
Bitmap: wizard1
Global Lists:

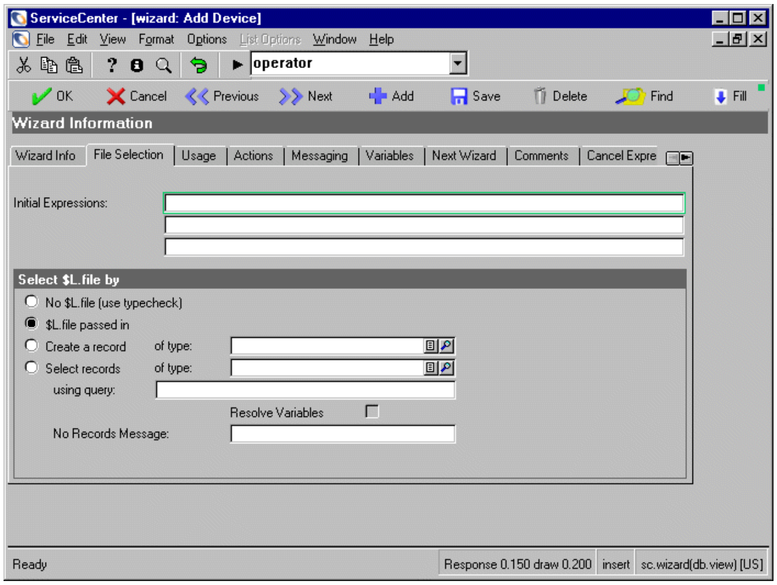
Ready | Response 0.150 draw 0.200 | insert | sc.wizard(db.view) [US]

Figure 6-11: The Add Device Wizard Information file

The Wizard Info tab details the wizard name, titles, prompts, and the bitmap to be displayed.

This wizard is defined as the “Start Node” since it is the only panel in the wizard. If this wizard had more than one panel, the “Start Node” check box would only be checked on the first panel.

5 Select the **File Selection** tab.


ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
operator
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
Wizard Info File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expre

Initial Expressions:

Select $L.file by
- No $L.file (use typecheck)
- [✓ $L.file passed in]
- Create a record of type:
- Select records of type:

using query:

Resolve Variables ☐

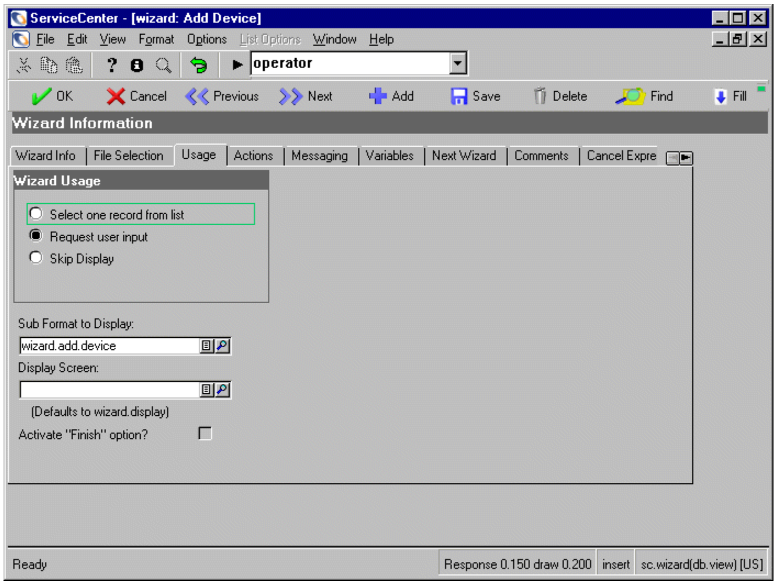
No Records Message:

Ready | Response 0.150 draw 0.200 | insert | sc.wizard(db.view) [US]

Figure 6-12: Sample wizard File Selection tab

If the current panel is the first panel, or the Start Node, $L.file is passed in from the user’s original location; in this sample, device. However, if it's a wizard panel that is called from another wizard panel then the $L.file variable will be passed to the wizard from a previous wizard.

1 Select the **Usage** tab.


ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
Wizard Info File Selection Usage Actions Messaging Variables Next Wizard Comments Cancel Expre

Wizard Usage

- ○ Select one record from list
- ● Request user input
- ○ Skip Display

Sub Format to Display: wizard.add.device

Display Screen:

(Defaults to wizard.display)

Activate "Finish" option? ☐

Ready | Response 0.150 draw 0.200 | insert | sc.wizard(db.view) [US]

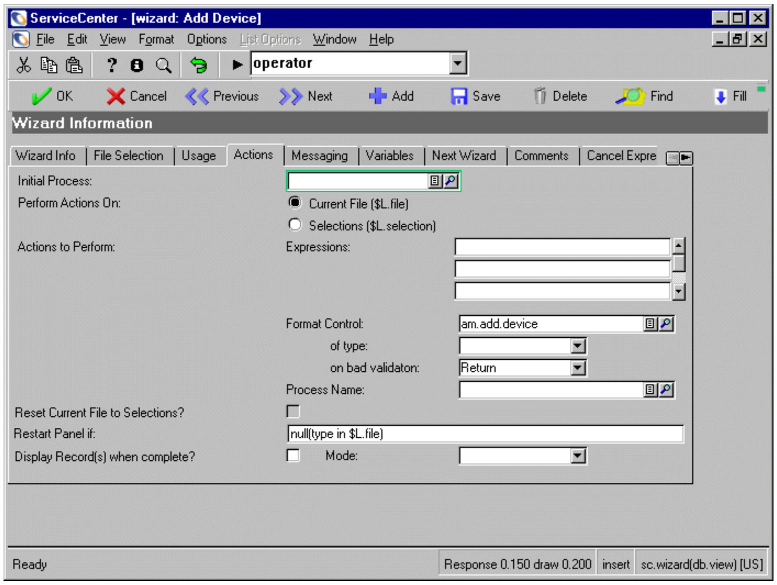
Figure 6-13: Sample wizard Usage tab

The radio button selected in the **Wizard Usage** area of the Usage tab indicates that user input is required.

The Sub Format to display is **wizard.add.device**. This indicates what format will be displayed for the user.

Note: Use Forms Designer to create a form, or use an existing form. Once you have created a form, enter the name of the form in the **Sub Format to Display** field. For more information on Forms Designer, see *System Tailoring, vol.1*.

2 Select the **Actions** tab.


ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
operator
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
Wizard Info | File Selection | Usage | Actions | Messaging | Variables | Next Wizard | Comments | Cancel Expre

Initial Process:
Perform Actions On: (● Current File ($L.file)) ○ Selections ($L.selection)
Actions to Perform: Expressions:
Format Control: am.add.device
of type:
on bad validaton: Return
Process Name:
Reset Current File to Selections? ☐
Restart Panel if: null(type in $L.file)
Display Record(s) when complete? ☐ Mode:

Ready | Response 0.150 draw 0.200 | insert | sc.wizard(db.view) [US]

Figure 6-14: Sample wizard Actions tab

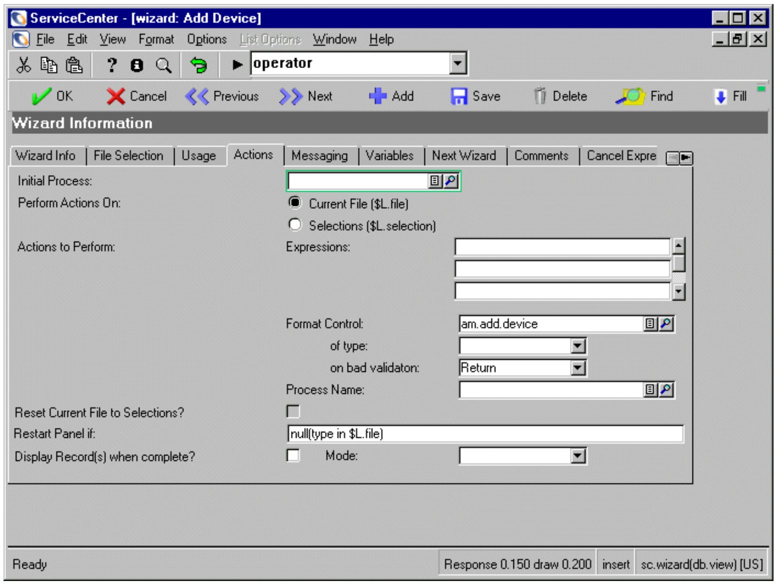
When the user presses the **Next** button, the **am.add.device** *format control* is called, and the **check.devtype.restrictions** *process* is called.

The "Return" in the **on bad validation** field indicates that the user will be returned to the wizard if any validations fail.

If the "type in $L.file" is NULL or $L.restricted=true, the user will be returned to the wizard and will not be allowed to continue.

Note: You can use existing Process or Format Control records, or you can create new Process and/or Format Control records. For more information on Format Control, see *System Tailoring, vol. 1*. For more information on Process records, see *System Tailoring, vol. 2*.

3 Select the **Variables** tab.


ServiceCenter - [wizard: Add Device]
File Edit View Format Options List Options Window Help
operator
OK Cancel Previous Next Add Save Delete Find Fill
Wizard Information
Wizard Info | File Selection | Usage | Actions | Messaging | Variables | Next Wizard | Comments | Cancel Expre

Initial Process:
Perform Actions On: [● Current File ($L.file)] ○ Selections ($L.selection)
Actions to Perform:
Expressions:
Format Control: am.add.device
of type:
on bad validaton: Return
Process Name:
Reset Current File to Selections? ☐
Restart Panel if: null(type in $L.file)
Display Record(s) when complete? ☐ Mode:

Ready | Response 0.150 draw 0.200 | insert | sc.wizard(db.view) [US]

Figure 6-15: Sample wizard Variables tab

All local variables ($L.variables) that are referenced anywhere besides the wizard itself must be defined on the Variables tab. Local variables must be defined if they are referenced anywhere in the Process, Sub Format, or Next Wizard forms.

7 Macro Editor

CHAPTER

ServiceCenter macros are discrete units of work a system administrator can invoke to do things like send email to a specific address, or page a specific phone number. ServiceCenter macros are more similar to Microsoft Access macros than to Microsoft Word macros, which simply record and play back keystrokes.

Macros are distinct actions, driven by predefined conditions, that are executed when a record is saved in the database. Macro actions are associated with files and reflect certain states in the records of those files. If a macro's condition evaluates to *true* when a record is saved, the macro's action is executed. A typical condition might be **priority.code in $L.new= "1"**, causing that macro's action to be executed when an incident ticket being saved has a priority code of *1*.

As a ServiceCenter administrator, you can create macros to run processes automatically when specified events occur. For example, you can create a macro to page a manager when an incident ticket hits a DEADLINE ALERT.

Macro conditions

Macro conditions are expressions, written in ServiceCenter RAD syntax, which are evaluated at run time. If the expression evaluates to *true*, the macro is executed. If the expression does not evaluate to true, the macro cannot proceed.

A macro's condition can be very simple, e.g., **true** or **tod**() <= '**17:00:00**'. Often, macro conditions include a check against the record being saved. For example, a macro condition can be expressed for an action or group of actions when a specific set of incident tickets is saved. Another set of tickets may require different actions altogether, requiring new macro conditions.

The record currently being saved is identified to macro conditions as **$L.new.** This variable can be used as **$file** is used in Format Control expressions. Macro expressions, however, also have **$L.old** available to them. This represents the state in which the record existed before it was altered. All the following expressions are valid:

- **priority.code in $L.new="1"**

Fires whenever a priority 1 ticket is saved.

- **assignment in $L.new~=assignment in $L.old**

Fires whenever the assignment group changes.

- **tod**() <= '17:00:00'

Fires whenever a ticket is saved before 5:00 PM.

The variable **$L.message** can be used to create *evaluating expressions* that gather certain information about incident tickets. This data is then sent as a message to specific users or groups defined in the Macro Parameters form (Figure 7-3 on page 135). **$L.message** is expressed as an array, using the following syntax:

```
$L.message={"Incident#" +number in $L.new, brief.description in $L.new}
```

The result is an array of the Incident ID and Incident Title in the record being saved (**$L.new**). The message might look like this:

> *Incident # IM1012*
>
> *Phone is going dead intermittently.*

Accessing Macro Records

You may access macros from either of the following locations:

- Tools menu in the **Utility** tab of the system administrator's home menu.
- Incident Management Security Files.

Accessing macros through the Utilities tab is discussed in this section. Either method is equally effective and displays the same search form (*macro.lister.g*)

Warning: Do not edit macros through the Database Manager. Certain processing does not occur, and your edits may not be saved.

You may select individual macro records from a list or search for groups of related macros (e.g., those associated with the problem file).

Note: You need **SysAdmin** privileges to run the macro editor.

To access a macro record:

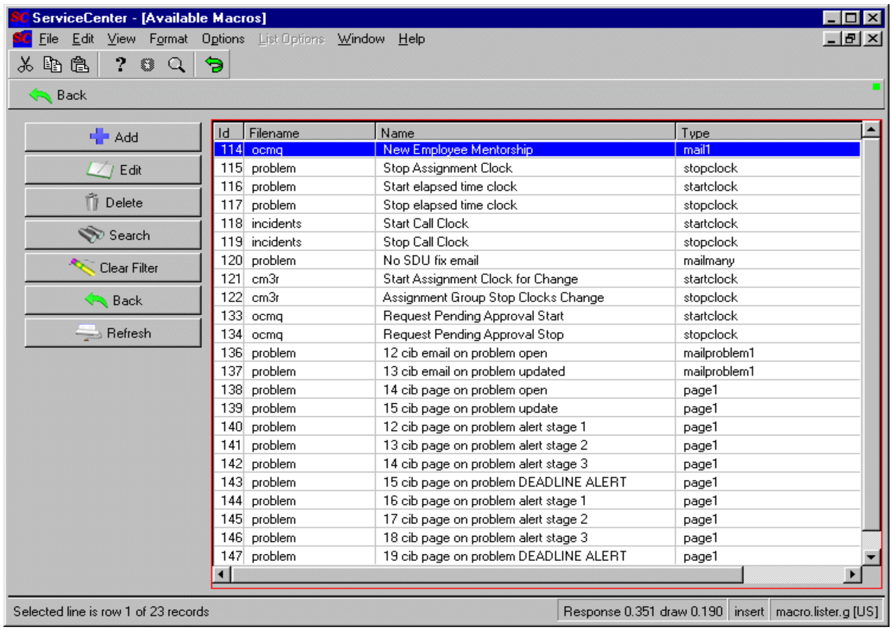
1 Select the **Utilities** tab in the system administrator's home menu.

2 Click **Tools**.

The Tools menu is displayed.

3 Click **Macros**.

The Macro List form is displayed (Figure 7-1 on page 132). All available macros in your system appear in the initial list.


ServiceCenter - [Available Macros]
File Edit View Format Options List Options Window Help
Back
Add
Edit
Delete
Search
Clear Filter
Back
Refresh

Id	Filename	Name	Type
114	ocmq	New Employee Mentorship	mail1
115	problem	Stop Assignment Clock	stopclock
116	problem	Start elapsed time clock	startclock
117	problem	Stop elapsed time clock	stopclock
118	incidents	Start Call Clock	startclock
119	incidents	Stop Call Clock	stopclock
120	problem	No SDU fix email	mailmany
121	cm3r	Start Assignment Clock for Change	startclock
122	cm3r	Assignment Group Stop Clocks Change	stopclock
133	ocmq	Request Pending Approval Start	startclock
134	ocmq	Request Pending Approval Stop	stopclock
136	problem	12 cib email on problem open	mailproblem1
137	problem	13 cib email on problem updated	mailproblem1
138	problem	14 cib page on problem open	page1
139	problem	15 cib page on problem update	page1
140	problem	12 cib page on problem alert stage 1	page1
141	problem	13 cib page on problem alert stage 2	page1
142	problem	14 cib page on problem alert stage 3	page1
143	problem	15 cib page on problem DEADLINE ALERT	page1
144	problem	16 cib page on problem alert stage 1	page1
145	problem	17 cib page on problem alert stage 2	page1
146	problem	18 cib page on problem alert stage 3	page1
147	problem	19 cib page on problem DEADLINE ALERT	page1

Selected line is row 1 of 23 records | Response 0.351 draw 0.190 | insert | macro.lister.g [US]

Figure 7-1: ServiceCenter Macro List Form

The *macro list* form is the access point for all your macro activities. You must go through this form to add, edit, or delete macro records. The results of macro queries are displayed in the macro list form. page 136 for column header and option button definitions. Option buttons in this form provide controls for viewing and processing macros.

4 To open an individual record (Figure 7-2 on page 133):

a Select an item in the list.

b Click Edit.

5 To display a list of related records:

a Click Search.

The macro search form is displayed.

b Enter search information in any field.

- **ID**—the number identifying the macro. This number is assigned by the system.
- **Filename**—the ServiceCenter file associated with the macros.
- **Name**—the unique name of the macro.
- **Type**—the macro type for the macro(s) for which you are searching.

c Click OK.

The system displays the macro list form showing all the macros matching your search criteria.

d Select an entry to display.

e Click Edit

The selected record is displayed in the macro editor (Figure 7-2 on page 133).

Edit existing macros and create new ones in the *macro editor*. Select macro types and set conditions for their execution. The values selected in this form determine which fields are displayed in the parameter form.

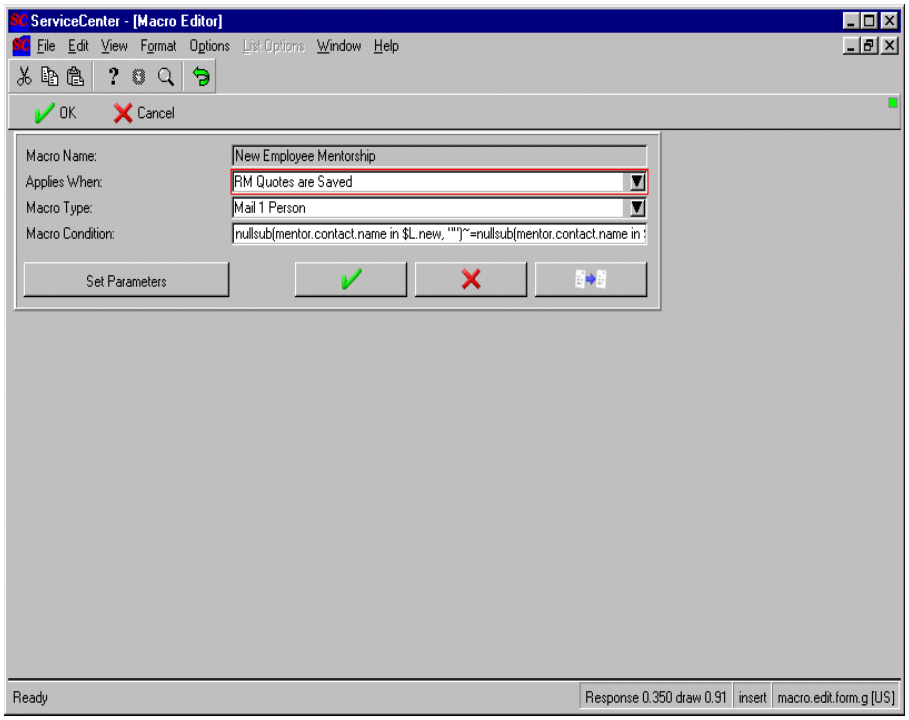

ServiceCenter - [Macro Editor]
File Edit View Format Options List Options Window Help
OK
Cancel
Set Parameters
Macro Name: New Employee Mentorship
Applies When: RM Quotes are Saved
Macro Type: Mail 1 Person
Macro Condition: nullsub(mentor.contact.name in $L.new, "")~=nullsub(mentor.contact.name in
Ready Response 0.350 draw 0.91 insert macro.edit.form.g [US]

Figure 7-2: Macro Editor

Creating a Macros

When creating a macro, you must name and define the conditions of the macro before setting the parameters for its execution.

To create a macro:

1 Select the **Utilities** tab of the system administrator's home menu.

2 Click **Tools**.

3 Click **Macros**.

4 Click **Add**.

5 .Enter a name for the new macro in the **Macro Name** field.

For the example, enter `test`.

6 In the **Applies When** field, select an event option from the drop-down list indicating when you want the macro to be executed.

For the example, select `Incidents are Saved`.

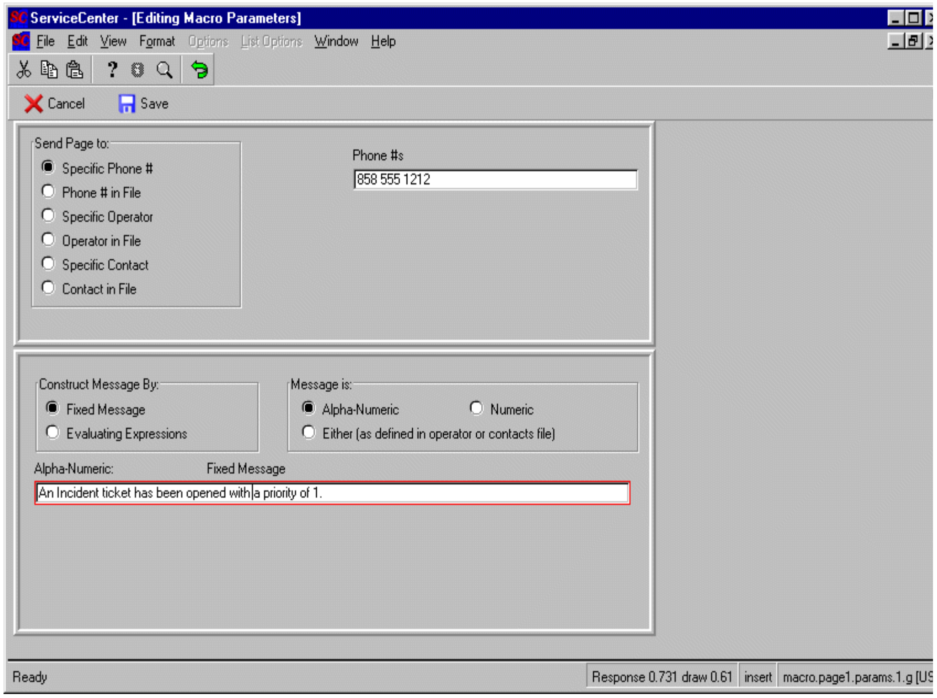
7 In the **Macro Type** field, select an action you want the macro to execute. Options include faxing, paging, mailing, starting and stopping clocks, executing a RAD function or evaluating an expression.

For this example, select `Page 1 Person`.

8 Enter a **Macro Condition** that will trigger the macro to execute. When this condition evaluates to *true*, ServiceCenter executes what is defined in the **Macro Type** field.

For example, you could have a specific person paged when a new incident ticket is set to *priority 1*. For the example, enter `priority in $L.new="1"`.

9 Click **Set Parameters** to establish the parameters for this macro.


ServiceCenter - [Editing Macro Parameters]
File Edit View Format Options List Options Window Help
Cancel
Save

Send Page to:
(● Specific Phone #) Phone # in File, Specific Operator, Operator in File, Specific Contact, Contact in File

Phone #s: 858 555 1212

Construct Message By:
(● Fixed Message) Evaluating Expressions

Message is:
(● Alpha-Numeric) Numeric, Either (as defined in operator or contacts file)

Alpha-Numeric: Fixed Message
An Incident ticket has been opened with a priority of 1.

Ready | Response 0.731 draw 0.61 | insert | macro.page1.params.1.g [US

Figure 7-3: Parameter Edit Form

Note: The available fields in this form vary depending on the value in the **Macro Type** field in the edit form for your new macro. The example in Figure 7-3 on page 135 displays a parameter form to **Page 1 Person**.

10 Provide additional information where needed, e.g., *Send Page to—Specific Phone, Specific Contact, Specific Operator*; *Construct Message By*.

11 Click **Save** or press F2.

You are returned to the macro edit form.

12 Click **OK**.

The macro record is saved and you are returned to the macro list form.

Updating a macro record uses the same edit forms as the creating process.

13 To refresh the list of macros, click **Search** from the macro list form, then click OK in the search dialog.

Definitions for Macro Forms

The following definitions can be found in the tables below:

- Macro list form
  - Column headers
  - Option buttons
- Macro editor
  - Fields

Macro list form

Column Label	Database Field Name	Description
Id	id	A unique number assigned to the macro to identify it.
Filename	filename	The ServiceCenter file to which the macro is attached, e.g., `problem`, `device`, etc.
Name	name	The name you give the macro.
Type	type	The type of action the macro takes when activated.

Button Label	Description
Add	Opens a blank macro editor form for adding a new macro.
Edit	Accesses the macro editor to change the selected macro record.
Delete	Deletes the selected macro record. **Warning!** *When deleting a record, no warning is displayed; the record is simply deleted.*
Search	Accesses a query form.
Clear Filter	Removes the current filter used for querying the macros and returns the list to its previous state.
Back	Returns to the previous process.

Macro editor

Field Label	Database Field Name	Description
Macro Name	name	Unique name for this macro provided by the system administrator who created it.
Applies When	filename	Predefined event option for execution of this macro. Select an event from the drop-down list (e.g., when a change request is saved).
Macro Type	type	Predefined macro type for this macro. The names and definitions of predefined macro types in ServiceCenter can be found in the table on page 138.
Macro Condition	condition	Define a condition under which this macro should execute. For example, when a *priority 1* incident ticket is opened, the members of an assignment group are paged.

Macros Provided with ServiceCenter

Macro Action	Description
Call A RAD Routine *	Executes a user-specified RAD routine and passes it parameters every time it executes. **Warning:** Macros do not work properly when calling a RAD application involving user interaction (e.g., **fill.recurse** or **validate.fields**). Continue to use Format Control to call these types of applications.
Evaluate Expressions	Executes a number of user-defined expressions whenever it fires.
Fax 1 Person	Sends a fax to one person only. This person can be defined as an operator, a contact, or a simple "raw" phone number.
Fax 1 Assignment Group	Sends a fax to an entire assignment group.
Fax Many People	Sends a fax to an arbitrarily defined list of people. These people can be defined as operators, contacts, or "raw" phone numbers.
Fax A Incident Ticket to 1 Person	Sends a fax of an incident ticket to one person only. This person can be defined as an operator, a contact, or a simple "raw" phone number.
Fax a Incident Ticket to A CM Message Group	Sends a fax of an incident ticket to all the members of a Change Management Message Group.
Fax A Incident Ticket to An Assignment Group	Sends a fax of an incident ticket to an entire assignment group.
Fax An Incident Ticket to Many People	Sends a fax of an incident ticket to an arbitrarily defined list of people. These people can be defined as operators, contacts, or "raw" phone numbers.
Get a Sequential Number	Fetches the next available sequential number of a specific class and stores it in a field in a file.
Mail 1 Person	Sends e-mail to one person only. This person can be defined as an operator, a contact, or a simple "raw" email address.
Mail A Change Request to 1 Person	Sends e-mail of a change request to one person only. This person can be defined as an operator, a contact, or a simple "raw" email address.
Mail A Change Request to A Message Group	Sends e-mail of a change request to all the members of a Change Management Message Group.

Macro Action	Description
Mail A Change Request to Many People	Sends an e-mail of a change request to an arbitrarily defined list of people. These people can be defined as operators, contacts, or “raw” phone numbers.
Mail a Change Task to 1 Person	Sends e-mail of a change task to one person only. This person can be defined as an operator, a contact, or a simple “raw” email address.
Mail a Change Task to Many People.	Sends an e-mail of a change task to an arbitrarily defined list of people. These people can be defined as operators, contacts, or “raw” phone numbers.
Mail 1 Assignment Group	Sends e-mail to an entire assignment group.
Mail Many People	Sends e-mail to an arbitrarily defined list of people. These people can be defined as operators, contacts, or “raw” email addresses.
Mail An Incident Ticket to 1 Person	Mails an incident ticket to one person. This person can be defined as an operator, a contact, or a simple “raw” email address.
Mail An Incident Ticket to An Assignment Group	Mails an incident ticket to all the members of an assignment group.
Mail An Incident Ticket to Many People	Mails an incident ticket to an arbitrarily defined list of people. These people can be defined as operators, contacts, or “raw” email addresses.
Mail An Incident Ticket to a CM Message Group	Mails an incident ticket to all the members of a Change Management Message Group.
Mail A Task to A Message Group	Mails a change task to all the members of a Change Management Message Group.
Page 1 Person	Sends a page to one person only. This person can be defined as an operator, a contact, or a simple “raw” phone number.
Page a CM Message Group	Sends a page to all the members of a Change Management Message Group
Page an Assignment Group	Sends a page to all the members of an assignment group.
Page Many People	Sends a page to an arbitrarily defined list of people. These people can be defined as operators, contacts, or “raw” phone numbers.
SC Mail 1 Person	Sends ServiceCenter mail to one person only. This person can be defined as an operator, a contact, or a simple “raw” phone number
SC Mail An Incident Ticket to Many People	Sends an incident ticket to an arbitrarily defined list of people with ServiceCenter mail. These people can be defined as operators, contacts, or “raw” email addresses.
SC Mail An Incident Ticket to a CM Message Group	Sends an incident ticket to all the members of a Change Management Message Group with ServiceCenter mail.

Macro Action	Description
SC Mail Many People	Sends ServiceCenter mail to an arbitrarily defined list of people. These people can be defined as operators, contacts, or “raw” phone numbers.
Start A Clock	Starts a specified clock.
Stop A Clock	Stops a specified clock

8 Development Audit Utility

CHAPTER

The development audit utility tracks changes made to ServiceCenter records during the development phase of ServiceCenter implementation. Whether you are making a few changes or extensively customizing your system, it is critical you have a record of the changes (*deltas*) to ensure loading of the correct version when you move to production.

The Dev Audit utility tracks changes to the following files:

application	format
formatctrl	link
scripts	Object
Process	States
trigger	code
eventregister	eventmap
eventfilter	datadict
dbdict	displayscreen
displayoption	

Development Auditor Menu and Functions

To access the Development Auditor Menu:

1. Select the **Utilities** tab of the system administrator's home menu.
2. Click **Development Auditing.** The Development Auditor Menu is displayed.
3. Select a function from the Audit menu.

Turn Auditing On/Off

This function allows you to enable or disable file auditing. With auditing turned on, a separate audit file is created each time you make a change to a file and save that changed file to the database.

ServiceCenter is delivered with a default set of audited files in place to enable you to track development changes.

To access the Audit Control form:

1. Click **Turn Auditing On/Off in the Development Auditor Menu.**

   The Audit Control record is displayed.
2. Update one or both of the following functions:

   Do you want to audit development changes?—enable/disable auditing of development changes.

   Do you want to keep backups of Changes?—enable/disable backups of development changes.
3. Click **Save** to save your changes.

   The following message is displayed in the status bar: *Record updated in the devauditcontrol file.*
4. Click **OK** to return to the Development Auditor Menu.

View Audit History

From the Audit History form, you can search for a particular audit record by entering specific data about that audit, or you can view a record/QBE list of all audits stored in the database. It is important to look at this file before purging audit records or unloading a development change to production.

To access the Audit History form:

1. Click **View Audit History**.

A blank Audit History record is displayed.

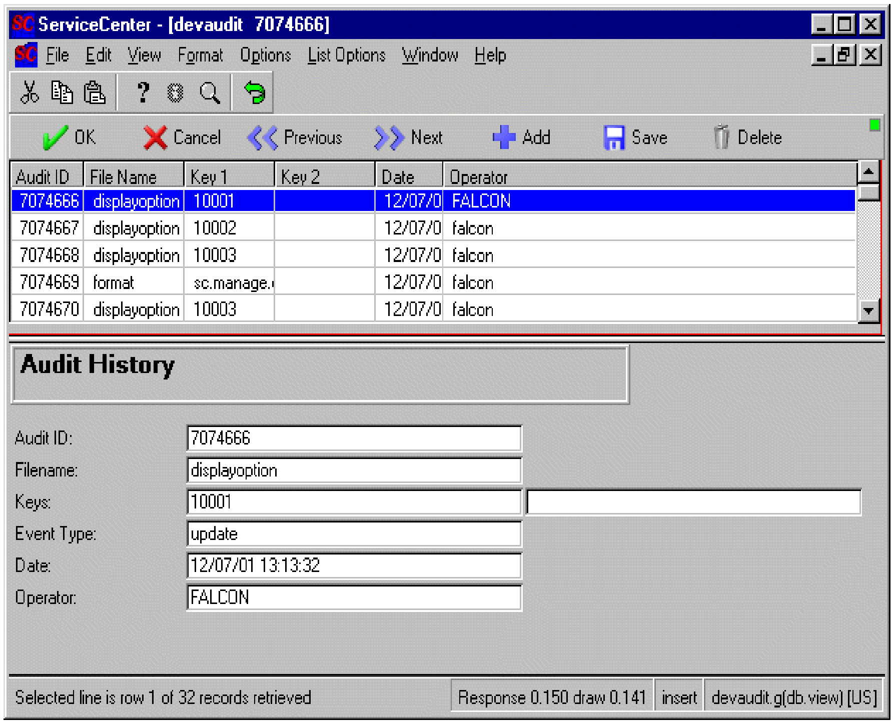
2 Open an existing Audit record using one of the following procedures:

- Enter any information you have from the record and click **Search** or press **Enter**.

  —or—

- Click **Search** or press **Enter.**

  If more than one record matches the search criteria, the returned screen is split with a record list at the top and the first record in the list displayed below.


ServiceCenter - [devaudit 7074666]
File Edit View Format Options List Options Window Help
OK
Cancel
Previous
Next
Add
Save
Delete

Audit ID	File Name	Key 1	Key 2	Date	Operator
7074666	displayoption	10001		12/07/0	FALCON
7074667	displayoption	10002		12/07/0	falcon
7074668	displayoption	10003		12/07/0	falcon
7074669	format	sc.manage.		12/07/0	falcon
7074670	displayoption	10003		12/07/0	falcon

Audit History

Audit ID:	7074666
Filename:	displayoption
Keys:	10001
Event Type:	update
Date:	12/07/01 13:13:32
Operator:	FALCON

Selected line is row 1 of 32 records retrieved | Response 0.150 draw 0.141 | insert | devaudit.g(db.view) [US]

Figure 8-1: Audit History Record

Note: There will be an entry for each time that a form or file was added or updated.

In the example above, the **problem** form is listed with Audit ID 1082514 (when the form was created) and again with Audit ID 1082516 (when it was updated). These entries were added automatically to the record/QBE list each time the form was added or saved.

3 Click the entry you want to view.

Information for this record is entered into the Audit History form.

4 Click Delete to remove the current record from the audit list.

Audit History fields

Audit ID—an identification number added automatically when an audit record is created.

Filename—file type, e.g. format, formatctrl, or link.

Keys—name of a key that can be used to search for an audit record, e.g., the form name.

Event Type—type of action performed on a file when the audit record was created, e.g., *add* (when a new file is created) or *update* (when a file is modified).

Date—time and date when the audit record was created. The format is **mm/dd/yy hh:mm:ss.**

Operator—login name used when the audit record was created.

Unload an Audit Delta

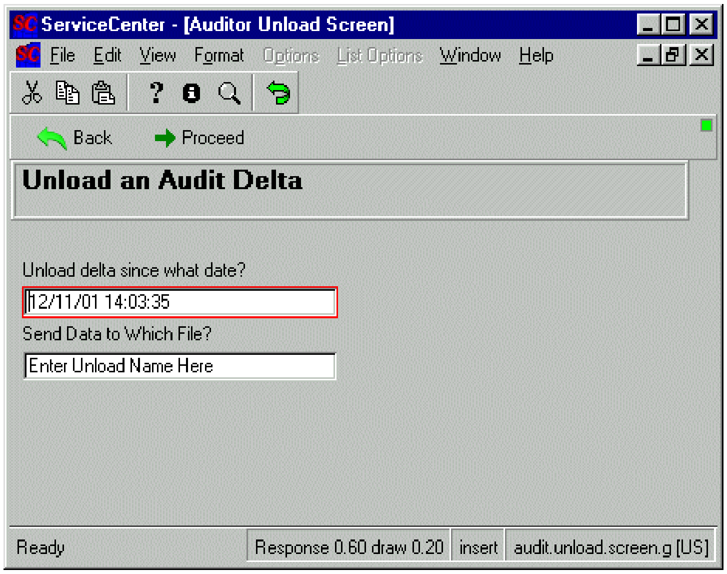
If you wish to specify what the system will be unload, you must first view the audit history. You can remove records from the development audit list by selecting them and then deleting them. See *View Audit History* on page 142.

Important: Before unloading a change you have made during the development phase, it is critical that you check the audit files for the correct date of this delta. See *View Audit History* on page 142. You will need the date shown in the Audit History form to enter into the Unload form.

To move a change to production with the Unload function:

1 Click **Unload an Audit Delta** on the Development Auditor Menu.

The Audit Unload form is displayed.



Unload an Audit Delta

Unload delta since what date?
12/11/01 14:03:35

Send Data to Which File?
Enter Unload Name Here

Ready | Response 0.60 draw 0.20 | insert | audit.unload.screen.g [US]

Figure 8-2: Audit Unload Form

The current date and time are automatically entered in the **Unload delta since what date?** field.

2 Enter the beginning date for the unload (found in the View Audit History record/QBE list.

3 Enter the path and name of the file to which you want to send the unloaded data in the **Send Data to Which File?** field.

4 Click **Proceed**.

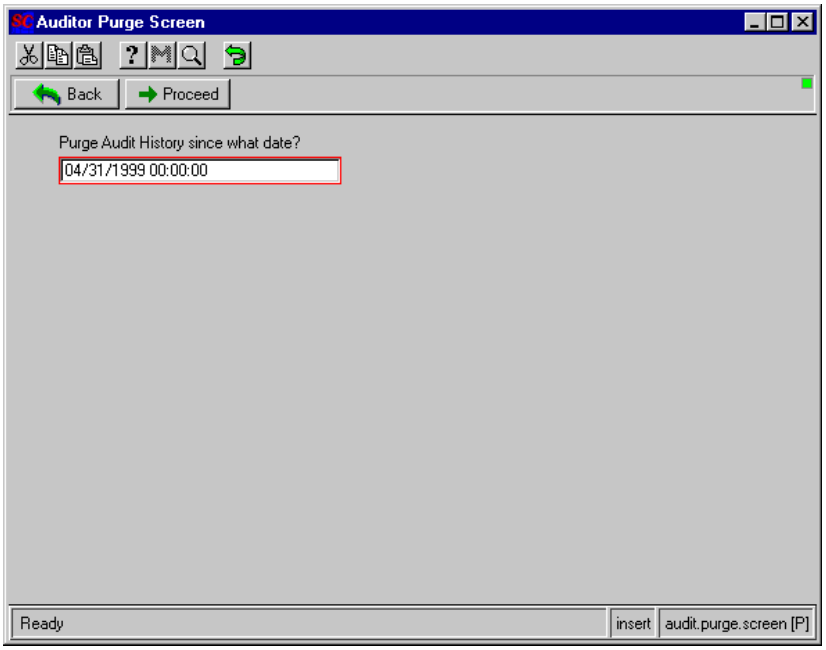
The Development Auditor Menu is displayed, with a message in the status bar indicating how many records were unloaded.

Purge Audit Records

To remove audit records that you no longer need:

1 Click on the **Purge Audit Records** button in the Development Auditor menu.

The Audit Purge form is displayed with the current date and time filled in.



Purge Audit History since what date?
04/31/1999 00:00:00

Ready | insert | audit.purge.screen [P]

Figure 8-3: Audit Purge Form

2 Replace the date displayed with the beginning date for the range of records you wish to delete.

Warning: Make sure that you carefully review the audit records in the Audit History record/QBE list before filling in this date. Purged files *cannot* be restored.

3 Click **Proceed**.

A confirmation prompt appears, telling you how many records have been purged.

4 Click **OK** to complete the process and return to the Development Auditor Menu.

9 Revision Control

CHAPTER

Revision Control provides developers and administrators a means of reverting to a previous version of a file or format. If during the process of creating or modifying forms you should find an error, Revision Control can be used to return to a working version of your file or form.

Revision Tracking allows a developer to:

- create a snapshot of a record
- add SCR information and comments to the snapshot
- replace the current version of the record at any time.

Note: Every revision made takes up as much disk space as the original record (plus a few bytes for comments).

Revision Control does not replace the Development Audit Utility, but is used in conjunction with that module to track, record and save changes to your system. The Development Audit utility provides a record of the changes (*deltas*) to ensure loading of the correct version when you move to production. See *Development Audit Utility* on page 141 for more information.

In ServiceCenter, revisions are handled as part of the Document Engine. As such they are available in all utilities that use the Document Engine as base code. This includes Database Manager, Format Control Editor, Link Editor, and others. In addition, special code has been added to make Revisions work for Forms Designer and the RAD Editor.

Revisions are stored in a separate file. The name of this file is specified in the Object record for the file, if one exists, or in the dbdict record otherwise. This file is generated by the system from an option on the data dictionary or the Object screen. Administrators can specify the maximum number of revisions to store for each record in a file. If no number is specified then an unlimited number is stored.

Purge scripts are included to help administrators with revision maintenance. The `sc.revision.purge.hanging` script purges all revisions that no longer have a parent record because the parent record was deleted or renamed. The `sc.revision.purge` script purges all revisions from the system. These scripts are accessed from Utilities - Maintenance area. See *Purging Revisions* on page 156.

Administrators should determine ahead of time the files for which they want to track revisions and then do minor setup to establish them. Administrators should also remember to purge revisions prior to migrating to a production system.

Creating Revisions

You can either create revisions for an entire set of records or for a single format or record. To create baseline revisions for an entire set of records, you must be at the start panel of the module you are working with. For instance, you create a baseline revision of all the forms in the Forms Designer from the Forms Designer main menu.

Note: When you create a revision, you create a copy of the record only, not the associated dbdict. As a result, any time a field is added to a dbdict, it will have to be added to the revision file dbdict also.

Create a baseline revision

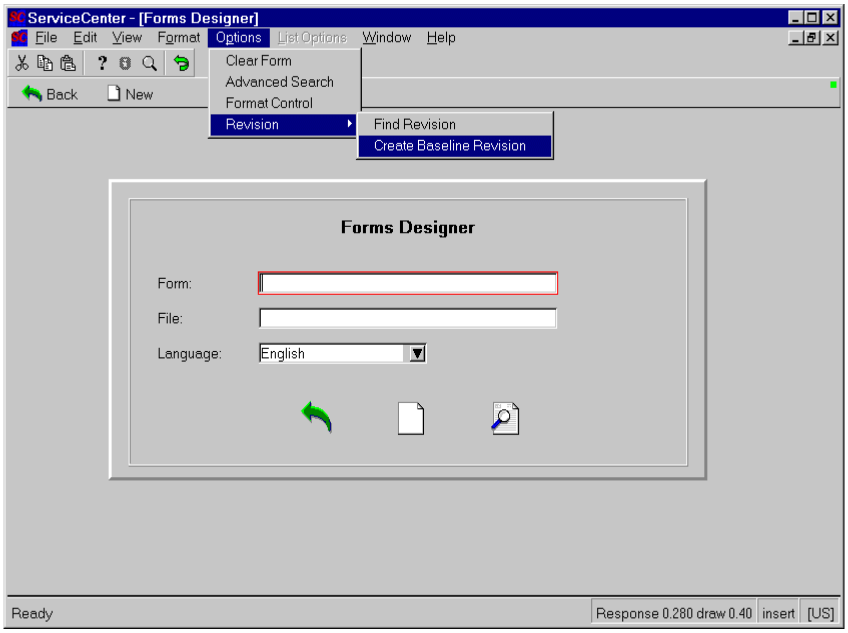


Figure 9-1: Creating a baseline revision

To create a baseline revision for all records in a module:

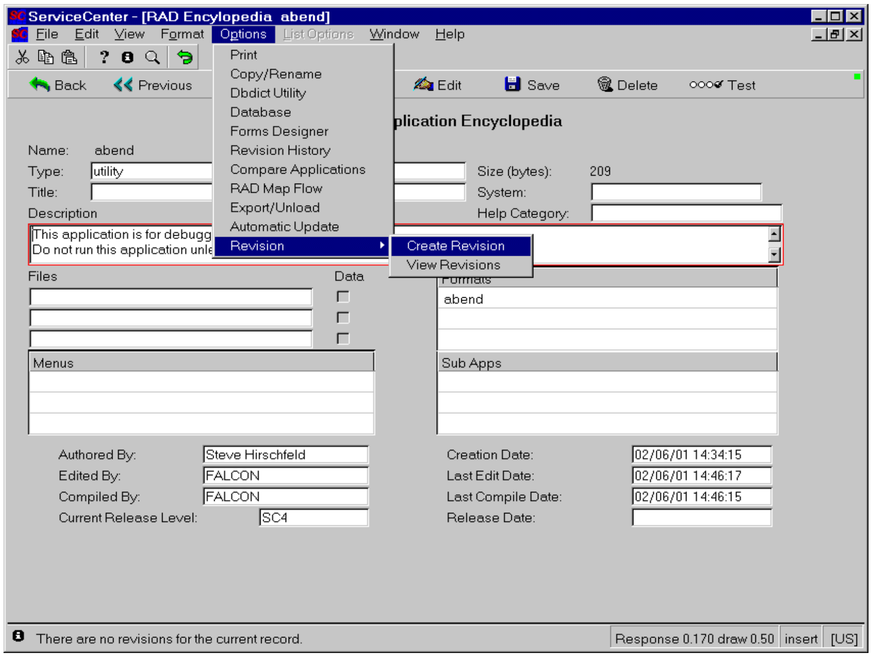
1 Open a search screen for a format or file that supports revisions. For example, the **Forms Designer** module.

2 Click **Options** from the options menu.

3 Select **Revision**.

4 Select **Create Revision**.
A dialog box appears, allowing you to enter SCR information to associate with the revision. This step is optional. Enter the SCR information or just click the **proceed** button (green checkmark) to begin.
The system will create a copy of all the records in the module, in this case every format in the Forms Design module. This operation could take some time.

Create a single revision



Name: abend
Type: utility
Title:
Size (bytes): 209
System:
Help Category:
Description
This application is for debugg
Do not run this application unle

Formats: abend

Authored By: Steve Hirschfeld
Edited By: FALCON
Compiled By: FALCON
Current Release Level: SC4
Creation Date: 02/06/01 14:34:15
Last Edit Date: 02/06/01 14:46:17
Last Compile Date: 02/06/01 14:46:15
Release Date:

There are no revisions for the current record. | Response 0.170 draw 0.50 | insert | [US]

Figure 9-2: Create a single revision

To create a revision of a single record or format:

1 Open a format or file that supports revisions. For example, a RAD application in the RAD editor. The **abend** application is used in Figure 9-2 on page 152.

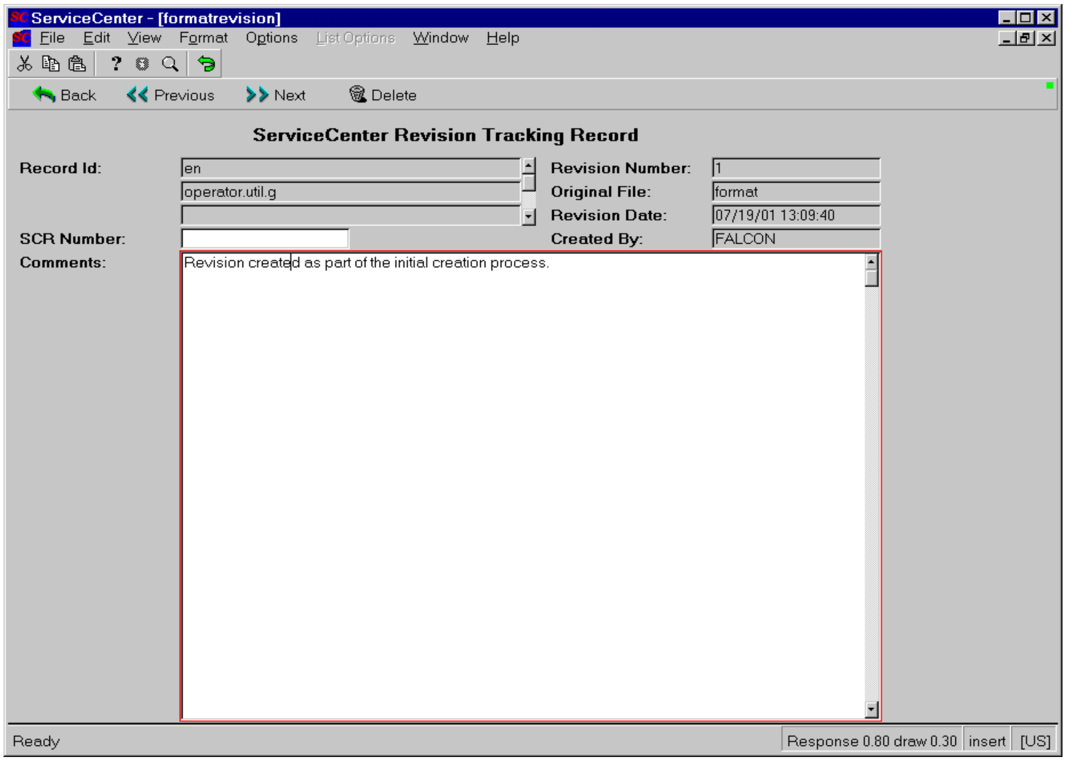
2 Click **Options** from the options menu.

3 Select **Create Revision.**
The revision tracking panel appears.

4 Fill in information relating to the revision. Making detailed comments will help others understand the reason for the revision, and will assist anyone needing to revert to this revision.

5 Click **Save.**
A copy of the revision is saved, and you are returned to the panel you started from, in this case the **abend** application in the RAD editor.

The Revisions panel


ServiceCenter - [formatrevision]
File Edit View Format Options List Options Window Help
Back Previous Next Delete

ServiceCenter Revision Tracking Record

Record Id:	en	Revision Number:	1
	operator.util.g	Original File:	format
		Revision Date:	07/19/01 13:09:40
SCR Number:		Created By:	FALCON
Comments:	Revision created as part of the initial creation process.		

Ready | Response 0.80 draw 0.30 | insert | [US]

Figure 9-3: The Revisions Panel

Fields on the Revisions Panel

SCR Number

Enter the SCR (Software Change Request) number to associate with this change, if applicable. This field is optional.

Comments

Enter any comments regarding the change here. Good developers make detailed comments.

Options Menu

Revert to this Revision

Select this option to make the revision currently displayed the record the system will use. You are prompted to save the current record as a revision before the revision is committed. This is a recommend step, though not required.

Print

Select this option to launch the print dialog.

Reverting to a Previous Revision

You can revert to a previous record, if one is available.

To revert to a previously saved copy of a record:

1 Click **Options**

2 Select **Revisions**>**Find Revisions**

- If there is no revision for the file, the following message will appear on the lower left of the GUI:

There are no revisions for the current record.

- If there is one revision available, it will display.
- If there is more than one revision available, a QBE list will appear. Select a revision from the list.

3 Select **Options**>**Revert to this Revision**

4 When you have selected the revision you want to restore, go to the **Options** menu and select **Revert to this Revision**. You are prompted as to whether you want to save the current version of the record as a revision. Peregrine recommends you do.

Searching for revisions

To search for revisions:

1 Open a module that supports revisions. For example, the RAD editor or the Forms Designer.

2 Click **Options** from the options menu.

3 Select **Find Revision.**

You can search for revisions using one or more of up to five search criteria:

syslanguage - Search for revisions by language indicator. For example, *en* for English.

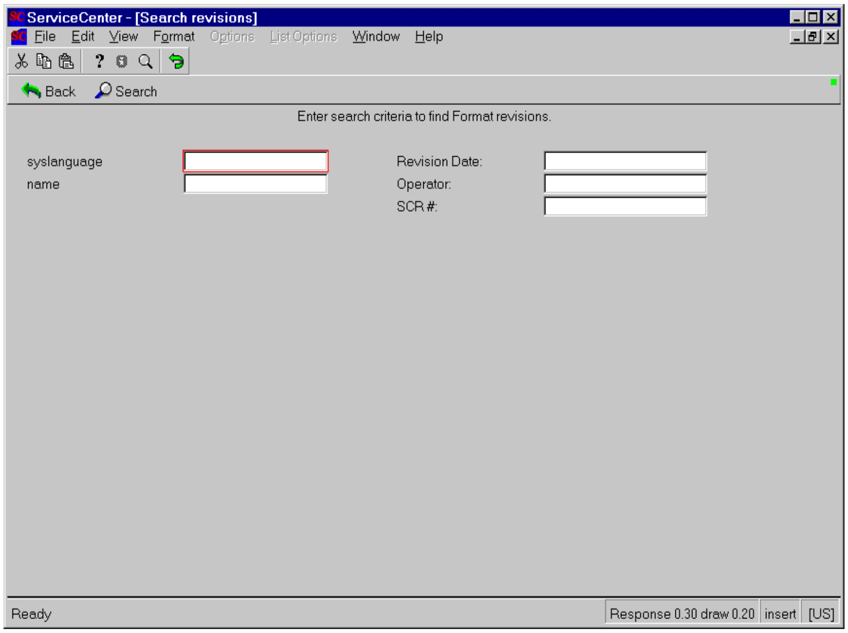
name - Search for revisions by name. Use the name of the form you are searching for. For example cc.incquick.

Revision Date - Search for revisions by date in the format month/day/year.

Operator - Search for revision by Operator name. Enter the name of the Operator who created the revision. For example, FALCON.

SCR# - Search for revisions by SCR number. Enter the SCR number of the revision you are seeking. For example, 42.

Revision search panels vary slightly by module. For instance, the RAD search panel does not include the *syslanguage* field shown in the example but otherwise performs in exactly the same manner.



Enter search criteria to find Format revisions.

syslanguage		Revision Date:	
name		Operator:	
		SCR #:	

Ready | Response 0.30 draw 0.20 | insert | [US]

Figure 9-4: Revision Search screen (Forms Design)

Purging Revisions

You can purge revisions from your system, which is useful when moving to a production system. There are two options available when purging revisions: **Purge all** and **Purge hanging**.

- **Purge all** will remove all revisions from your system.
- **Purge hanging** will remove only hanging revisions from your system. A hanging revision is a revision with no associated file or form. For example, if you create a format, create a revision of the format, then delete the original format, the revision would be left on the system, but with no associated file or format.

Note: You can restore a deleted format from its revision file.

To Purge revisions from your system:

1. Login as a System Administrator
2. Click the **Utilities** tab
3. Click the **Maintenance** button
4. To remove Hanging revision from your system, click the **Purge Hanging Revisions button.**
5. To remove ALL revisions from your system, click **the Purge All Revision Records** button.

10 CHAPTER DDE Support

ServiceCenter Dynamic Data Exchange (DDE) support is available in both Windows 16-bit (Windows 3.11) and 32-bit environments (Windows 95, NT, 98, etc.). DDE client support is created in a ServiceCenter RAD (Rapid Application Development) application.

DDE support in ServiceCenter is two way: you can create a DDE script to call *against* ServiceCenter from a Windows application or you can create a script using the DDE Script panel to make a DDE call *within* ServiceCenter. The difference being which application originates the call. For example, a ServiceCenter client in a Windows environment can push information to Microsoft® Excel, or Excel can pull from ServiceCenter. Basic DDE functions like exporting a file to Word or Excel are discussed in the ServiceCenter *User's Guide*. An example Visual Basic for Applications (VBA) DDE script for a call against ServiceCenter is included in this chapter on page 163. The DDE Script panel for creating DDE calls within ServiceCenter is covered on page 167.

Note: Some ServiceCenter records also can be exported to a text file. This option is not part of the DDE support and can be run from clients other than those operating in Windows.

ServiceCenter DDE server support provides an interface to applications outside ServiceCenter, allowing the use of DDE functions like **poke** and **execute**.

DDE Server

Integration of the ServiceCenter Work and Incident Management applications is achieved using the DDE server functionality of the ServiceCenter 32-bit Windows client.

Applications that implement the Microsoft Windows DDE server permit external applications to get and set data (called DDE Request and DDE Poke respectively), and execute commands. Typical use of get and set is to inspect and change data that is part of a document. For example, Excel allows the contents of spreadsheet cells to be read and written. Commands such as File / Save can be issued using DDE execute facility. There is little standardization between DDE server applications either in capabilities offered or the formatting of the commands sent over the DDE link.

DDE clients contact DDE servers using an application and topic name. The application name must be unique on the machine and the topic is typically the name of a document or the "Actions" topic typically used for command execution dealing with the entire application, rather than a specific document.

Implementation—System Events

ServiceCenter system events were created to permit ServiceCenter to react to events on client platforms external to the ServiceCenter system. System events are an arbitrary set of events that can be sent to and from either RAD or the RTE. They are generally used to start new RAD applications.

ServiceCenter DDE server implementation provides the DDE execute facility. ServiceCenter's DDE service name is "ServiceCenter" and the topic name is "Actions". The DDE Execute facility can be used to initiate ServiceCenter System Events. ServiceCenter System Events were created to permit ServiceCenter to react to events on the client platform external to the ServiceCenter system.

For example, the TAPI implementation creates a System Event to start a RAD application when the phone rings to handle the call. RAD programs written to extract the System Event parameters and act upon them handle ServiceCenter System Events. With the advent of the DDE server functionality, an external application, such as Excel, Access, or Delphi, can hook up to ServiceCenter using application *ServiceCenter* and topic *Actions* to trigger ServiceCenter system events.

Events in the Standard System

The standard ServiceCenter system provides a number of predefined system events. These events are of two types:

- Hardcoded
- Editable

Hardcoded events

There are currently 12 system events whose parameters are hardcoded into the system and are not user definable. These events can be used to exchange data with applications external to ServiceCenter through the use of Dynamic Data Exchange (DDE) conversations.

The following system events are hardcoded to pass predetermined field values to external applications:

Event Name	Application Called	Parameter
EditCM3Request	dde.editcm3request	Number
EditCM3Task	dde.editcm3task	Number
EditIncident	dde.editincident	Number
EditOCMLineItem	dde.edit.ocm.lineitem	Number
EditOCMQ	dde.editocmq	Number
EditOCMRequest	dde.edit.ocm.request	Number
EditProblem	dde.editproblem	Number
ListCM3R	dde.listcm3r	Query
ListProblems	dde.listproblems	Query

Event Name	Application Called	Parameter
ShowChangePages	dde.show.change.pages	Number
ShowPages	dde.showpages	Number
ShowTaskPages	dde.show.task.pages	Number

Editable events

Editable system events are accessed through the `pmtapi` file and can be configured to pass any user-defined field value to an external application.

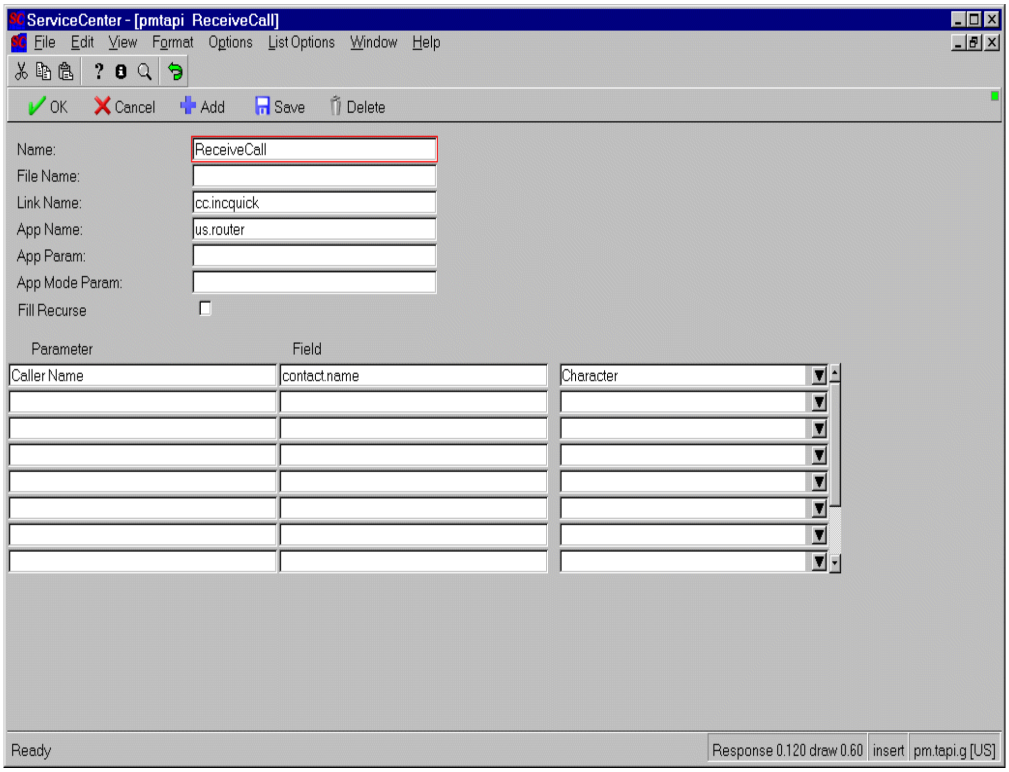
Event Name	*pmtapi* Record	Routing Application	Application Called
ReceiveCall	incident	us.router	cc.first
ReceiveCallList	incident list	us.route.list	cc.list.incident
ReceiveProblem	problem	us.router	apm.first
ReceiveProblemList	problem list	us.route.list	apm.list.problems
ReceiveRequest	ocmq	us.route	ocmq.access
ReceiveRequestList	ocmq.list	us.route.list	ocmq.access

To edit an event using *pmtapi*:

1. Select the Toolkit tab in the system administrator's home menu.
2. Click Database Manager.
3. Enter `pmtapi` in the File field.
4. Click Search or press Enter.
5. Enter the name of a `pmtapi` record from the table on the previous page in the **Name** field.

   For example, enter `ReceiveCall`.
6. Click Search or press Enter.

The requested record is displayed.


ServiceCenter - [pmtapi ReceiveCall]
File Edit View Format Options List Options Window Help
OK Cancel Add Save Delete

Name:	ReceiveCall
File Name:	
Link Name:	cc.incquick
App Name:	us.router
App Param:	
App Mode Param:	
Fill Recurse	☐

Parameter	Field	
Caller Name	contact.name	Character

Ready | Response 0.120 draw 0.60 | insert | pm.tapi.g [US]

Figure 10-1: pmtapi record for an editable system event

7 Add or delete field names.

These names must match fields listed in the Database Dictionary record of the file for which the event is used.

8 Select the data type of the field from the drop-down list in the right column.

This data type must match the data type of the field as it appears in the Database Dictionary record.

9 Create the parameter for the field that will be passed to the external application by the event.

10 Click Save.

Field values

Field	Description
Name	Name of the **pmtapi** record
File Name	
Link Name	Name of the link record used
App Name	Name of the RAD application. For example cc.incquick
App Param	
App Mode Param	
Fill Recurse	Determines whether or not to use recursive fill based on the link record and parameters.
Parameter	Parameter name passed to an external application
Field	Name of the field defined by the parameter
[Data Type]	Data type of the field

Note: Fields that do not appear in this table are not currently in use.

Example

This example uses Visual Basic for Applications.

Note: The ServiceCenter server must be running in order for the following example to work.

The format of the ServiceCenter execute command string for system events is:

```
SystemEvent( event name, parameter name, parameter value, …)
```

- The **event name** corresponds to the event name in the ServiceCenter SystemEvents table. See *Events in the Standard System* on page 159 for a list of system events.
- The **parameter name / value pairs** are known to the RAD program. See *Editable events* on page 160 for instructions on how to access, edit, and define parameters for events.

```
Sub ReceiveCall()
channel = DDEInitiate("ServiceCenter", "Actions")
DDEExecute channel, "[SystemEvent(""ReceiveCall"", ""Caller Name"", ""KENTNER""
)]"
DDETerminate channel
End Sub
```

The example above:

- Initiates a conversation with the ServiceCenter system
- Executes a DDE command starting the System Event **Receive Call,** passing the parameter name **Caller Name** with the parameter value of **Kentner**.
- Terminates the conversation.

Usage notes

To use the DDE Server functionality, the ServiceCenter Windows client must be started and a user should be logged in. DDE does not automatically start the server (as OLE does). The user must be logged in so that the environment is set up for the ServiceCenter user.

Additional DDE Server Capabilities

The DDE server can also handle DDE client transactions that perform:

- Requests: Get the value of a named item and return it as a string
- Pokes: Set the value of a named item
- Executes: Ask the ServiceCenter GUI to execute a transaction or set the focus to a named item

This functionality can be used to *script* user interaction for common operations such as closing a ticket, which may require several fields to be filled in and several transactions to be made.

Requests and pokes

Requests and Pokes are the DDE mechanisms for obtaining a copy of or setting the value of named items. ServiceCenter uses the widget's input property to name the item. For example, to set the user name on the ServiceCenter login screen, the following DDE command could be used (this example is in Visual Basic for Applications):

```
DDEPoke nChannel, "$user.id", "falcon"
```

Requests and pokes take and return string type data.

Executes

Executes are the DDE mechanisms for requesting that an application process data or perform an action. ServiceCenter provides two execute capabilities:

- Transact
- SetFocus.

Transact

The Transact execute function directs ServiceCenter to execute a transaction as though a user has pressed a function key, or pressed a button. The Transact execute function requires one operand that designates the number of the function key or button ID of the button that was pressed. This example in Visual Basic for Applications shows how to tell ServiceCenter that the "fill" key was depressed:

```
DDEExecute nChannel, "[Transact( ""9"" )]"' issue a fill command
```

SetFocus

The SetFocus execute function directs ServiceCenter to place the focus in a named widget (using the field's input property as the name). This example in Visual Basic for Applications shows the focus being set to the file name input field in the data base manager format:

```
DDEExecute nChannel, "[SetFocus( ""file.name"" )]" '
```

Example

The following example, written in VBA, illustrates a DDE script that takes a user directly to the Database Manager from the login screen. Before executing this script, you must be on a ServiceCenter login screen (**login.prompt.g format**)

```
Sub SC()
Dim nChannel As Long
Dim strReturned As String
nChannel = DDEInitiate("ServiceCenter", "ActiveForm")
DDEPoke nChannel, "$user.id", "falcon"
DDEExecute nChannel, "[Transact( ""0"" )]"' login
DDEExecute nChannel, "[Transact( ""1"" )]"' go to the command interface
DDEPoke nChannel, "$command", "db"' go to the database manager
DDEExecute nChannel, "[Transact( ""0"" )]"
DDEExecute nChannel, "[SetFocus( ""file.name"" )]" ' set the focus in the file name box
DDETerminate nChannel
End Sub
```

The DDE Script panel

The DDE script panel, dde.script.g, assists in creating ServiceCenter DDE scripts for DDE calls inside ServiceCenter against an outside application, such as Excel.

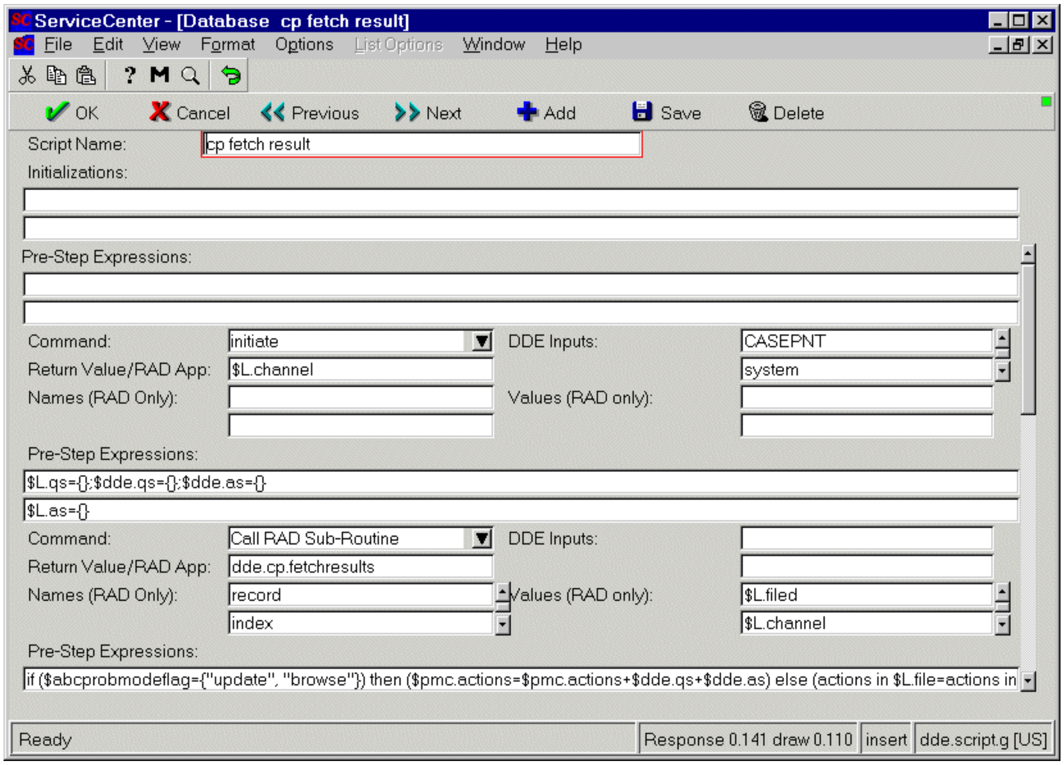
Accessing the Script Panel

To access the DDE script panel from the system administrator’s home menu:

1 Click the Utilities tab.

2 Click the Tools button.

3 Click the DDE Script button. The DDE script panel appears.


ServiceCenter - [Database cp fetch result]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete

Script Name: cp fetch result

Initializations:

Pre-Step Expressions:

Command: initiate
Return Value/RAD App: $L.channel
Names (RAD Only):
DDE Inputs: CASEPNT, system
Values (RAD only):

Pre-Step Expressions:
$L.qs={};$dde.qs={};$dde.as={}
$L.as={}

Command: Call RAD Sub-Routine
Return Value/RAD App: dde.cp.fetchresults
Names (RAD Only): record, index
DDE Inputs:
Values (RAD only): $L.filed, $L.channel

Pre-Step Expressions:
if ($abcprobmodeflag={"update", "browse"}) then ($pmc.actions=$pmc.actions+$dde.qs+$dde.as) else (actions in $L.file=actions in

Ready | Response 0.141 draw 0.110 | insert | dde.script.g [US]

Figure 10-2: The DDE script panel

Fields on the DDE script panel

Script Name - Enter a name for the new script.

Initializations - Inititializations call or start certain ServiceCenter services using RAD programming calls. See the RAD programming Guide for usage and syntax. This field is optional.

Pre-Step Expressions - Pre-Step expressions start certain ServiceCenter services using RAD programming calls. This field is optional.

Command - Select a command from the drop-down menu list. Your choices are:

- Initiate - start a session
- Terminate - end the session
- Poke - set the value of the named item.
- Request - get the value of a named item and return it as a string.
- Execute - ask ServiceCenter to execute a transaction or set the focus to a named item.
- Call RAD sub-routine - call an already defined RAD subroutine.

DDE Inputs - If you selected one of the first five options from the **Command** drop-down menu list, then you must enter a value in this field.

Return Value - Enter a value, such as $L.channel. The value you enter here will be based on what you are calling. If you selected a DDE command from the **Command** drop-down menu list, then enter parameters for the DDE application. If you selected “Call RAD sub-routine” from the **Command** drop-down menu list, then enter the name of the RAD application you are calling in this field.

Names (RAD Only) - Enter the name of the RAD application you are calling, if you selected “Call RAD sub-routine” in the **Command** drop-down menu list.

Values (RAD Only) - Enter value to be passed to the RAD application you are calling, if you selected “Call RAD sub-routine” in the **Command** drop-down menu list.

DDE Client

This section provides an example of a DDE client conversation executed as a RAD application. Refer to the RAD guide for specifics on programming a RAD application.

There are six different *actions* associated with a DDE client conversation that can be initiated from a **DDE** RAD panel. The actions are the standard DDE actions:

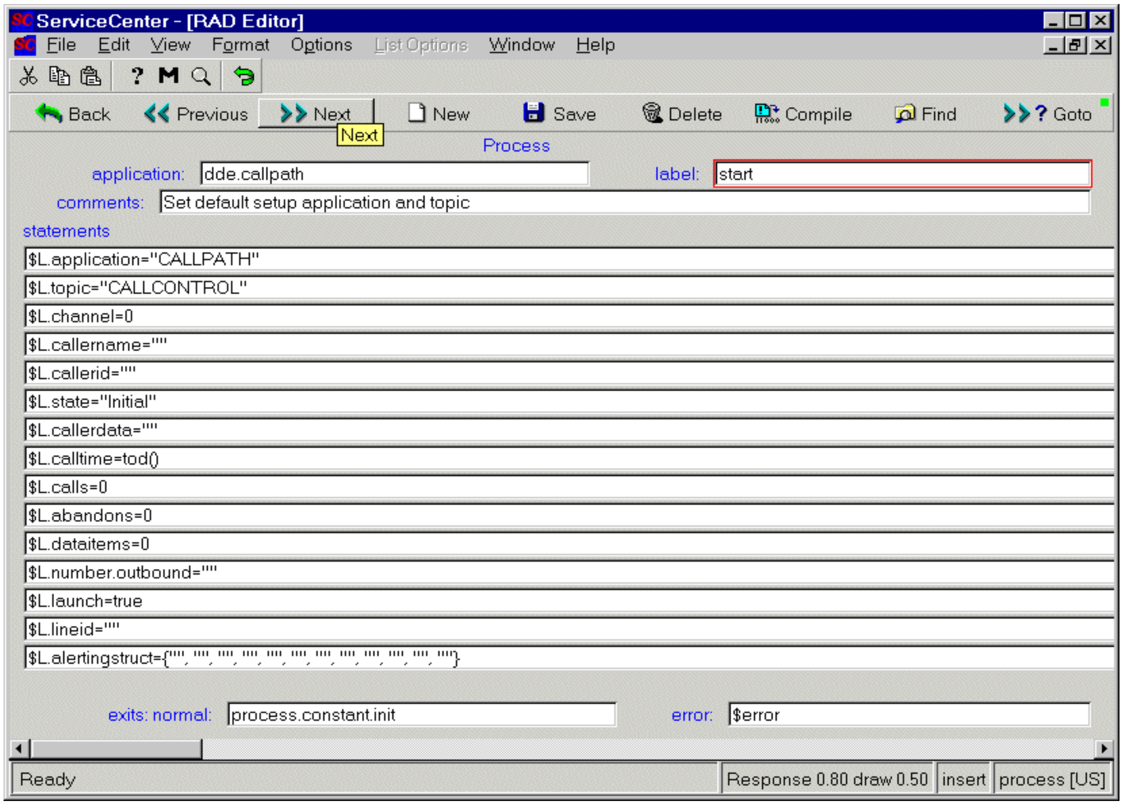
- Initiate
- Advise
- Request
- Poke
- Execute
- Terminate

The section below outlines the initiate action, and uses an IBM product named CallPath as an example. Any of the above listed DDE actions can be performed with the DDE RAD panel.

Note: Constructing a DDE RAD application requires you follow RAD conventions. For a complete description of programming in RAD, along with requirements for any RAD application to function, please refer to the RAD guide.

The Process panel

A process panel is a type of RAD panel that is used to initialize or set variables used later in your RAD application. Process panels can also process expressions. The Process panel shown is labeled **start** as it is the first panel in the example DDE RAD application.


ServiceCenter - [RAD Editor]
File Edit View Format Options List Options Window Help
Back Previous Next New Save Delete Compile Find Goto
Next

Process

application: dde.callpath label: start

comments: Set default setup application and topic

statements

```
$L.application="CALLPATH"
$L.topic="CALLCONTROL"
$L.channel=0
$L.callername=""
$L.callerid=""
$L.state="Initial"
$L.callerdata=""
$L.calltime=tod()
$L.calls=0
$L.abandons=0
$L.dataitems=0
$L.number.outbound=""
$L.launch=true
$L.lineid=""
$L.alertingstruct={"", "", "", "", "", "", "", "", "", "", "", ""}
```

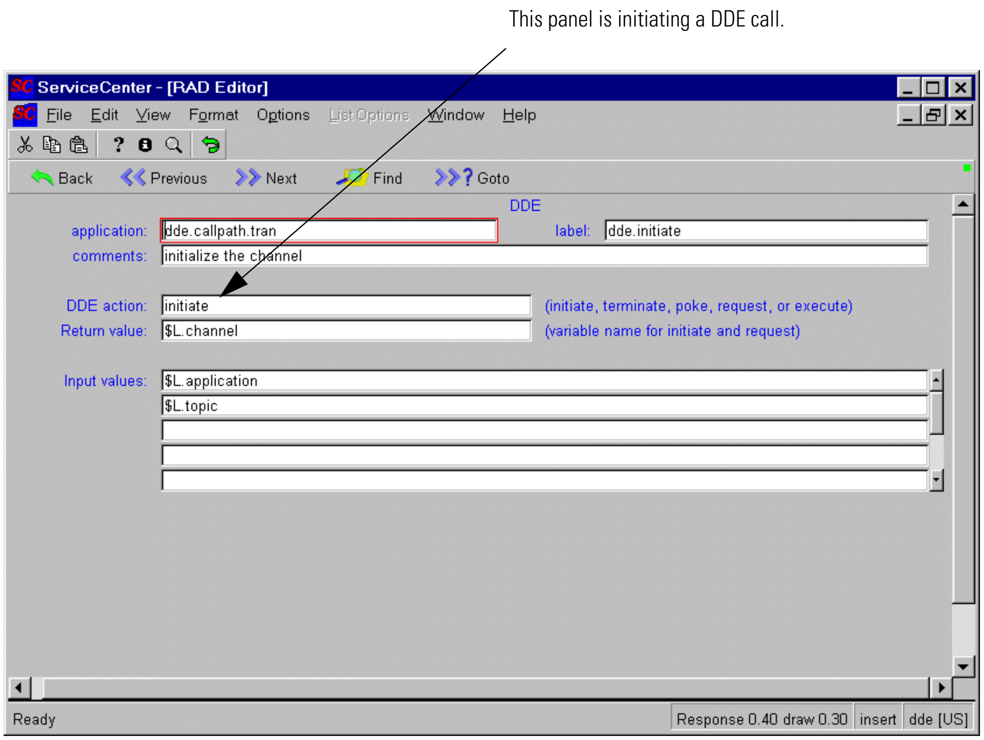
exits: normal: process.constant.init error: $error

Ready | Response 0.80 draw 0.50 | insert | process [US]

Figure 10-3: RAD start panel

The DDE RAD panel

The **dde** RAD panel is used to perform one of the five DDE commands. The RAD **start** panel is used to set variables referenced on this panel and later in the RAD application.



application: dde.callpath.tran
label: dde.initiate
comments: initialize the channel
DDE action: initiate (initiate, terminate, poke, request, or execute)
Return value: $L.channel (variable name for initiate and request)
Input values: $L.application
$L.topic

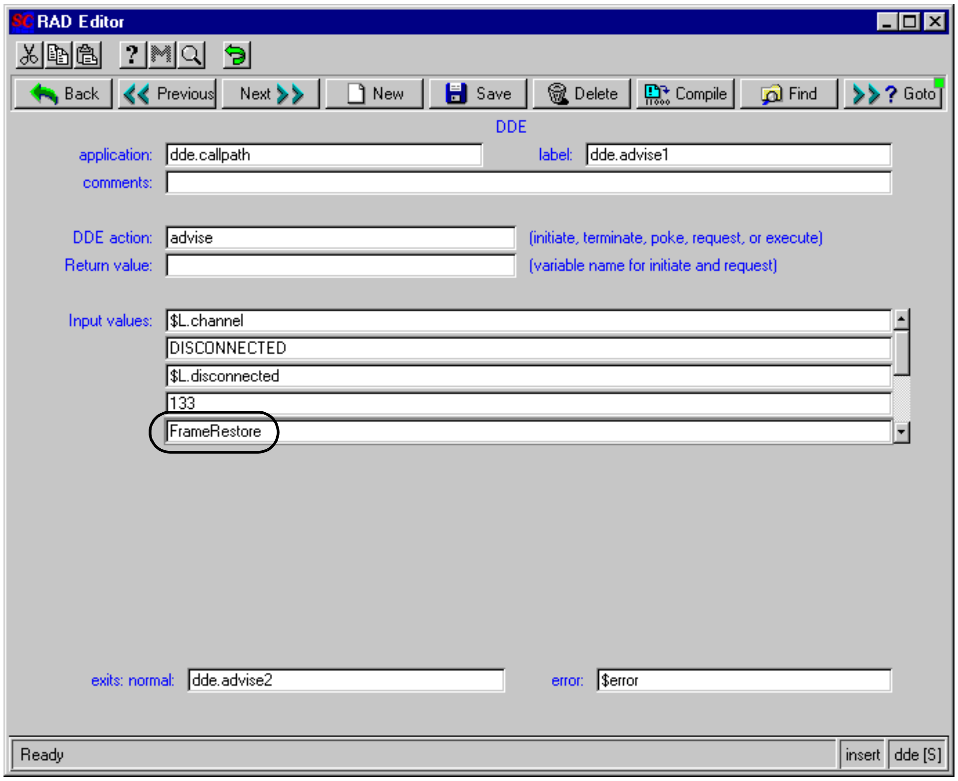
Figure 10-4: The DDE Panel.

Field	Value	Description
DDE action	Initiate, Poke, Request, Advise, or Terminate.	The DDE action to perform.
Return value	$L.channel	The channel used throughout to identify the conversation. This value is set on the RAD start panel. A ServiceCenter client can carry on multiple conversations.
Input values	$L.application $L.topic	These values are set on the start panel of the RAD application. In this case they are set to describe CALLPATH and CALLCONTROL respectively

Frame Restore option

The *FrameRestore* option directs ServiceCenter to take the focus when a DDE advise hot link is updated. For example, if the `FrameRestore` parameter is added to an advise **dde** RAD panel, a software telephone being used by an agent to receive a call takes the focus. This allows the agent to answer the call without searching for the softphone application.

To activate the FrameRestore option in RAD, enter `FrameRestore` as the fifth input value parameter in the appropriate DDE **advise** panel.



application: dde.callpath
label: dde.advise1
comments:
DDE action: advise (initiate, terminate, poke, request, or execute)
Return value: (variable name for initiate and request)
Input values:
$L.channel
DISCONNECTED
$L.disconnected
133
FrameRestore
exits: normal: dde.advise2
error: $error
Ready | insert | dde [S]

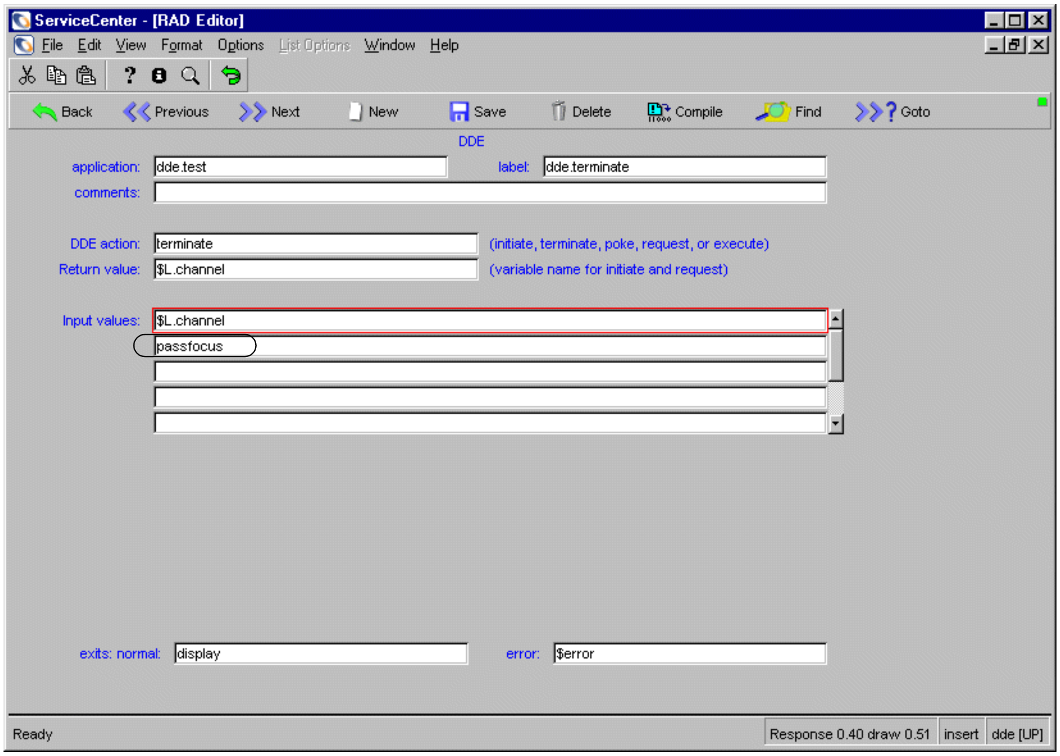
Figure 10-5: DDE RAD panel activating FrameRestore option

PassFocus Option

The *PassFocus* option directs ServiceCenter to pass the focus when a DDE terminate command is issued. For example, if the `PassFocus` parameter is added to a terminate **dde** RAD panel, the focus stays on the receiving application and focus does not return to ServiceCenter. Without the Passfocus command, ServiceCenter will gain the focus.

To activate the PassFocus option in RAD, enter `PassFocus` as the second argument of a DDE terminate panel.

For example when the ServiceCenter client is used to export information to Excel, if `passfocus` is the second argument of the DDE terminate panel, the focus will go to Excel instead of ServiceCenter.

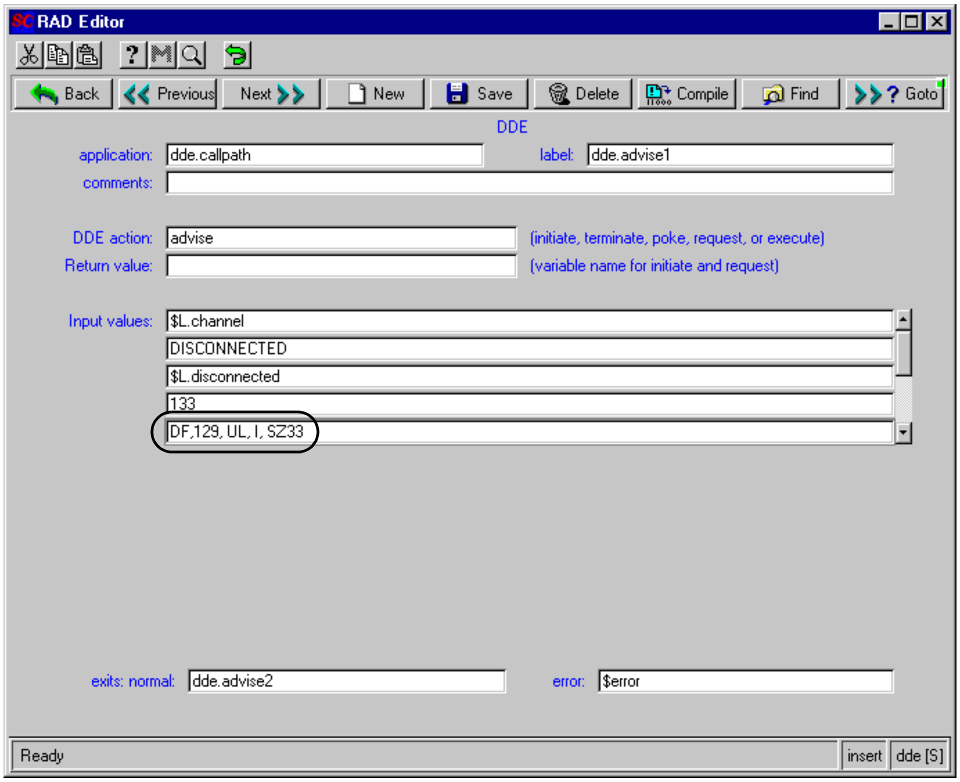

ServiceCenter - [RAD Editor]
File Edit View Format Options List Options Window Help
Back Previous Next New Save Delete Compile Find Goto
DDE

application: dde.test
label: dde.terminate
comments:
DDE action: terminate (initiate, terminate, poke, request, or execute)
Return value: $L.channel (variable name for initiate and request)
Input values: $L.channel
passfocus
exits: normal: display
error: $error
Ready
Response 0.40 draw 0.51 | insert | dde [UP]

Figure 10-6: DDE terminate panel with Passfocus option

Structure support option

The *structure support* option allows a DDE advise action to use a user-defined data format. An example might be **DF,129, UL, I, SZ33**. All values are comma delimited.



DDE

application: dde.callpath
label: dde.advise1
comments:

DDE action: advise (initiate, terminate, poke, request, or execute)
Return value: (variable name for initiate and request)

Input values:
$L.channel
DISCONNECTED
$L.disconnected
133
DF,129, UL, I, SZ33

exits: normal: dde.advise2
error: $error

Ready | insert | dde [S]

Figure 10-7: DDE RAD panel expressing a user defined data format

Sample values

Value	Description
DF	Identifies the string as a data format
129	Integer value for the clipboard data format value required for the advise action.
UL	Unsigned long integers
I	Integers
SZ33	Null terminated string 33 characters long

SystemEvents File

System Events are defined in the SystemEvents file in ServiceCenter. This file contains records that have an Event Name and a RAD Application name. When an event is received, the corresponding RAD application is invoked in a new RAD thread. The SystemEvents file contains all the events that are used to invoke a RAD application. The events used to invoke a RTE function register themselves upon start-up of the client. If you edit a system event record, you must re-login before the system recognizes the changes. See *Events in the Standard System* on page 159 for a list of Hardcoded and editable events.

Note: You can have multiple records for one event; each application will be invoked in a separate RAD thread.

Accessing records

To access system events records:

1 Select the **Toolkit** tab in the system administrator's home menu.

2 Click **Database Manager**.

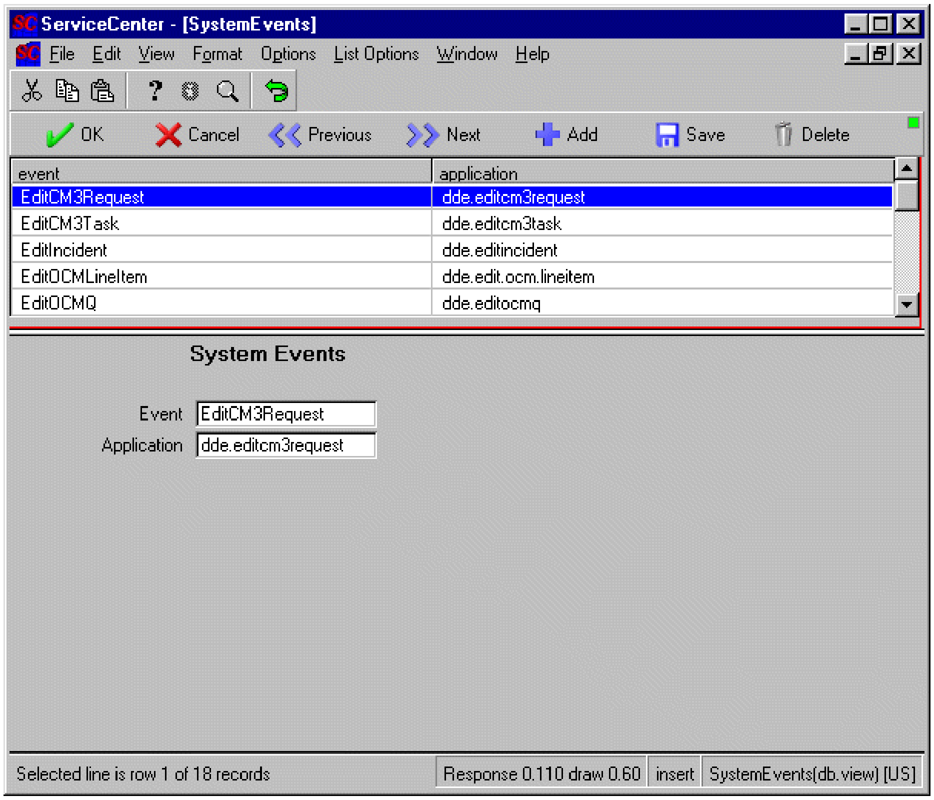
The Database Manager dialog box is displayed.

3 Enter SystemEvents in the **File** field.

4 Click Search or press Enter.

A blank system events record is displayed.

5 Enter the name of the record you want to view, or click Search to display a record list of all events in the system.


ServiceCenter - [SystemEvents]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete

event	application
EditCM3Request	dde.editcm3request
EditCM3Task	dde.editcm3task
EditIncident	dde.editincident
EditOCMLineItem	dde.edit.ocm.lineitem
EditOCMQ	dde.editocmq

System Events

Event: EditCM3Request

Application: dde.editcm3request

Selected line is row 1 of 18 records | Response 0.110 draw 0.60 | insert | SystemEvents(db.view) [US]

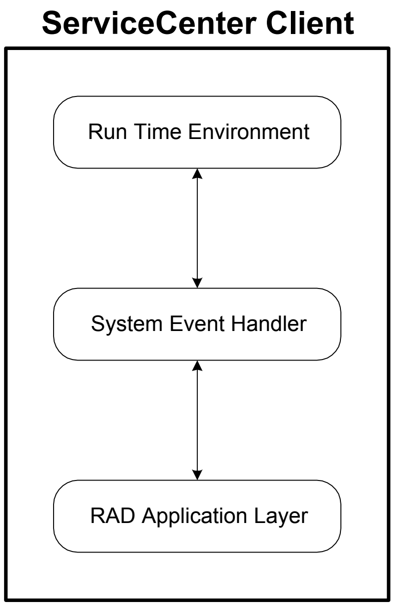
Figure 10-8: System events record

Architecture

In the case of ServiceCenter Telephony (standard system), when a phone call comes in:

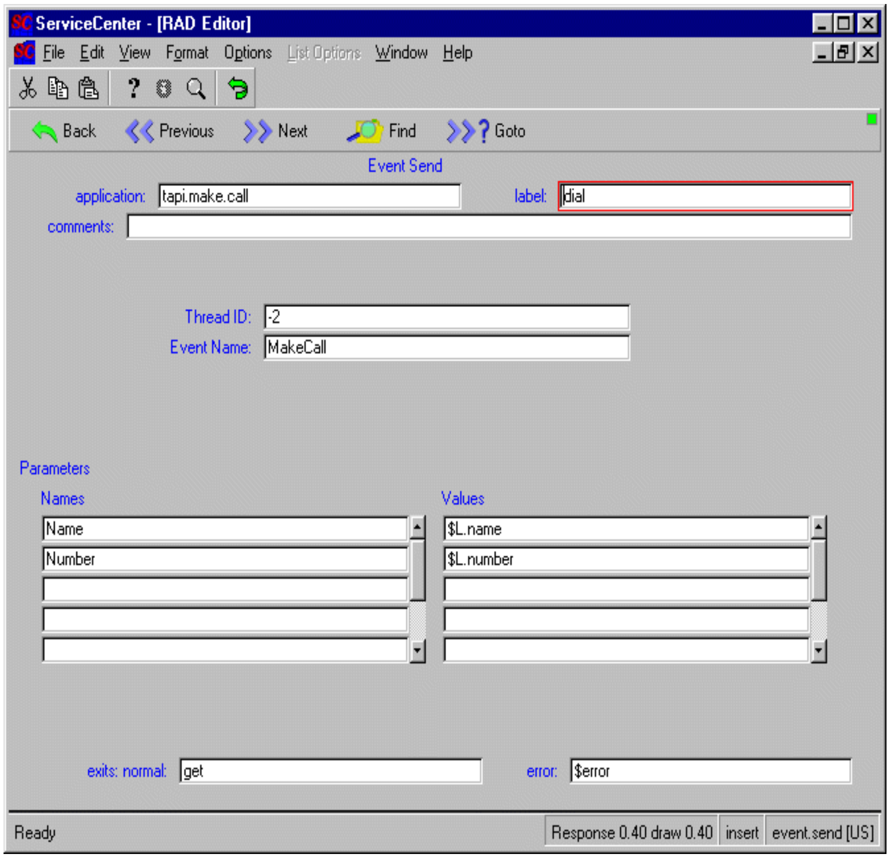
- the RTE generates a ReceiveCall event
- that event gets passed to the System Event Handler (SEH)
- the SEH starts the **us.router** RAD application in a new RAD thread

When you make a phone call the RAD application sends a MakeCall event to the SEH which invokes the RTE function to make a phone call.



While System Events are normally used to communicate between the application layer and the RTE, they can also be used to pass events between RAD applications and between RTE functions.

To send a System Event from RAD, use the **event.send** RAD Command panel. You must use -2 as the Thread ID. Fill in the **Event Name** with the name of the event and pass any Parameters in the **Names** and **Values** arrays. Since each event is arbitrary, so are the parameters it requires. The event name and parameters are listed on page 159. For instructions on accessing and editing RAD command panels, refer to the ServiceCenter *RAD Guide*.



Event Send

application: tapi.make.call
label: dial
comments:

Thread ID: -2
Event Name: MakeCall

Parameters

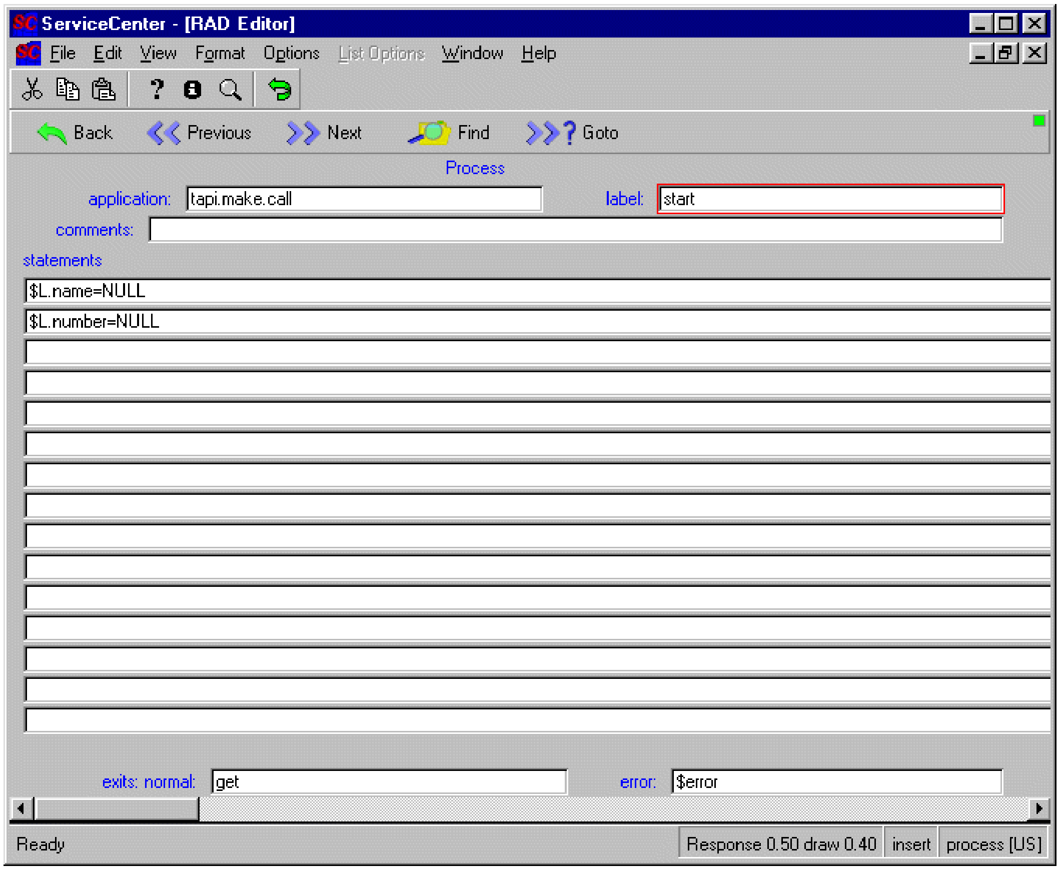
Names	Values
Name	$L.name
Number	$L.number

exits: normal: get
error: $error

Ready | Response 0.40 draw 0.40 | insert | event.send [US]

Figure 10-9: event.send RAD panel

When receiving a System Event in a RAD application you can use the event.name() and event.value() functions to get the event name and parameters for that event. See the appropriate table for the event parameters you need.


ServiceCenter - [RAD Editor]
File Edit View Format Options List Options Window Help
Back Previous Next Find Goto
Process

application: tapi.make.call
label: start
comments:
statements
$L.name=NULL
$L.number=NULL
exits: normal: get
error: $error
Ready
Response 0.50 draw 0.40 | insert | process [US]

Figure 10-10: process RAD panel

11 Data Policy

CHAPTER

Many common data tailoring tasks in ServiceCenter are performed by Format Control. Format Control is an often complex procedure applied at the form level and, if overused, can affect system performance. The *data policy* feature operates at the table level and can achieve many of the same results as Format Control without the complexity and without taxing system resources.

The data policy feature provides a simple interface where system administrators can apply default values, mandatory fields, or lookup validations, to a specific table. *These policies, once set, will be enforced across the entire system, regardless of what form is being used to display the data*.

Accessing Data Policy

To use this feature, select **Data Policy** from the Options menu of a table in the Database Dictionary.

Data Policy Expressions

Data Policy rules apply to the GUI presentation of data. For example, in the **Invisible** or **Read Only** fields, you can specify an expression. If this expression evaluates to *true* at the time the record is being displayed, any controls referencing the field in question are set to *read-only* or *visible* as appropriate.

This allows you to place some degree of data hiding or control into your system without the need to construct numerous different forms and views. For example, if you want only system administrators to be able to modify the **contact.name** field in the contacts file, use this expression:

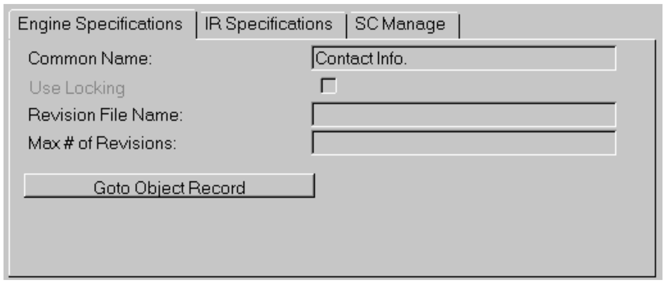
```
index("SysAdmin", $lo.ucapex)>0
```

The fields on a file's Data Policy record are defined on the record's dbdict. You cannot add new fields directly to a Data Policy record.

Changes require that you cycle the client by logging out of the system and logging in again.

Data Policy and the Object record

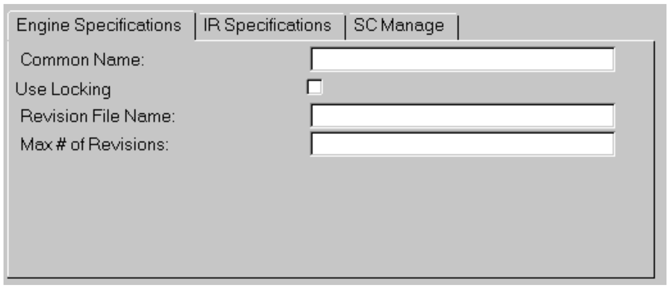
Data Policy control for a table can be associated and expanded with an Object record. If there is an associated Object, then the Object offers control over revisions, IR searches, and displaying records through SC Manage. If a record in Data Policy has an Object associated with it, then the fields on the Engine Specifications tab will be greyed out and unavailable. A button will appear that, when clicked, will take you to the Object record, allowing you to make changes there.



Common Name: Contact Info.
Use Locking
Revision File Name:
Max # of Revisions:

Figure 11-1: Engine Specifications tab with Object Associated

If a Data Policy record does not have an associated Object, then the Engine specifications and the SC Manage tabs will contain editable fields that control revisions and display.



Common Name:	
Use Locking	☐
Revision File Name:	
Max # of Revisions:	

Figure 11-2: Engine Specifications tab, no Object Associated

Data Policy and Revisions

The revisions option for Data Policy creates a revision dbdict, with the name specified on the Engine Specifications tab. See *Revision Control* on page 149 for details on working with revisions.

To create a revision dbdict:

1 Enter a revision file name in the **Revision File Name** field on the Engine Specifications tab. You can limit the number of revisions allowed by filling in the **Max # of Revisions** field.

2 Click Options.

3 Select **Create Revision File**.

4 The system creates a new dbdict with the name you chose in step 1.

Example: Creating and Managing Revisions

The following example steps through the process of creating and controlling a revision. The example uses the location file.

1 Open the **location** record in Data Policy.

2 Enter a revision name on the Engine Specifications tab, for instance **LocationRevision**.

3 Enter the maximum number of revisions allowed on the system for files controlled by this record. For example, enter **2**.

4 Save the **location** Data Policy record.

5 Go to the Database Manager and enter location in the Form field. A QBE list appears.
Every file in the QBE list shown is controlled by the Data Policy record, *Locations*.

6 Select a record, for example ACME HQ, based in Chicago.

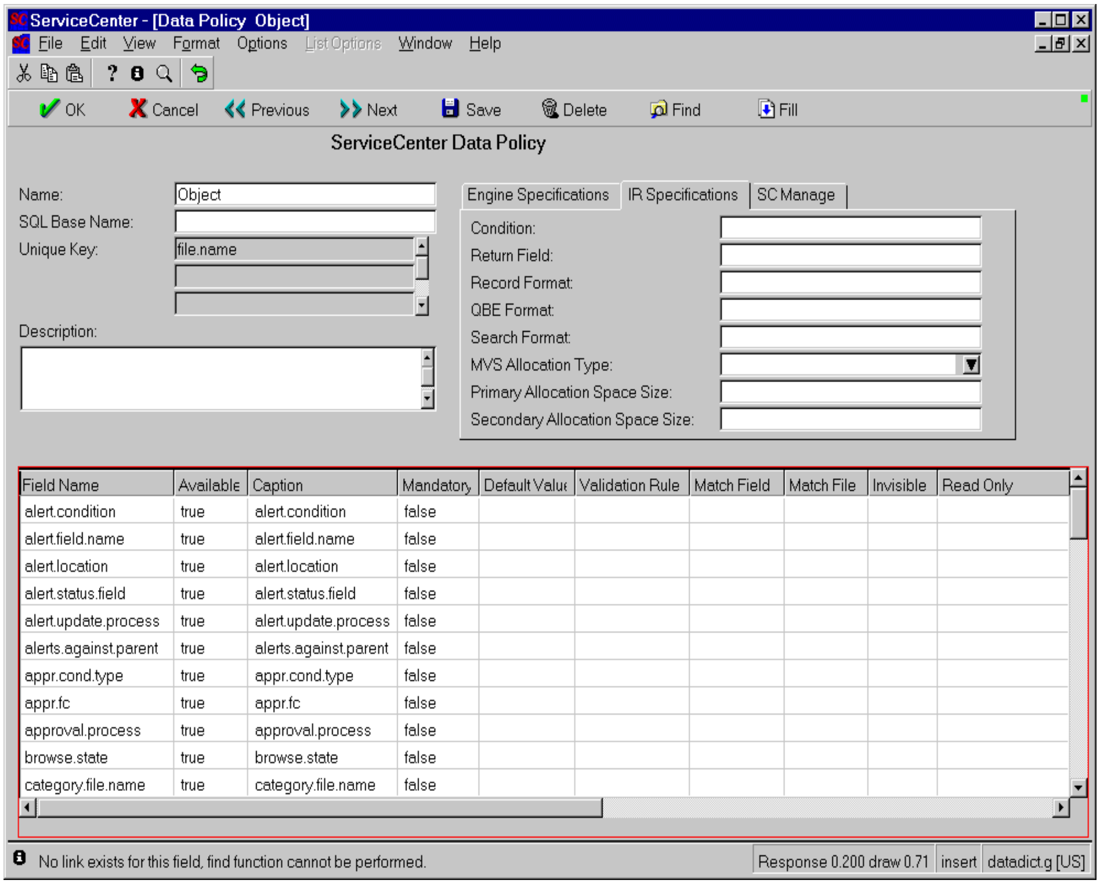
7 With the record open, select **Options** > **Revision** > **Create Revision**. Enter any relevant text to help you understand the revision and its purpose. Label this revision **one**.

8 Make a change to the ACME HQ record and save.

9 Select **Options** > **Revision** > **Create Revision**. Label this revision **two**.

The maximum number of revisions has now been reached, at least for the files controlled by the Locations Data Policy record. The next revision you make will eliminate the revision you labeled **one**.

Fields on the Data Policy Form


ServiceCenter - [Data Policy Object]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Save Delete Find Fill
ServiceCenter Data Policy
Engine Specifications IR Specifications SC Manage

Name: Object
SQL Base Name:
Unique Key: file.name
Description:

Condition:
Return Field:
Record Format:
QBE Format:
Search Format:
MVS Allocation Type:
Primary Allocation Space Size:
Secondary Allocation Space Size:

Field Name	Available	Caption	Mandatory	Default Valu	Validation Rule	Match Field	Match File	Invisible	Read Only
alert.condition	true	alert.condition	false						
alert.field.name	true	alert.field.name	false						
alert.location	true	alert.location	false						
alert.status.field	true	alert.status.field	false						
alert.update.process	true	alert.update.process	false						
alerts.against.parent	true	alerts.against.parent	false						
appr.cond.type	true	appr.cond.type	false						
appr.fc	true	appr.fc	false						
approval.process	true	approval.process	false						
browse.state	true	browse.state	false						
category.file.name	true	category.file.name	false						

No link exists for this field, find function cannot be performed. Response 0.200 draw 0.71 insert datadict.g [US]

Figure 11-3: Data Policy Form

Name - The name of this record.

SQL Base Name - The name used for SQL mapping. The maximum number of characters you can enter in this field varies depending on the database you are mapping to. To be safe, Peregrine recommends you enter no more than 12 characters in this field.

Unique Key - Enter a Unique key, which should match the associated dbdict's Unique Key.

Description - Enter a brief description of the data policy record and its objective.

Field Name - This column shows the fields on the associated dbdict. The fields in this column are read only.

Available - This column determines whether the field is available to a user on a form. True or an expression that evaluates to True indicates it is available, while False indicates the field is not. The default is True.

Caption - The Caption field can be used as an alternate heading for columns on a QBE list.

To make a column heading visible on a QBE list, you must be on the QBE list in question. From there, select **Options** >**Modify Columns** and add columns to the QBE list. If **View Record** list is enabled, then the Modify Columns option can be found under **List Options**.

Mandatory field - The **Mandatory** field allows you to specify whether or not a field is required. If a required field is left blank in a record, the record cannot be saved. The ServiceCenter GUI automatically marks required fields by placing a small red triangle in the upper left corner of the edit field. Text clients cannot see a red triangle; however, mandatory fields are still enforced.

Mandatory fields are only enforced when entering records through the user interface. Records being added via events or via batch loads are not checked for mandatory fields.

Default Value - The **Default Value** field allows you to specify a default value for a particular field. This default value can either be a literal (e.g., *Bob)*, or an expression (e.g., *operator()*). The system differentiates between expressions and literals by looking for a caret (^) at the beginning of expressions.

Example:

`Bob`—field value will be set to **Bob.**

`^operator()`—field value set to the current operator.

`^lng(contact.name in $file)`—field value set to the length of the **contact.name** field

Note: Default value processing occurs before mandatory field testing.

Default values are applied to all fields, whether or not they are on screen.

Validation Rule - Validation rules are analogous to Format Control validations, except that they apply globally to an entire table, rather than being applied only to a particular form. Use the **Validation Rule** field to define a conditional expression which must be true in order for a record to be saved.

Example:

```
lng(contact.name in $file)>4
```

This rule would force all **contact.name** entries to have at least 4 characters.

Note: Validation rules are enforced both at the database level, -such as records added via events or batch loads, and at the GUI level,- when records are added through the user interface.

Not null is a valid validation and can be used to insure that a field isn't null; however, it is usually easier to set the Mandatory field value to *true*.

Match Fields - Use Match fields to define and enforce foreign key relationships. For example, we can dictate that the **contact.name** field in the `problem` file must match a valid **contact.name** from the `contacts` file. Matched fields are defined in two successive fields: **Match Field** and **Match File**.

Example:

Field Name	**Match Field**	**Match File**
Reported.by	contact.name	contacts

The above example shows that the **reported.by** field must match a valid **contact.name** from the `contacts` file.

Note: Matched fields are only enforced when entering records through the user interface. Records being added through events or batch loads are not checked for matched fields.

Invisible - Determines whether the field is invisible to a user on a form. True or an expression that evaluates to True indicates it is invisible, while False indicates the field is visible. The default is False.

Read-Only - Determines whether the field is read-only to a user on a form. True or an expression that evaluates to True indicates it is read-only, while False indicates the field is not. The default is False.

Encrypted - Set this field to True to encrypt data at the field level within the database. Data is encrypted on a field by field basis. Setting the field to False will revert the data back to its un-encrypted state in the database.

Important: If you have encrypted any field in your database, make sure that you store the KEY value defined in the *sc.ini* file in a safe place. Without the correct KEY value, you will not be able to decrypt.

When the Data Policy for a table is changed, the system checks to see if the encryption status has changed. If the status has changed from False to True then each record in the file is read, the field is encrypted, and the data written back to the file. If the status has changed from True to False then each record in the file is read, the field is decrypted, and the data written back to the file. The entire process takes as long as it takes to read and update each record in the file, and a performance loss may result.

Note: You can set the KEY value used to encrypt data on your system with the **encryptionkey** parameter in the *sc.ini* file. This field must be exactly 8 bytes in length.
If you change the KEY value, then every field in every file must be checked and re-encrypted, if necessary. Changing the KEY value will result in a significant performance hit to the system while re-encryption is taking place, which should be considered *before* changing the encryption key.

Changing the encryption key value

Changing the key value involves shutting down ServiceCenter:

1 Shut down ServiceCenter.

2 Restart ServiceCenter from the command line using the **changeencrkey** parameter. For example, **scenter -changeencrkey:XXXX** where XXXX is the new 8-byte key.

Starting ServiceCenter in this way will decrypt all encrypted fields using the key defined in the *sc.ini* file and then re-encrypt those fields using the key specified in the command line parameter **changeencrkey**. The length of time the conversion takes depends on the size of the database and the number of encrypted tables. You will need to update your *sc.ini* file to the new key immediately after performing this action.

Warning: Encrypting SQL data that is already mapped will increase the size of the data. Therefore, the existing SQL mapping and column definition may not provide enough space to store the whole encrypted value. If the encrypted value gets truncated, then the value can no longer be decrypted. Use this formula to calculate the new field length:
encrypted_length = (unencrypted_length + 12) * 2

If you convert a file which includes encrypted fields from P4 to SQL, the SQL mapping process will automatically take the increase of length into account.

Engine Specifications Tab

Common Name - Enter a name for the Data Policy record, or, if associating with an Object, use the Object's name. Entering the name of an Object in this field automatically associates the object to the Data Policy record.

Use Locking - With this field enabled, only one operator will be able to modify a record at a time. All users will still be able to open and view the record.

Revision File name - Enter the revision file name. When you first create a revision, a dbdict with the name you enter here will be stored on the system.

Max # of Revisions - How many revisions allowed on the system. The oldest revision stored will be the first overwritten when new revisions are saved and the Max # of Revisions limit has been reached.

IR Specifications tab

Condition - This field determines if a table should be searchable using IR. Enter True or an expression that evaluates to True to enable this feature. See the Data Administration guide for more information on IR searches.

Return Field - Enter the field to be populated back to the Data source. This is passed when "Use resolution" is chosen in the IR search.

Record Format - Enter the name of the form to use when showing a record. You can create different forms for different users and establish a default for each group.

QBE Format - Enter the name of the QBE form to use when showing record lists, i.e., lists of records selected as a result of a query. You can create different QBE forms for different users and establish a default for each group.

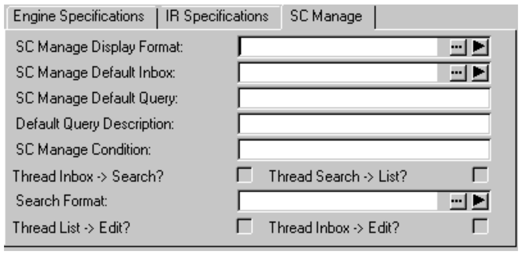
Search Format - Enter the name of the search form to use when searching record lists. You can create different search forms for different users and establish a default for each group.

MVS Allocation Type - MVS only. Choose between **block**, **trk** or **cyl**. You are allocating system resources for the IR query, with **cyl** or cylinder taking the most resources and Block taking the least.

Primary Allocation Space Size - Enter a number for the initial allocation space in terms of either Blocks, Tracks or Cylinders as selected in the **MVS Allocation Type** field.

Secondary Allocation Space Size - Enter a number that defines the overflow buffer, in Blocks, Tracks, or Cylinders as selected in the **MVS Allocation Type** field.

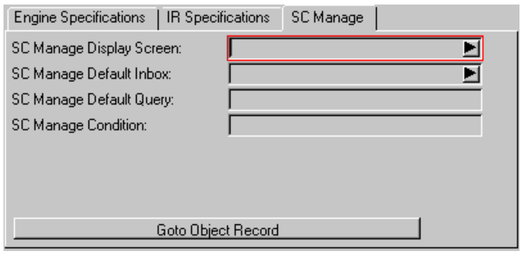
SC Manage tab



Engine Specifications	IR Specifications	SC Manage	
SC Manage Display Format:			
SC Manage Default Inbox:			
SC Manage Default Query:			
Default Query Description:			
SC Manage Condition:			
Thread Inbox -> Search?	☐	Thread Search -> List?	☐
Search Format:			
Thread List -> Edit?	☐	Thread Inbox -> Edit?	☐

Figure 11-4: SC Mange tab, no associated Object record.

The fields on the SC Manage tab control Queues and how they are displayed as well as threading and who can create inboxes. The SC manage tab can be used to control any file in the system. There are two versions of the SC Manage tab that will display, depending whether there is an Object associated with the Data Policy record. See the example figures shown.



Engine Specifications | IR Specifications | SC Manage

SC Manage Display Screen:

SC Manage Default Inbox:

SC Manage Default Query:

SC Manage Condition:

Goto Object Record

Figure 11-5: SC Mange tab, with associated Object record.

Modifying columns

The Modify Columns functionality has been added to the SC Manage queues.

To use the modify columns functionality:

1. Search for a list of records controlled by SC Manage, for example `caldaily`.
2. When the QBE list appears, select Options > Modify Columns. Or, if View > Record list is enabled, select List Options> Modify Columns.
3. Select which columns to appear in the QBE list and click Proceed.

Alternate column names for QBE lists

Alternate column names are specified in the Data Policy record for a file. The following example will modify a column heading on the Service Management QBE list:

To Modify a Column Heading:

1. From the ServiceCenter main menu, go to the Toolkit tab and select Data Policy.
2. Open the record you want to modify. In the name field, enter the name of the record and press Enter.

   For example, Service Management is controlled by the **incidents** Data Policy record. Type **incidents** in the name field and press Enter. The Data Policy record for the Incident file displays.
3. Within Data Policy, fields under the Caption column are used to display alternate headings for columns on QBE lists. Enter a new name in the Caption field and save the changes. For example, enter My Test in the first field name. (affected.item)
4. Log out of ServiceCenter and log back in.
5. Open the Service Management QBE list by performing a True search in Service Management.
6. Modify columns. If View > Record List is selected, then use List Options > Modify Columns to access the modify columns dialog. If View > Record List is not selected, then use Options > Modify Columns.
7. Add the new column name in the first slot. Using the pull-down menu available, change the current column name to My Test and click Proceed.

8 The new column name will appear the next time you view the QBE list. To check our example, perform a true search in Service Management. The column My Test will be the first column in the QBE list.

Fields on the SC Manage tab

SC Manage Display Format - This field determines how to display queues for data on a file by file basis. ServiceCenter ships with a default display format: `sc.manage.generic`. Peregrine recommends you do not modify the `sc.manage.generic` file.

SC Manage Default Inbox - State the default inbox for this queue. By specifying a user inbox record for a particular user, a specific list of inboxes can be set up for the SC Manage queues. If a user does not have a record, the DEFAULT user inbox record is used.

SC Manage Default Query - Enter a default query to run. Enter a query if you do not have an inbox defined in the Default Inbox field.

Default Query Description - Enter a description of the above field. You can associate a message with this field. For example, **scmsg(492, "us").**

SC Manage Condition - Enter a condition that allows only certain users to add queues as inboxes. For example, **index("SysAdmin", $lo.ucapex)>0**

Thread Inbox ->Search? - Check this box to open a new thread when moving from a List screen to a Search screen.

Thread Search -> List? - Check this box to open a new thread when moving from a Search screen to a List screen.

Search Format - Enter a default search format. The system will try to use the search State, if the this record is associated with an Object.

Thread List ->Edit? - Check this box to open a new thread in a new window when moving from a list screen to an Edit screen.

Thread Inbox ->Edit? - Check this box to open a new thread in a new window moving from an Inbox to an Edit screen.

12 Clocks

CHAPTER

Clocks allow you to track time in specific areas of ServiceCenter. This section uses incident tickets as an example. Incident tickets can be associated with multiple clocks, one clock or no clocks. Clocks allow you to track the following:

- The time an incident ticket spends in an incident state or different states.
- The time an operator spends editing a ticket.
- The time a ticket spends in an assignment group or multiple groups.

What is a Clock?

Clocks are based on records in the **clocks** Database Dictionary. Each record has the following fields:

Field Name	Data Type	Description
type	*character*	The type of clock, e.g., *problem* or *downtime*. All clocks associated with problem tickets have a clock type of *problem*.
name	*character*	The name of this clock, e.g., *Time spent in alert 3*.
key.char	*character*	An arbitrary character key used to associate a clock with a particular external record. All clocks associated with problem tickets store the problem number in this field.

Field Name	Data Type	Description
key.numeric	*number*	An arbitrary numeric key used to associate a clock with a particular external record. All clocks associated with problem tickets have NULL in this field. (In Incident Management, the problem number is a character field.)
total	*date/time*	The total time that this clock has been running. Note that this value may not always be accurate for clocks which are currently running.
events	*array*	Array of events.
events	*structure*	Event structure.
start	*date/time*	Date and time when this clock was started.
stop	*date/time*	Date and time when this clock was stopped. A clock may start and stop multiple times over its lifetime.

Clocks Example on page 197 provides an example of how clocks work with incident tickets. The following example uses three states: Open, Pending and Closed.

- Whenever an incident enters the Open state, a clock named **total.time** is started.
- Whenever an incident enters the Pending state, the **total.time** clock is stopped, and a clock named **pending.time** is started.
- Whenever an incident ticket leaves the Pending state, the **total.time** clock is started again, and the **pending.time** clock is stopped.
- Whenever an incident enters the Closed state, the **total.time** clock and the **pending.time** clock are both stopped.

1 **Applying this model to the example in Figure 12-1 on page 197:**

2 At 1:00 PM on July 1, *incident ticket 104* is created and saved in the Open state. A clock named *total.time* is created and started.

3 At 4:00 PM on July 2, the problem ticket is moved to the Pending state.

   - The **total.time** clock is stopped, after running for 27 hours.

- The *pending.time* is created and starts running.

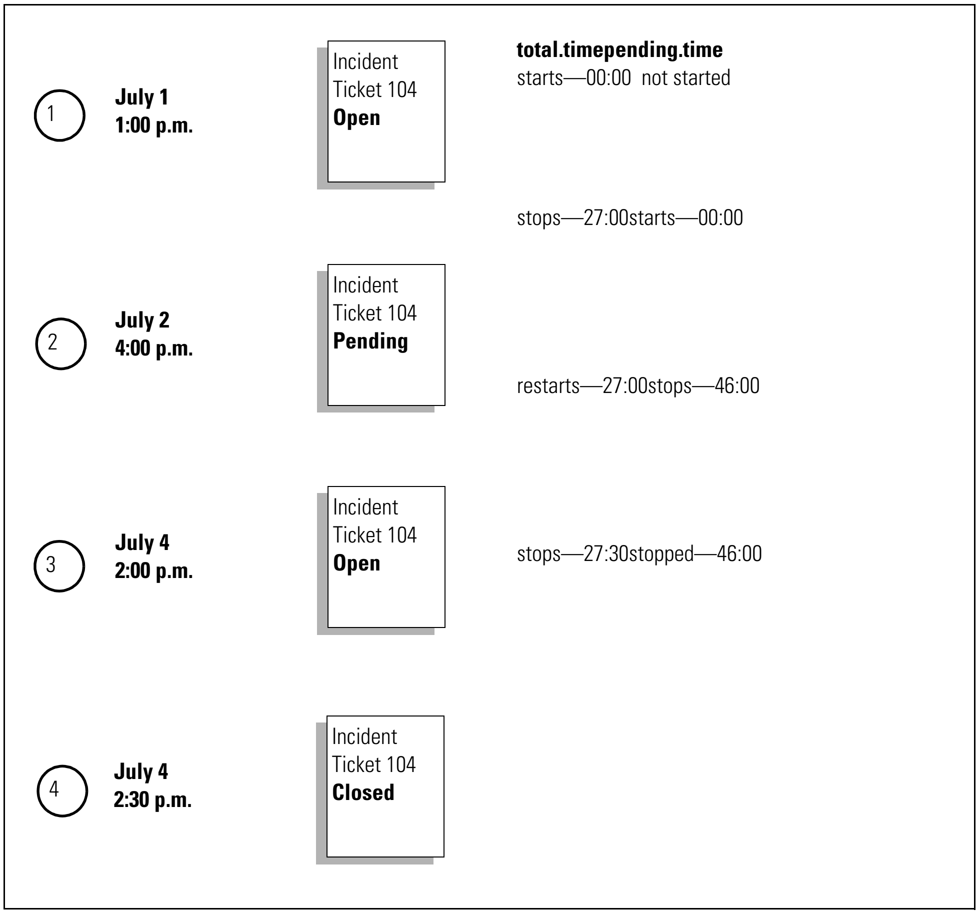


Figure 12-1: Clocks Example

4 At 2:00 PM on July 4th, the ticket is returned to the **Open** state.

- As the ticket leaves the **Pending** state, the pending.time clock is stopped, after running for 46 hours.
- As the ticket leaves the **Pending** state, the total.time clock is restarted.
- **Open** state, ServiceCenter instructs the total.time clock to start. Since the clock is already running, nothing happens.

5 At 2:30 PM on July 4th, the ticket is **Closed**.

- As the ticket enters the **Closed** state, ServiceCenter attempts to stop the pending.time clock, which is already stopped.
- As the ticket enters the **Closed** state, the system stops the *total.time* clock. This clock has been running for 30 minutes since being restarted.

6 The clock totals are:

- total.time—ran from 1:00 p.m. July 1st until 4:00 p.m July 2nd: 27 hours; and also ran from 2:00 PM July 4th until 2:30 PM July 4th for 0.5 hours. The total running time: 27 + 0.5 = 27.5 hours.
- pending.time—ran from 4:00 PM on July 2 until 2:00 PM on July 4, for a complete running time of 46 hours.

Starting and stopping clocks

ServiceCenter provides four methods for starting and stopping clocks. You can start and stop clocks by:

- Status changes.
- Editor tracking.
- Format Control.
- RAD changes.

Note: When a clock is started, a record is automatically created in the clocks file.

Starting and stopping clocks by status changes

In Incident Management, you can define an incident *status*. These status definitions are stored in the pmstatus file. Each time an incident ticket changes status, ServiceCenter checks for any clocks that need to be started or stopped. For example, if an incident ticket changes from Pending to Open, ServiceCenter checks for any clocks associated with the problem ticket that need stopping or starting.

To access a pmstatus record:

1 Select the **Toolkit** tab in the system administrator's home menu.

2 Click **Database Manager**.

3 Type pmstatus in the **File** field.

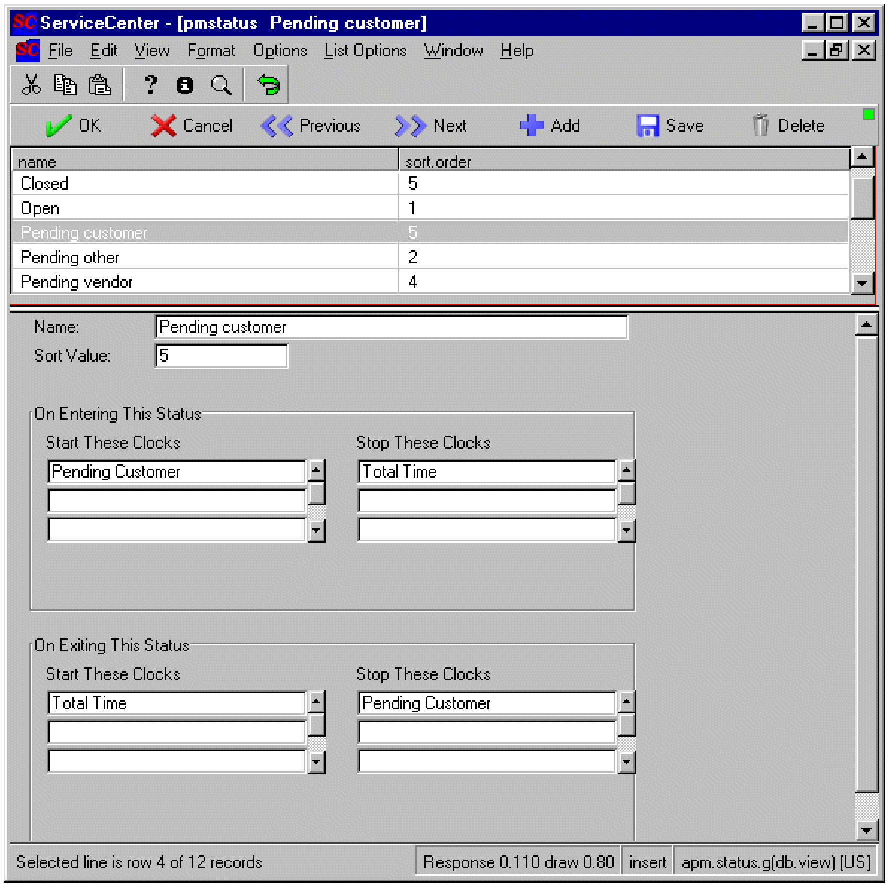
4 Click **Search**.

5 Select *apm.status.g* from the QBE list displayed.

6 Click **Search.**

7 Select the name of the status from the record list for which you want to set a clock.

For example, select **Pending customer.**



ServiceCenter - [pmstatus Pending customer]
File Edit View Format Options List Options Window Help
OK
Cancel
Previous
Next
Add
Save
Delete

name	sort.order
Closed	5
Open	1
Pending customer	5
Pending other	2
Pending vendor	4

Name: Pending customer
Sort Value: 5

On Entering This Status

Start These Clocks	Stop These Clocks
Pending Customer	Total Time

On Exiting This Status

Start These Clocks	Stop These Clocks
Total Time	Pending Customer

Selected line is row 4 of 12 records | Response 0.110 draw 0.80 | insert | apm.status.g(db.view) [US]

Figure 12-2: PMStatus Record

8 Modify the record as needed.

- Click **Save** or press F2 to save the changes made to the record.

  A message appears in the status bar stating: *Record updated in the pmstatus file.*

- Rename the record and click Add or press F5 to create a new *pmstatus* record.

PMStatus fields

Name—the name of the Incident Management status that will trigger the clock.

Sort Value—the order in which statuses are displayed in a combo box.

On Entering This Status

The clocks listed in this structure are affected when an incident ticket enters the status listed in the **Name** field.

Start These Clocks—clocks that you want Incident Management to start when a ticket enters the status indicated in the **Name** field.

In the example shown, each time an incident ticket enters the *Pending customer* status, a clock named *pending.customer* is started. This clock keeps track of how long the ticket remains in the *Pending customer* state.

Stop These Clocks—clocks you want Incident Management to stop when a ticket enters the status indicated in the **Name** field.

In the example shown, each time an incident ticket enters the *Pending customer* status, a clock named *total.time* is stopped. This clock keeps track of how long the ticket has been in states other than the *Pending customer* state.

On Exiting This Status

The clocks listed in this structure are affected when an incident ticket enters a status other than the status listed in the **Name** field.

Start These Clocks—clocks that you want Incident Management to start when a ticket enters this status.

In the example shown, each time an incident ticket exits the *Pending customer* status, a clock named *total.time* is started. This clock keeps track of how long the ticket has been in states other than the *Pending customer* state.

Stop These Clocks—clocks that you want Incident Management to stop when a ticket enters this status.

In the example shown, each time an incident ticket exits the *Pending customer* status, a clocked named *pending.time* is stopped. This clock keeps track of how long the ticket remains in the *Pending customer* state.

Stopping and starting clocks via editor tracking

ServiceCenter includes an option available in the Incident Management Environment record called **Track Operator Times?**. When this option evaluates to *true*, the system automatically starts a clock whenever an operator begins editing a record. The clock is stopped when the operator stops editing the record. The name of this clock is Time viewed by: <operator>, where <*operator*> is the user editing the record. For example if **falcon** is editing a record, the clock is named Time viewed by: falcon.

Stopping and starting clocks with Format Control

Clocks can be started and stopped through ServiceCenter's Format Control utility. Two RAD routines are used:

- **apm.start.clock** – Starts a clock
- **apm.stop.clock** – Stops a clock

To start or stop a clock from Format Control,

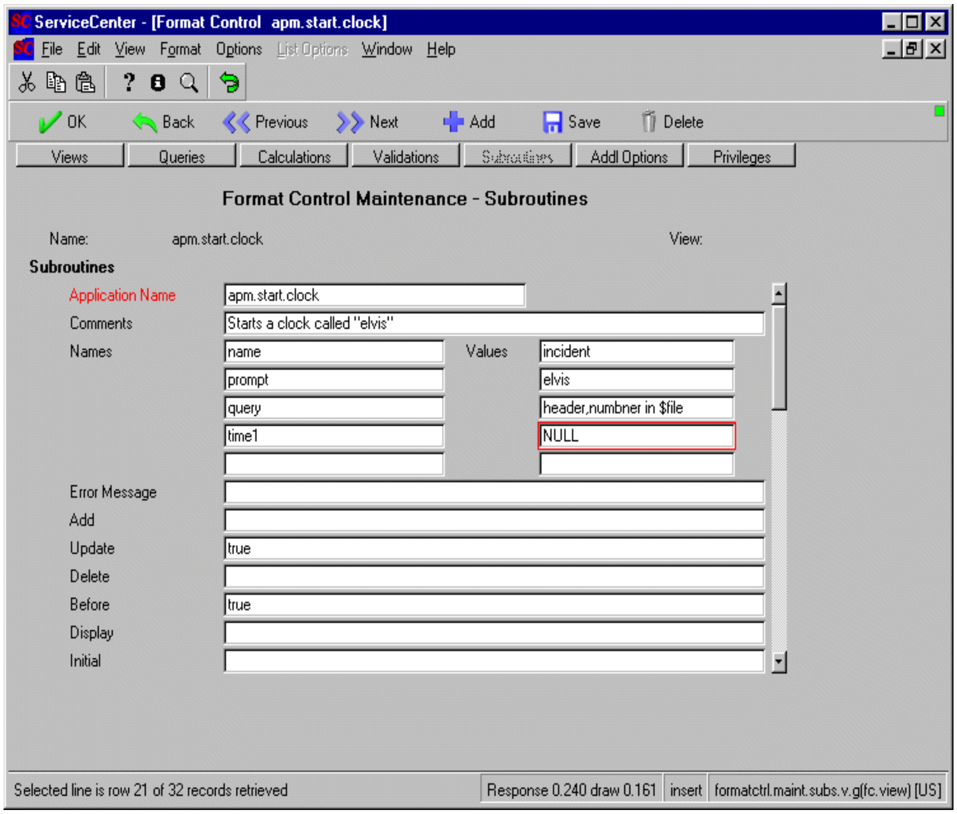
1. Invoke the appropriate subroutine.
2. Pass it the appropriate parameters.

Refer to the *ServiceCenter Format Control* guide for details on using this utility.

The following table describes the parameters that are entered in the **Names** and **Value** arrays for *apm.start.clock*:

Note: The apm.start.clock and apm.stop.clock records do not exist in the system. Create these Format Control records using the parameters shown in the table.

Names	Type	Description/Values
name	*character*	The *type of clock* to start all problem clocks are of type problem.
prompt	*character*	The name of this clock, e.g. elvis. The value you put in here needs to be unique within a given problem ticket. You cannot have two clocks named *boris* associated with the same problem ticket. The system would restart the existing boris clock when you try to open a second clock named boris. No requirement exists that clock names be unique between problem tickets. You could have several thousand problem tickets, each with a clock called Total Time.



ServiceCenter - [Format Control apm.start.clock]

File Edit View Format Options List Options Window Help

OK Back Previous Next Add Save Delete

Views Queries Calculations Validations Subroutines Addl Options Privileges

Format Control Maintenance - Subroutines

Name: apm.start.clock View:

Subroutines

Application Name	apm.start.clock		
Comments	Starts a clock called "elvis"		
Names	name	Values	incident
	prompt		elvis
	query		header,numbner in $file
	time1		NULL
Error Message			
Add			
Update	true		
Delete			
Before	true		
Display			
Initial			

Selected line is row 21 of 32 records retrieved | Response 0.240 draw 0.161 | insert | formatctrl.maint.subs.v.g(fc.view) [US]

Figure 12-3: Setting Clocks in Format Control

Names	Type	Description/Values
query	*character*	The unique character key for this clock. Pass in the problem number of the problem ticket with which you want this clock to be associated, e.g., header,number in $file.
time1	*date/time*	The time at which this clock is to start (defaults to current date and time). Note that putting in a time prior to the present rolls back all clock events since that time. Usually, you should leave this blank and allow it to default to *tod()*.

The following table describes the parameters that are entered in the **Names** and **Value** arrays for *apm.stop.clock*:

Names	Type	Description/Values
name	*character*	The *type of clock* to stop all problem clocks are of type problem.
prompt	*character*	The name of this clock, e.g. elvis. The value you put in here needs to be unique within a given problem ticket. You cannot have two clocks named *boris* associated with the same problem ticket. The system would stop the existing boris clock when you try to stop a second clock named boris. No requirement exists that clock names be unique between problem tickets. You could have several thousand problem tickets, each with a clock called total.time.
query	*character*	The unique character key for this clock. Pass in the problem number of the problem ticket with which you want this clock to be associated, e.g., header,number in $file.
string1	*character*	Either stop or strobe. If you want to stop a clock, set this value to stop. *Strobing* a clock simply forces it to recalculate its current running time.
time1	*date/time*	The time at which this clock is to stop (defaults to current date and time).

Starting and stopping clocks via RAD

You can start and stop clocks via ServiceCenter's RAD programming.

- Clocks can be started by calling **apm.start.clock**.
- Clocks can be stopped by calling **apm.stop.clock**.

Note: Clocks can be in various applications in Service Center other than Incident Management. If you choose to use a clock for an application other than Incident Management, ensure that you select a clock type that is not already in use.

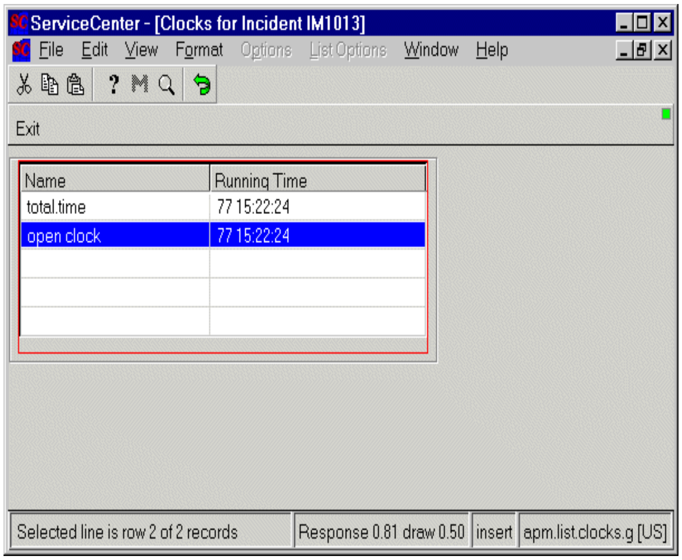
Accessing a clock

To view all the clocks associated with an incident ticket:

1 Click **Clocks** when *editing* an incident ticket.

A pop-up window is displayed listing all the clocks currently associated with this ticket and the time that each clock has been running.

2 Click **Clocks** again to update the display.


ServiceCenter - [Clocks for Incident IM1013]
File Edit View Format Options List Options Window Help
Exit

Name	Running Time
total.time	77 15:22:24
open clock	77 15:22:24

Selected line is row 2 of 2 records | Response 0.81 draw 0.50 | insert | apm.list.clocks.g [US]

Figure 12-4: Clocks

13 CHAPTER System Language

System language is the vocabulary ServiceCenter uses to communicate internally with its various routines and processes and interact externally with its users. Administrators and managers use this language daily to apply ServiceCenter to a specific purpose or enterprise. User access to various ServiceCenter operations is controlled with this syntax, as is the workflow itself. The key to ServiceCenter's remarkable flexibility can be found in the proper use of the system language.

Included in this chapter are:

- *Reserved Words* on page 213. – This section lists words with special meaning. These are words which cannot be used for any other purpose except that defined in RAD.
- *Rules for Forming Literals* on page 213. – This section lists all the explicit values (literals) accepted in RAD programming.
- *Rules for Forming Variables* on page 216. – This section defines variables as they are used in RAD programming.
- *Using Operators* on page 218. – This section defines and lists the *operators* used by ServiceCenter. Included are arithmetic, string, logical, relational, and special operators.
- *Using Expressions and Statements* on page 223. – This section defines expressions and statements and explains how they are formed.
- *RAD Functions* on page 225. – This section lists and defines all current RAD functions

Data Types Available in ServiceCenter

A data type characterizes a set of values and a set of operations applicable to those values. The values are denoted either by literals (page 213) or by variables (page 216) of the appropriate type; they also can be obtained as a result of operations. Data types can be either **primitive** or **compound**. Compound data types consist of several elements, each of a specific data type.

RAD has seven primitive data types and four compound data types.

Primitive Data Types

ServiceCenter internally represents each data type by its numeric representation. Certain functions (i.e., **val**() and **type**()) allow the user to determine or modify the numeric representation of a data type, thus changing the data type.

Name	Numeric Representation
Number	1
Character	2
Time (date/time)	3
Boolean (logical)	4
Label	5
Operator	10
Expression	11

Number

A number indicates an amount and can be either a whole number, a positive or negative number, or a floating point number. The following number formats display valid numbers accepted by RAD:

- 1 = 1.0 = 1E0 = 1.0E0
- -1 = -1.0 = -1E0 = -1.0E0
- 32.1 = 32.10 = 3.21E1
- -32.1 = -32.10 = -3.21E1
- 1.257E05 = 125700 = 1.25700E5

Character

A character type is a string of zero or more letters, digits, or special characters, and is delimited by double quotes (“ ”). The following character formats show valid strings accepted by RAD:

- “function”“12379”
- “Vendor473”“#$$^%+@”
- “$VEN471”“#73264”
- “vendor”

In addition, non-characters can be included in strings using their hexadecimal code preceded by a backslash.

Alternatively, any special or reserved character can be treated as a literal character by preceding it with a backslash. Use a double backslash to represent a literal backslash. For example:

```
$L.instruction="Enter the value \"securepassword\" in your
c:\\servicecenter\\run\\sc.ini file”
```

Date/Time

The date/time data type is delimited with single quotes (‘ ’). Time is represented in either absolute format (‘MM/DD/YY HH:MM:SS’) or relative format (‘DDD HH:MM:SS’).

- ‘10/12/90 12:12:46’
- ‘1 00:00:00’ (meaning 1 day)

The order of month-day-year is determined by the **set.timezone**() function. (See this function for possible values.)

Boolean

A boolean data type is true, false, or unknown (of undetermined truth value). The following formats are accepted:

- t = T= true = TRUE = y = Y = yes = YES
- f = F = false = FALSE = n = N = no = NO
- u = U = unknown = UNKNOWN

Label

Labels are used to define exits. For example:

- $exit
- $normal
- *<panel name>*

Operator

Arithmetic operators:

+, -, *, /, indicating addition, subtraction, multiplication and division

Expression

Expressions are sequences of *operators* and *operands* used to compute a value:

- $X * 25
- $Y > 32

Compound Data Types

You may combine primitive data types to form compound data types. Any compound data type can have elements of any other valid data types. The compound data types most commonly defined in the RAD language are as follows:

Data Type	Numeric Representation
offset	7
array	8
structure	9
file/record	6

Data Type	Numeric Representation
pseudo field	12
global variable	13
local variable	14

Array

An array is a list of elements of the same data type accessed by an index (element) number. The term *array* is synonymous with the terms *list*, *vector*, and *sequence*. Elements in arrays can be of any data types (including arrays or structures). A fully qualified array name (e.g., *array field in $file*) can be used in place of an array variable. The number of items in an array can vary and does not have to be allocated in advance. Arrays are delimited by curly braces ({ }).

For example:

- {1, 2, 3}
- {5, 12, 28}
- {“a”, “c”, “e”, “f”}
- {‘12/7/42 00:00,’ ‘1/3/62 00:00’}
- {true, true, unknown}
- {{[1,“a”,true]},{[2,“b”,false]},{[3,“c”,unknown]}}
- { } denotes an empty array (an array containing no elements).

You can use either of the following equivalent syntaxes to access an element in an array:

```
$array[element_number]
```

~ *or* ~

```
element_number in $array
```

For example, to extract the value of the first customer number (2753) in the array *$customer* with value {2753, 2842, 2963}, use *any* of the following equivalent syntaxes:

```
1 in $customer
```

~ or ~

```
$customer[1]
```

If the accessed array element does not exist, the element is created and set to NULL. This effectively extends the array.

To insert a value into an array, see the **insert** function. To delete an element from an array, see the **delete** function. Arrays should be denulled before they are assigned to a record or added to the database.

Structure

A *structure* is a group of named elements of (possibly) differing data types accessed by an index (element) number. Structures are delimited with both curly braces and square brackets ({[ ]}). Note the following examples:

- {[1,“a”,true]}
- {[true,‘10/16/90 00:00’,false]}
- {[1,1,“b“,0]}

You can use either of the following syntaxes to extract an element from a *structure*:

```
$structure[index of field]
```

~ or ~

```
index of field in $structure
```

For example, to extract the value of the part number (672) in the **part.no** field within the structure **$order.line1** with the value of [{672,10,“ball.bearings”]} with the field names **part.no**, **quantity** and **description**, use the following syntax:

```
$order.line1[1]
```

~ or ~

```
1 in $order.line1
```

In addition, elements of structures may be extracted using their element number. For example, the **1** in $order.line1 is the same as part.no in $order.line1.

Note: Field names are contained in the database dictionary and can only be used when associated with a file variable.

For example, you can use this command:

```
$x=part, part.no in $file
```

…while these commands,

```
$y=part in $file
$x=part.no Sin $y
```

…do not work because the second statement is not associated with a file variable.

File/Record

A *file* is a set of related records that is treated as a single unit and is stored on disk. (A file is a particular kind of structure, although a structure is not a kind of file.) The term *file* is synonymous with the terms *relation*, *table*, or *dataset*. Every file must have a file name and other descriptive data in the database dictionary, *dbdict*. The file name should be a single word without spaces or periods. Using the **rinit** command panel creates a file variable into which all or any portion of a file's *records* can be selected. The records then reside in memory for display or modification.

Note: File variables should be created on the **rinit** command panel.

A file variable has four parts:

- **File name**: The file name in the file variable is the name of the file in the database dictionary. If you want to use the RAD Database Manager, the file must be a single word without spaces or periods. The file variable is bound to the file with a given name using the **rinit** command panel. The file name of a file variable can be accessed using the **Name** pseudo-field (see Chapter 6), or using the **filename**() function.

- **List of records selected from the file:** This part of the file variable is a list of the records selected from the file. Execution of a **select** command panel retrieves the file's records from the database and sets up the list. Execution of a **fdisp** (File Display) command panel displays summary information from records in the list.
- **Current record:** This part of the file variable is the current record. You can move to the next record in the list by using the **next** (Read Next Record) command panel. You can move to the previous record in the list by using the **previous** (Read Previous Record) command panel. The **select** command panel always sets the current record to the first record that satisfies the selection query. The **fdisp** (File Display) Command Panel sets the current record to the record selected by the cursor when <**enter**> is pressed.
- **Descriptor:** This part of the file variable allows fields to be extracted by name.

Each record is a *structure* and has the structure delimiter {[ ]}. The field names in the structure are listed in the database dictionary and descriptor.

For example:

- *$file* is a file variable
- **lastname** is a field name listed in the database dictionary for the file that $file has been bound to using the **rinit** command panel,
- **lastname in $file** is a valid expression whose value is the **lastname** field in the current record in **$file**, or the nth field, based on the index of the **lastname** field in the descriptor.

Note: ServiceCenter is case sensitive. The case for field names, file names, etc. must match. For example, *STATUS.CUST* is not the same as *status.cust*.

See also the functions: **contents**, **descriptor**, **filename**, **file.position,** and **modtime.**

Reserved Words

Reserved Words are words recognized by RAD as having a specific meaning. You cannot use a reserved word for any other purpose or give it any other meaning. Reserved Words can be entered in either upper or lower case, but the cases cannot be mixed. The following is a list of RAD language Reserved Words:

AND	ELSE	FOR	NOT	STEP
WHILE	BEGIN	END	IF	NULL
THEN	UNKNOWN	DO	IN	OR
TRUE	FALSE	ISIN		

Rules for Forming Literals

A literal is an explicit value. You can use a literal wherever you can use a variable, except on the left-hand side of an assignment statement.

Literal	Category	Example
Character	Alphabetic	“abcde”
	Numeric	“12379”
	Alphanumeric	“a1276b45”
	Special	“$%=+*#@/”
	Mixed	“$vendor”
	Mixed	“$12379@6:54”
	Mixed	“A1276%#B”
	Hexadecimal	“\01\ff\c2”
	Containing backslash	“\\”
	Containing double quote	“\””
Number	Integer	1
	Negative Integer	-1
	Fixed Point	32.1

Literal	Category	Example
	Floating Point	1.257E05
Date/Time	Relative Time	‘197 05:00’
	Absolute Time	‘07/16/83 04:30:00’
Boolean	True	TRUE/true/T/t/YES
		/yes/Y/y
	False	FALSE/false/F/f/NO
		/no/N/n
	Unknown	UNKNOWN/unknown
		/U/u
Array	Numeric	{1,2,3}
	Character	{“a”,“b”,“c”}
	Nested	{{1,2},{3,4}}
	Structures	{{[1,"a"]}{[2,"b"]}}
Structure		{[1,“a”,true]}
		{[true,‘10/12/87 00:00’]}

Character Strings

A string is a sequence of zero or more numeric, alphabetic or special characters enclosed with double quotation marks. Rules for forming strings are as follows:

- A string can be any combination of letters, digits or special characters.
- All strings must be enclosed in double quotation marks.
- A string can be any length.
- A string may include any special characters including those used as operators.

- A double quotation mark may occur in a string, but it must be preceded by a backslash (\) to enable the system to distinguish between double quotation marks as part of the string and a double quote at the beginning or end of the string. For example, "`Vendor=\"473\"`" denotes the string: `Vendor="437"`
- Non-characters may appear in strings as two hexadecimal digits preceded by a backslash. For example, "`\c1`".
- NULL (or anything that evaluates to NULL), appended to or inserted in a string converts the entire string to NULL.
- An empty string is not the same as NULL

Numbers

Enter numbers as an optional sign followed by digits which are optionally followed by a decimal point, then more digits which are optionally followed by the letter **E** and an exponent.

Times

Time can be either relative or absolute. The following rules govern the use of time:

- Time must be enclosed in single quotation marks.
- Enter absolute time in the following format:
  `'MM/DD/YY HH:MM:SS'`
  (See the **set.timezone**() function for month-day-year order.)
- Enter relative time in the following format: `'DDD HH:MM:SS'`
- The use of seconds (**:SS**) is optional in either relative or absolute time. If you do not use seconds, the default is :00.
- The use of time (HH:MM:SS) is optional in absolute time. If you omit this syntax, time defaults to 00:00:00 (i.e. `'MM/DD/YY 00:00'`). Time is recorded on a 24-hour clock, i.e., `'00:00:00'` is midnight, `'12:00:00'` is noon, `'18:00:00'` is 6 p.m.
- The use of **DDD** is optional in relative time. If you omit it, DDD defaults to **0** (i.e. 'HH:MM').
- Use the following to set a time variable to a 0 (zero) value:
  `$time = '00:00'`

Booleans

A boolean indicates true, false or unknown (of undetermined truth value). The rules for forming booleans are as follows:

- Enter the boolean directly as **true**, **false**, **unknown** (**t**, **f** or **u**).
- Enter the boolean in either upper or lower case characters.
- **yes** or **y** is a synonym for **true**; **no** or **n** for **false**

Rules for Forming Variables

A variable is a named entity that refers to data to which you can assign values. The data type and value of a variable can be different at different times and can have a primitive or compound data type as its value. The rules for forming variables are as follows:

- A variable must begin with a dollar sign ($) and an alphabetic character followed by zero or more alphanumeric characters that may include periods, but may not include blanks.
- A variable name may be any length.
- The value of any variable is initialized to a null string.
- The data type of a variable is determined by the system as the type associated with the value of the data assigned to it.
- Any variable may be set to a null value by assigning the value NULL into it (i.e., `$variable=NULL`).

Note: Unless designated by an initial identifier, all variables, with the exception of parameter variables, are treated as global variables among all applications executed within the process or task.

Using Variables

The rules for using variables are as follows:

- You may use any number of variables in an application.
- You can use a variable in any field in an application (RAD) panel except in the **application** field and the **label** field.

Variable Pools

Variable pools are functional groupings of variables within ServiceCenter. There are currently five variable pools:

- Global
- Thread
- Local
- Parent
- Parameter

Global

Global variables are visible to the entire system. ServiceCenter uses four global variables:

- $G.
- $lo.
- $SYSPUB.
- $MARQUEE.

Thread

Thread variables are only visible to the thread in which they were defined. The same variable in different threads will have a different value, even if the threads were spawned by the same *parent*. Examples of Thread variables would be:

- $file
- $array
- $post

Local

A *local* variable (**$L.**) is only visible to the RAD application in which it was defined. There is only one local variable—**$L.**

Parent

A parent variable (**$P.**) defines the value of a variable for multiple threads of a single parent.

For example, a variable in *thread0,* called **$source.file**, has the same value in *thread1* or *thread2* when defined as **$P.source.file** in those threads.

Parameter

A parameter variable is defined on a parameter panel and may contain a value passed in from another application. By convention, parameter variables are written in uppercase letters, such as **$PHASE** or **$GROUP.LIST**. Parameter variables are invisible to the debugger.

Using Operators

ServiceCenter uses several different kinds of plenary *operators*:

- Arithmetic Operators
- String Operators
- Logical Operators
- Relational Operators
- Special Operators

Arithmetic Operators

An *arithmetic operator* indicates actions to be performed under the terms of an arithmetic expression. Arithmetic operators have the following precedence: exponentiation, followed by multiplication, division and modulus (all equal); then addition and subtraction (both equal). Operators with higher precedence are evaluated first. When the operators have equal precedence, they execute from left to right.

Operator	Description
Addition (+)	Indicates that two numbers are to be added together Example: *49 + 51 = 100*
Subtraction (–)	Indicates that one number is to be subtracted from another number Example: *40 – 20 = 20* **Note:** To distinguish subtraction from unary minus, the subtraction operator must be followed by a blank.
Multiplication (*)	Indicates that one number is to be multiplied by another number Example: *5 * 5 = 25*

Operator	Description
Division (/)	Indicates that one number is to be divided by another number Example: *300 / 10 = 30*
Exponentiation (**)	Indicates that the exponential value of a number is to be calculated Example: *2 **5 = 32*
Modulus (mod) or (%)	The modulus is the remainder of a division operation. You may specifically want the remainder for a division operation, or you may want to generate a circular number sequence within a given range. Example: *5 mod 2 = 1 or 5 % 2 = 1*

String Operators

String operators allow you to combine two strings into a single string (*concatenate*). One string operator is available: **Concatenation Operator**.

Concatenation (+)

This operator combines two strings together to make one string.

Example

```
"a" + "b" = "ab"
```

Concatenation is also available for arrays.

Example

```
{"a", "b", "c",} + {"d", "e"} = {"a", "b", "c", "d", "e"}
```

Logical Operators

A *logical operator* evaluates one or two boolean expressions and determines whether the expression is true or false. The unknown truth value is treated according to the *Substitution Principle*, which dictates:

- If UNKNOWN occurs as a logical operand, then the result of the operation is TRUE if substituting TRUE or FALSE for UNKNOWN always yields TRUE.

- The result is FALSE if substituting TRUE or FALSE for UNKNOWN always yields FALSE.
- The result is UNKNOWN if substituting TRUE or FALSE for UNKNOWN sometimes yields TRUE and sometimes yields FALSE.

The Logical Operators are as follows (executed in the order shown):

- not
- and
- or

The highest precedence is executed first. When they are equal, the operators are executed from left to right.

Logical Operator	Description
not: ¬ (in EBCDIC) or ~ (in ASCII)	Inverts the boolean value of the boolean expression. If the expression is true, the system returns FALSE. If the expression is false, the system returns TRUE. For example: not TRUE = FALSE ¬ FALSE = TRUE (MVS) ~ FALSE = TRUE (Unix) UNKNOWN = UNKNOWN ~ UNKNOWN = UNKNOWN
and: (AND and &)	Evaluates two expressions and returns a value of TRUE if both expressions are true. If one or both of the expressions is false, the system returns FALSE. For example: TRUE and TRUE = TRUE TRUE and FALSE = FALSE TRUE and UNKNOWN = UNKNOWN FALSE and UNKNOWN = FALSE
or: (OR or |)	Evaluates two expressions and returns a TRUE if either or both of the expressions is true. If both expressions are false, it returns a FALSE. For example: TRUE or FALSE = TRUE FALSE | FALSE = FALSE FALSE OR UNKNOWN = UNKNOWN TRUE OR UNKNOWN = TRUE

Relational Operators

A *relational operator* makes a comparison, then generates logical results on whether the comparison is true or false.

Relational operators treat null operands according to the NULL substitution principle:

- If the relation is true regardless of the value of the null operand, the result is true.
- If the relation is false, the result is false.
- Otherwise, the result is unknown.

An operand can be checked for null using the special value NULL; `$x=NULL` is true if `$x` is null.

Rational Operator	Description
Less Than: (<)	Indicates that the value of one item is less than the value of another item. For example: 400 < 500 is TRUE
Less Than or Equal To: (< = or = <)	Indicates that the value of one item is less than or equal to the value of another item. For example: 400 < = 500 is TRUE 400 = < 500 is TRUE
Equal To: (=)	Indicates that the value of one item is equal to the value of another item. For example: 1 = 1 is TRUE
Greater Than: (>)	Indicates that the value of one item is greater than the value of another item. For example: ‘08/01/83 00:00’ > ‘07/20/83 00:00’ is TRUE
Greater Than or Equal To: (> = or = >)	Indicates that the value of one item is greater than or equal to the value of another item. For example: 600> =300 is TRUE 600= >300 is TRUE

Rational Operator	Description
Not Equal To: (¬ =) or (~=) or (< >) or (> <)	Indicates that the value of one item is not equal to the value of another item. For example: 1 ¬ =2 is TRUE (MVS) or 1 ~=2 is TRUE (Unix)
Starts With (Truncated Equals): (#)	Indicates that the value of the first string starts with the value of the second string. For example: “abc“#“ab” is TRUE **Note:** The order of the operands affects this operations.
Does Not Start With (Truncated Not Equal To): (¬#) or (~#)	Indicates that the value of the first string does not start with the value of the second string. For example: “ab” ¬ #“abc” is TRUE (MVS) “ab” ~ #“abc” is TRUE (Unix) **Note:** The order of the operands affects this operation.

Special Operators

ServiceCenter supports the use of two special operators. These are as follows:

- Statement Separation
- Parentheses

Special Operator	Description
Statement Separation: (;)	This operator separates two or more statements on the same line. For example: $A=$B+$C;$B=$C+$D
Parentheses: ( )	This operator groups together expressions or statements. For example: 3*($x + $y) IF ($x=1) THEN ($y=“z”) ELSE ($y=3) IF ($x=1) THEN ($x=2;$z=1) ELSE ($y=3) ServiceCenter follows the standard order of operations: operators inside the parentheses are evaluated first. and parentheses themselves are evaluated from left to right.

Using Expressions and Statements

An *expression* is a combination of one or more operations used in combination with functions. An expression may be a literal or a variable. An expression can also be used in combination with operators or used with a function call.

A *statement* does not have a value. A statement is comprised of expressions combined with key words. ServiceCenter uses three statements from the BASIC language: **IF**, **WHILE** and **FOR**. These statements perform conditional processing and looping. In addition, ServiceCenter uses three assignment statements: **assign**, **increment,** and **decrement**.

Assignment Statements

Assignment Statement	Description
Equals: (=)	Assigns the value of the right hand operand to the left hand operand. For example: $x=1 assigns 1 to $x
Increment: (+ =)	Increment the left hand operand by the right hand operand. For example: $x=$x+1 or the shortcut version is: $x += 1 increments $x by 1
Decrement: (- =)	Decrement the left hand operand by the right hand operand. For example: $x=$x-1 or the shortcut version is: $x -= 1 decrements $x by 1 **Note:** For increment and decrement, $x (the 1 value) must be initialized to a number.

FOR Statements

Allows you to perform a loop. You can set a variable, perform a statement, and increment the variable until the variable is greater than a maximum value. Format this statement as follows:

FOR *variable name* = *initial value* TO *maximum value* [DO] *statement*

Example:

```
FOR $I=1 TO 10 DO $J=$I * $I+$J
```

Note: The brackets ([ ]) indicate that the DO keyword is optional.

IF Statements

Specifies a condition to be tested and a statement to be executed if the condition is satisfied and a statement to be executed if the condition is not satisfied. Format these statements as follows:

IF *boolean condition* THEN *statements* [ELSE ***statements***]

Example

```
IF $location = “Seattle” THEN $x = $x + 1 ELSE $x = $x - 1
```

Note: If **boolean condition** evaluates to UNKNOWN, then neither **statement** is executed.

The brackets ([ ]) indicate that the ELSE clause is optional.

WHILE Statements

Specifies a condition to be tested and a statement to be executed when the condition is TRUE. Format these statements as follows:

WHILE (*expression*) [DO] *statement*

Example:

```
WHILE ($x>6) DO ($x=$x-1; $y=$y-1)
```

Note: The brackets ([ ]) indicate that the DO keyword is optional.

FOR, IF and WHILE statements can be nested (e.g., for…to…if…then…else…if…then…)

RAD Functions

A RAD function is an *operand* used in any expression on the right hand side of an equals sign. Functions provide a method of performing certain commands that will return a value when executed. For example:

`operator()`—returns the logged-on operator ID.

`tod()`—returns the current date and time.

`option()`—returns the value of the last option selected.

RAD supports a complete set of business data processing functions. This section provides a list of the supported functions, a brief definition of each, and a detailed description of their use.

Processing Statements

Processing statements can be used in conjunction with RAD functions to further define selection criteria and/or execute commands, initialize values, and perform calculations. ServiceCenter syntax rules must be followed.

Locating Functions

To locate a RAD function in ServiceCenter

1. Open the Database Manager.
2. Type `=application` in the Form field of the Database Manager dialog box.
3. Click Search or press Enter.

   A blank application file record is displayed.
4. Select **Options** > **Advanced Search**.

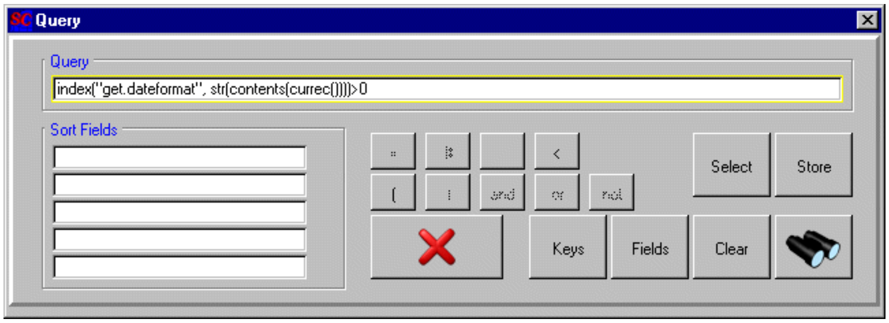
   A query window is displayed.
5. Type one of the following queries in the **Query** field:
   - index("<*function name*>", str(contents(currec())))>0

     Use this syntax to search for any non-rtecall function.
   - index("rtecall(\"<*function name*>\"", str(contents(currec())))>0

     Use this syntax to search for all instances of a particular **rtecall** function. For this query to run properly, a backslash must precede each double quotation mark within the **rtecall** parentheses.

- index("rtecall(\"<*first letter*>", str(contents(currec()))))>0

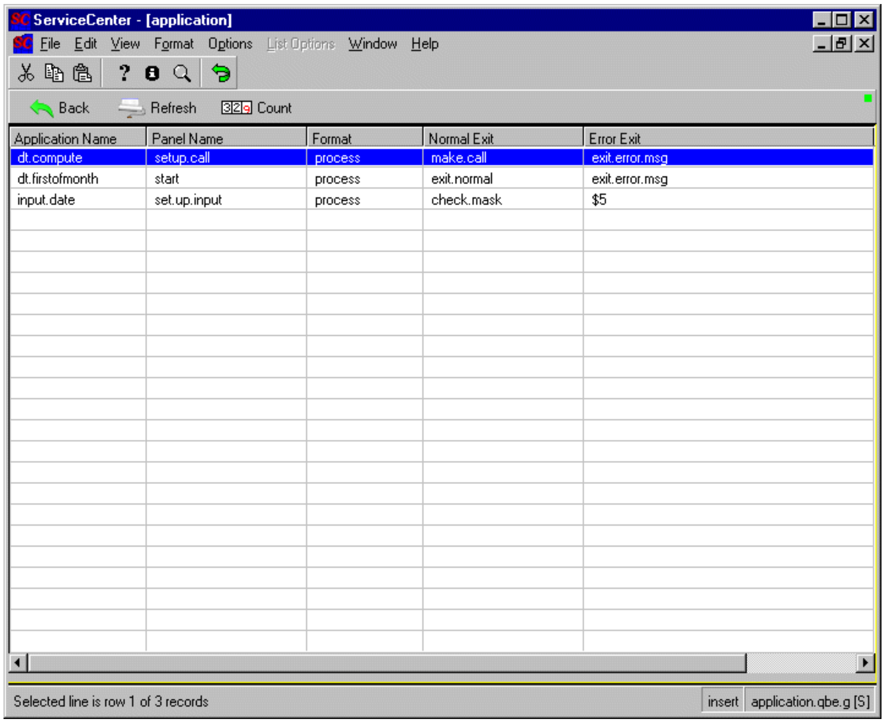
  Use this syntax to search for all instances of all **rtecall** functions beginning with a particular letter (for example, the letter **r**). For this query to run properly, a backslash must precede the first set of double quotation marks. The closed parenthesis is not necessary in this syntax.



index("get.dateformat", str(contents(currec())))>0

6 Click Search.

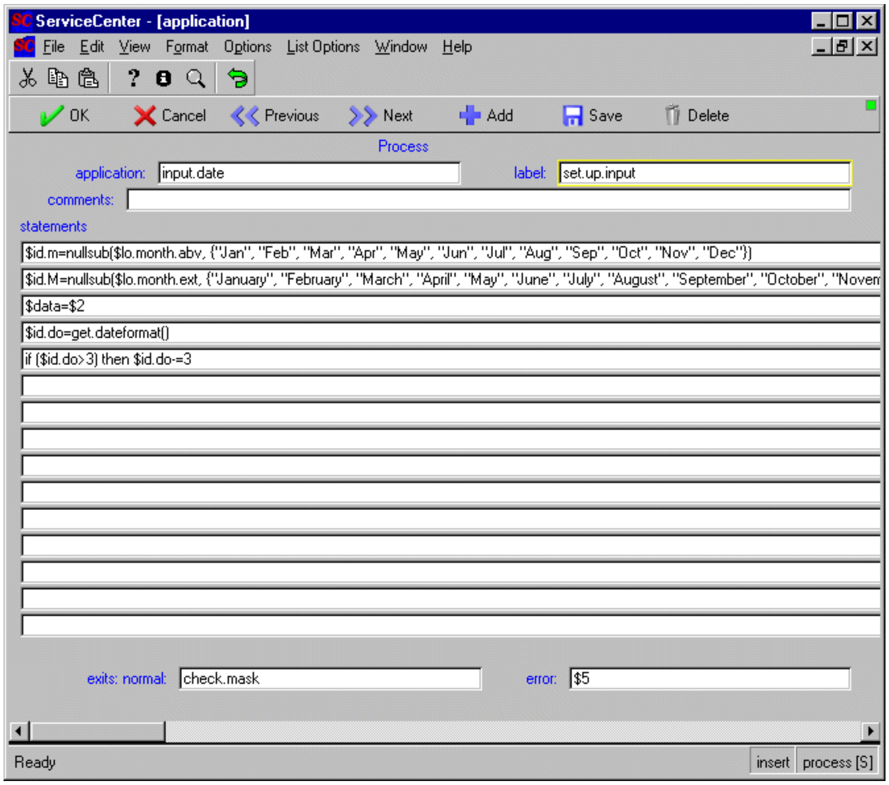
A QBE list of RAD panels in which the function you have named is displayed.


ServiceCenter - [application]
File Edit View Format Options List Options Window Help
Back
Refresh
Count

Application Name	Panel Name	Format	Normal Exit	Error Exit
dt.compute	setup.call	process	make.call	exit.error.msg
dt.firstofmonth	start	process	exit.normal	exit.error.msg
input.date	set.up.input	process	check.mask	$5

Selected line is row 1 of 3 records   insert   application.qbe.g [S]

7 Double-click on a panel to display that instance of the function.


ServiceCenter - [application]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete
Process
Ready
insert
process [S]

application: input.date
label: set.up.input
comments:
statements

```
$id.m=nullsub($lo.month.abv, {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"})
$id.M=nullsub($lo.month.ext, {"January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "Novem
$data=$2
$id.do=get.dateformat()
if ($id.do>3) then $id.do-=3
```

exits: normal: check.mask
error: $5

Quick Reference List

Function	Description
axis	Returns a string with the boundaries for a bar or point plot graph (page 235)
cleanup	Frees the storage associated with a variable (page 236)
contents	Returns a structure containing the current record in a file variable (page 236
copyright	Returns a string with the Peregrine Systems, Inc., copyright notice (page 237)
currec	Represents the current file handle in queries (page 237)

Function	Description
current.device	Returns a string with the name of the device associated with the current task (page 237)
current.format	Returns a string containing the name of the current format (page 238)
current.screen	Returns an array of strings containing the current screen image (page 238)
current.scroll	Returns and sets the current scroll state (page 238)
current.window	Returns a string containing the name of the currently selected window. (page 239)
cursor.column	Returns the number of the column in which the cursor was positioned when the last interrupt key was pressed (page 240)
cursor.field.contents	Returns a string containing the contents of the input field in which the cursor was positioned when the last interrupt key was pressed (page 240)
cursor.field.name	Returns a string containing the name of the field in which the cursor was positioned when the last interrupt key was pressed (page 241)
cursor.field.name.set	Moves the cursor to a field (page 242)
cursor.field.readonly	Returns *true* if the selected field is read-only and *false* if the field is not read-only (page 242)
cursor.filename	Returns a string containing the name of the file that is associated with the format where the cursor is positioned (page 243)
cursor.line	Returns the number of the line, relative to the screen, in which the cursor was positioned when the last interrupt key was pressed (page 243)
cursor.window	Returns a string containing the name of the window in which the cursor was positioned when the last interrupt key was pressed (page 243)
date	Returns the date portion of a date/time variable and defaults time portion to ‘00:00’ (page 244)
day	Returns the day of the month for a date regardless of the date format (page 244)
dayofweek	Returns a number from 1 to 7 representing the day of the week for a specific date (page 245)

Function	Description
dayofyear	Returns the day of the year for a date regardless of the date format (page 245)
delete	Returns an array with specific elements deleted (page 245)
denull	Returns an array with all trailing NULL entries deleted (page 246)
descriptor	Returns the database dictionary descriptor record for a file variable (page 246
evaluate	Executes a specific string as though it were a processing statement (page 247)
exists	Checks for the existence of a field in a file (page 247)
fduplicate	Copies an entire file variable from one record to another (page 248)
filename	Returns a string containing the name of the file for a specified file variable (page 248)
file.position	Returns an index into the record list for various records (page 249)
filesize	Returns the size (in bytes) of a specified system file of your ServiceCenter implementation (page 250)
filesizes	Returns an array of numbers representing the bytes for each of the files that make up a Service Center database (page 250)
fillchar	Assigns/retrieves a field input character other than underscore (page 251)
fixed.key	Returns and sets fixed key function (page 251)
frestore	Restores all fields in a file variable to their original database values (page 252)
genout	Generates a string containing the contents of a format used to export ServiceCenter database information. It produces either fixed or variable length output (page 252)
get.base.form.name	Returns the base name of a form, stripping off the form extension for GUI or Web forms (page 253)
get.dateformat	Returns the date format of the current operator ID (page 254)

Function	Description
get.timezoneoffest	Returns the absolute time difference between GMT and the time zone of the operator (page 254)
gui	Determines whether or not ServiceCenter is running in GUI mode (page 256)
index	Returns the element number of an array or the position in a string that matches a specified string (page 256)
insert	Returns a NULL element into an array (page 257)
iscurrent	Determines if the record with which you are working is the most current version in the database (page 259)
lng	Returns the number of elements in an array or characters in a string (page 259)
locks	Returns array of current outstanding locks (page 259)
logoff	Logs a user off (page 260)
logon	Logs a user on (page 261)
mandant	Allows an application to create a *subset* of the current Mandanten values (page 261)
max	Returns the largest value in a list of values or in an array (page 262)
messages	Provides logging functions for use in the memory message log. The log is implemented as a wrap-around cache of the x most recent messages (error, informational, and action) as displayed at the bottom of the screen in the **message** field (page 262)
min	Returns the smallest value in a list of values or in an array (page 264)
modtime	Returns the time a record was last modified (page 265)
month	Returns the month of year for a date regardless of the date format (page 265)
null	Returns true if the value of a variable is NULL. Returns false if the value of a variable is not NULL (page 266)
nullsub	Substitutes a null field with the value given (page 266)
operator	Returns the string of the name of the currently logged on operator (page 267)
option	Returns the number of the last option key pressed (page 267)

Function	Description
parse	Returns an evaluative expression from a given string value (page 268)
pfdesc	Returns a string containing the description of the option key number provided (page 269)
pfmap	Returns array of remapped option keys (page 269)
printer	Returns a string of the name of the current printer (page 270)
priority	Lowers or raises the priority of a task (page 270)
processes	Returns array of current logged on processes (page 271)
prof	Returns the value requested on various system performance profiles (page 272)
recordcopy	Copies a set of fields from one record to another record (page 273)
recordtostring	Takes the value of a field from an array and appends the value to a string (page 273)
replicate	Replicates a named file from a remote site to the current site (page 274)
round	Returns a number rounded to a specified number of digits (page 275)
rtecall("alalnum")	Checks to make sure a string contains only alphanumeric characters, or only alphanumeric characters and the provided non-alphanumeric characters (page 276)
rtecall("alnum")	Checks to make sure a string contains only numeric characters, or only numeric characters and the provided non-numeric characters (page 276)
rtecall("alpha")	Checks to make sure a string contains only alphabetic characters, or only alphabetic characters and the provided non-alphabetic characters (page 278)
rtecall("counter")	Turns counters on or off for the current session. Other SC users are unaffected (page 279)
rtecall("datemake")	Returns a date, in the proper form, based upon a series of numbers passed to it (page 279)
rtecall("escstr")	Precedes special characters in a string with an escape character (page 281)

Function	Description
rtecall("FILLDATE")	Places the current date and time in a field in the current record (page 282)
rtecall("filecopy")	Copies all of the data in a collection to another file variable (page 283)
rtecall("fileinit")	Initializes a new file (rinit) in $targetfile (page 284)
rtecall("getnumber")	Replaces the getnumb RAD application (page 284)
rtecall("getrecord")	Retrieves the record identified by unique key values in $L.array (page 286)
rtecall("getunique")	Returns into $L.array the values for the unique key from the current record in $L.file (page 286)
rtecall("log")	Sends a message to the external sc.log file (page 287)
rtecall("notypecheck")	Turns type checking off or on (page 289)
rtecall("passchange")	Changes this user's password (page 290)
rtecall("policycheck")	Imposes data policy as defined in the datadict table (page 291)
rtecall("qbeform")	Returns a QBE form, which can be passed into an rio or fdisp panel (page 291)
rtecall("radhistory")	Keeps track of the RAD panels a user has executed when running ServiceCenter applications (page 292)
rtecall("recdupl")	Copies the contents of the current record into the contents of another record (page 293)
rtecall("rfirst")	Places the pointer at the first record in a record collection (a QBE list) (page 295)
rtecall("rgoto")	Places the pointer at the indicated record.id in a record collection (a QBE list) (page 295)
rtecall("rid")	Returns the record number of the current record (represented by $L.file) (page 296)
rtecall("sort")	Sorts a list or a list of lists in ascending or descending order (page 297)
rtecall("transtart")	Measures the amount of data transferred, elapsed time and CPU usage of any transaction (page 298
rtecall("transtop")	Measures the amount of data transferred, elapsed time and CPU usage of any transaction. It is commonly invoked from the GUI debugger and is used in conjunction with transtart (page 299)

Function	Description
rtecall("trigger")	Turns triggers on or off for the current session (page 300)
same	Compares two compound or null values. Returns true if two values are identical, otherwise returns false (page 301)
scmsg	Returns a message of a particular type and number from the scmessage file and substitutes text for the variables in that message (page 303)
set.timezone	Sets time zone and other system parameters (page 304)
setsort	Sorts the fields of an array in a file (page 304)
share	Shares the data in the current record with the same file at several remote sites (page 304)
shutdown	Shuts down ServiceCenter from inside the system. This function does *not* return any values (page 305)
str	Returns a string of a string or non-string data variable (page 305)
stradj	Makes a string a specific length by either clipping or adding trailing blanks (page 305)
strchrcp	Replaces part of a string with copies of another string (page 306)
strchrin	Inserts copies of a string into another string (page 307)
strclpl	Clips a number of characters from the beginning of a string (page 307)
strclpr	Clips a number of characters from the ending of a string (page 308)
strcpy	Replaces part of a string with a substring (page 308)
strdel	Deletes a number of characters from a specific spot in a string (page 309)
strins	Inserts a substring into another string (page 310)
strpadl	Pads a string with leading blanks until the string is a certain size (page 310)
strpadr	Pads a string with trailing blanks until the string is a certain size (page 311)
strrep	Returns a string, replacing a specified character sequence with another string (page 312)

Function	Description
strtrml	Removes all leading blanks from a string (page 312)
strtrmr	Removes all trailing blanks from a string (page 313)
substr	Returns a string from a portion of a string (page 313)
sysinfo.get	Returns various information about the session the user is running (page 314)
time	Returns the time portion of a date/time variable (page 317)
tod	Returns the current date and time (page 318)
tolower	Returns a string, replacing upper-case letters with lower-case letters (page 318)
toupper	Returns a string, replacing lower-case letters with upper-case letters (page 319)
translate	Returns a string translated from one character set to another (page 319)
trunc	Truncates the decimal digit number to a specified number of digits (page 320)
type	Returns the numeric type of its argument (page 320)
val	Returns a non-string data type converted from a string (page 321)
variable.send	Sets the value of a RAD variable on the server when using a full client (page 323)
version	Returns a list of version information (page 323)
year	Returns the full year for a date regardless of the date format (*page 324*)

Function Definitions

axis

Returns a string with the boundaries for a bar or point plot graph (i.e., the lowest, mid, and highest point in the array to be plotted).

Format

```
axis(array variable,n)
```

Where *array variable* is the variable name for the array that contains the data to be graphed, and *n* is the length of the display field on the format where the graph appears.

Factors

- The array variable may contain all numeric or all time values. The system ignores all other values.
- Mixing numeric and time values produces incorrect results because numeric and time values cannot be compared.
- The array variable must contain at least 2 values, insert ý if the length is only 1.

Example

```
$graph = axis($balance,61)
```

Where *$balance* contains numeric values to be graphed in a format field of *61* characters, and *$graph* contains the minimum, middle, and maximum value in the array, $balance. To illustrate the $graph values:

```
$balance = {2, 4, 6, 8, 10}
$graph = axis($balance, 10)
$graph would then = "2  6  10"
```

Note: The display engine no longer supports display of histograms or point plots.

Where 2 represents the minimum value in $balance, 6 represents the middle value in $balance, and 10 represents the maximum value in $balance. The extra blanks between the 2 and 6, and 6 and 10 are placed there to make the whole string equal to 10. If the string were to be made equal to 12, there would be four blanks between the numbers. The numbers represented in the character string $graph can be used for plotting purposes.

Note: If the length (*n*) specified is not large enough to return the three values with at least one blank between each value, the result will be a character string of *n* length of asterisks.

Example

```
$balance = {2, 4, 6, 8, 10}
$graph = axis($balance, 4)
$graph would then = "****"
```

cleanup

Frees the storage associated with a variable.

Format

```
cleanup()
```

Where the variable is the variable name to be freed.

Factors

- This function does not return a value, but simply executes the cleanup of the variable name.
- Should not be run on local variables.
- This is the preferred way to clean up both variables and fields in records (as opposed to $x=NULL or field in $file=NULL.

Example

```
cleanup($system)
```

contents

Returns a structure containing the contents of the current record in a file variable.

Format

contents(*file variable*)

Where *file variable* is the file variable whose contents you wish to obtain.

Factors

- If you use the *contents()* function to compare two records (e.g., *before* and *after* versions), always do a denull of both. If one record was displayed and the other was not, one record may have extra null values in any arrays.

- Using denull on an empty array of structures converts the array of structures to a simple array. Always test for null first as in the following example:

```
denull(contents($save.rec))=denull(contents($rec))
```

Example

```
contents($file.variable)
```

copyright

Returns the Peregrine Systems, Inc., copyright notice in a string.

Format

```
copyright()
```

Example

```
copyright() returns Copyright(c) Peregrine Systems, Inc. 2002-2003
```

currec

Represents the current file handle on queries.

Format

```
currec()
```

Example

```
modtime(currec())>'1/1/97'
```

Used as a select statement, selects all records modified since 1/1/97 from the current file.

current.device

Returns a string containing the name of the device (terminal) associated with the logged-on user.

Format

```
current.device()
```

Example

```
current.device() returns /dev/tty00 (on Unix)
current.device() returns LU0101A (on MVS)
```

Note: $lo.device is set to the value of current.device upon login.

current.format

Returns a string containing the name of the most recently displayed format.

Format

```
current.format()
```

Example

```
current.format() returns problem.open
```

current.screen

Returns an array of strings containing the screen image of the current screen, including all windows.

Format

```
current.screen()
```

current.scroll

Sets and returns a structure that describes the current scroll state of the screen, or sets the cursor into a field with a given name.

To maintain transparency, **current.scroll** can be used to set the cursor into a field with a given name. However, the **cursor.field.name.set** function should be used instead of **current.scroll** to set the cursor into a field with a given name.

Format

```
current.scroll(s)
```

Factors

- If the programmer wishes to remember the current cursor position of the screen and restore it later, use:

  ```
  $x=current.scroll()
  <other screen processing>
  current.scroll()=$x
  ```

- If the programmer wishes to set the cursor to the field named *header, number* use:

  ```
  current.scroll()=current.scroll("header,number")
  ```

current.window

Returns the name of the currently selected window (the **rio, fdisp, or wselect** command panel that is active—the one whose system tray is active) without regard to cursor position.

Format

```
$cwindow=current.window()
```

Examples

Assume two windows are displayed: **MainWindow** and **field.window** (as in the dbdict utility). The function tray displays functions of the **field.window**. The cursor is in **MainWindow**.

```
$cwindow=current.window()
```

After execution, value of **$cwindow** is **field.window**

Note: The current window is always **MainWindow** until another window is opened (using **wopen**) and/or selected (using **wselect**). The opened/selected window is then the current window until the current window is closed.

See also: See the **wopen** and **wclose** command panels in Chapter 4 for more information on how windows are used.

cursor.column

Returns the number of the column on the displayed format where the cursor was positioned when the last interrupt key (e.g., <**enter**>, **option** key) was pressed.

Format

```
cursor.column(n)
```

Example

Syntax	Returns
cursor.column()	The numeric value of the column designated by the cursor.
cursor.column(1)	The numeric value of the relative column of the cursor position in the array or structure within the format (considers horizontal scrolling as well). The value can be greater than the terminal's screen width.

cursor.field.contents

Returns a string with the contents of the field in the displayed format where the cursor was located when the last interrupt key (e.g., **enter**, **option** key) was pressed.

Format

```
cursor.field.contents()
```

Example

cursor.field.contents() returns *a101a01*

If the field is an array, the content of the current element is returned.

cursor.field.name

Returns a string containing the name of the input field (defined in ServiceCenter Format Manager) in the displayed format where the cursor was located when the last interrupt key (e.g., **enter**, **option** keys) was pressed.

Format

```
cursor.field.name(n)
```

Factors

If the function is called with an argument, the fully-qualified field name is returned.

cursor.field.name() returns *number*

cursor.field.name(1) returns *header,number*

Note: To position the cursor in a specific field, use the **cursor.field.name.set** function.

See also: **current.scroll** function.

cursor.field.name.set

Sets the cursor to a specific field on a form.

Format

```
cursor.field.name.set($field.name, $row.number)
```

Parameter	Description
$field.name	Name of the field.
$row.number	For arrays, the row number (optional)

Example

```
cursor.field.name.set("address", 2)
```

After execution, the cursor will appear in the second row of the **address** input field on the next form that is displayed.

Factors

- This function returns no value. It is not necessary to use it in an assignment statement.
- If the field is an array, you can position the cursor on a specific row by specifying **$row.number** .
- In GUI mode, if the cursor is set to a field that is off-screen, the cursor will be correctly positioned off-screen, i.e., the cursor will not be visible until the field is scrolled into view.
- In text mode, the cursor is never positioned off-screen. In text mode, if the cursor is set to a field that is off-screen, the cursor will be positioned in the first input field whose **ctrl** setting is **16.**

cursor.field.readonly

Returns *true* if the cursor is in a read-only field and *false* if the cursor is in a field that is not read-only.

Format

```
cursor.field.readonly()
```

cursor.filename

Returns the file name associated with the format in which the cursor is located.

Format

```
cursor.filename()
```

cursor.line

Returns the line number where the cursor was positioned when the last interrupt key (e.g., **enter**, **option** keys) was pressed.

Format

```
cursor.line(n)
```

Factors

- `$l=cursor.line()=` the numeric value of the line where the cursor was positioned.
- `$l=cursor.line(1)=` the numeric value of the index of the array or structure where the cursor was last located (taking vertical scrolling into consideration). If the cursor was in an array input field, *$l* contains the numeric value of the index of the array in which the cursor was positioned. If the cursor was not in an array input field, *$l* contains the numeric value, relative to the format in which the cursor was positioned (accounts for vertical scrolling, and may be greater than the screen length).
- `cursor.line(n)` is not supported in multi-line text boxes in GUI mode.

cursor.window

Returns the name of the window in which the cursor was last set.

Format

```
$cwindow=cursor.window()
```

Example

Assume two windows are displayed: **MainWindow** and **field.window** (as in the dbdict utility). The function tray displays functions of the **field.window**. The cursor is in **MainWindow**.

```
$cwindow=cursor.window()
```

After execution, value of **$cwindow** is **MainWindow.**

See also: **wopen** and **wclose** Command Panels and **current.window** function.

date

Returns the date portion of a date/time variable.

Format

```
date($date.time.variable)
```

Factors

- The *date* portion of a date/time value is midnight of that day (i.e., the time is **00:00:00**). Note that the date of a given date/time may be different in different time zones.
- The argument can be any absolute date/time value, an absolute time, including *tod*.

Example

```
$date.opened=date($date.time.variable)
```

Where $date.time.variable contains the current date and time or some other valid date/time value. If $date.time.variable = '08/01/90 10:10:10,' then $date.opened would equal '08/01/90 00:00:00'.

See also: **time**, **tod**, **dayofweek**, and **set.timezone** functions.

day

Returns the day of month for a date regardless of the date format.

Format

```
$day=day($dao1te)
```

Where $date is a date/time value.

Example

```
$mday=day('2/15/96')
```

After execution, value of $mday is 15

Note: The **set.timezone** function affects the way a date is presented; however, the **day** function *always* extracts the proper day value.

dayofweek

Returns the number of the day of the week for a specific date as follows:

1 = Monday	2 = Tuesday	3 = Wednesday	4 = Thursday
5 = Friday	6 = Saturday	7 = Sunday	

Format

```
dayofweek($date.time.variable)
```

Where **$date.time.variable** is the date whose day you want to know.

Factors

The **dayofweek** of a given date/time may be different in different time zones.

Example

```
dayofweek('1/4/90 00:00')—returns 4
```

See also: **set.timezone** function

dayofyear

Returns the day of year for a date regardless of the date format.

Format

```
$yday=dayofyear($date)
```

Where `$date` is a date/time value.

Example

```
$yday=dayofyear('2/15/96')
```

After execution, value of `$yday` is `46`

delete

Deletes one or more elements in an array and returns a new array without the deleted elements.

Format

```
delete(array, element number, [number of elements])
```

Where *array* is the array, *element number* is the index number of the first element to delete, and *number of elements* is the number of elements to delete. The *number of elements* is optional, and the default number of elements to delete is 1.

Example

delete({1,2,3},2) returns *{1,3}*

delete({1,2,3},2,2) returns *{1}*

See also: **insert**, **lng**, **denull** functions.

denull

Compresses an array by removing all trailing NULL entries and returns the compressed array.

Format

denull(***array***)

Factors

- If the array contains a NULL entry in an index position before the last non-NULL entry, that entry will **NOT** be removed.
- Using denull on an empty array of structures converts the array of structures to a simple array. (Always test for NULL first!)
- Displaying arrays extends the length of the array to accommodate window size. It is good practice to denull arrays before records are added/updated. Using denull (**contents**()) will accomplish this, but be sure there are no empty arrays of structures.

Example

denull({1,2,3,,})—returns {1,2,3}

denull({1,,2,,})—returns {1,,2}

See also: **delete** and **null** functions

descriptor

Returns the database dictionary descriptor structure for the specified file.

Format

descriptor(*$filename*)

Where *$filename* is the file variable for the file.

See also: structure discussion in Chapter 5.

evaluate

Evaluates an operator and may return a value.

Format

```
$e = evaluate($x)
```

Factors

- An operator is created using the parse() function.
- The evaluate function is also valuable when the statement that needs to be evaluated is contained in a ServiceCenter field which is not qualified when the application is compiled.
- The parse flag is used to parse data in a form. This converts the data to an operator (type 10).
- Although an **lvalue** is always required, a value is not always assigned to the **lvalue**.

Example

```
$x=parse("1+1",1) returns 2.
$x=evaluate($x)
```

See also: **parse** function

exists

Checks for the existence of a field in a file.

Format

```
exists(<field name>, $file)
```

Factors

- Returns true or false.
- Replaces `index(<field.name>, descriptor($file))>0` expression
- **$file** must be a file variable (type 6); any other type will return false
- The first parameter can be either a character type variable or a quoted string

Example

```
$L.return=exists("schedule.id", $L.schedule)
```

$L.return is **true** if $L.schedule is a file variable containing a schedule record

fduplicate

Copies an entire file variable from one record to another.

Format

fduplicate(*$target*, *$source*)

Factors

- The system will not recognize **fduplicate** alone.
- Returns a Boolean value: *true* if it is successful, and *false* if it fails.
- Used primarily in RAD **process** panels and in Format Control.

Example

```
$L.void=fduplicate($file0, $file)
```

Creates **$file0**, an independent copy of **$file**.

filename

Returns the name of the file for a specified file variable.

Format

filename(*$filename*)

Factors

- *$filename* is the variable that has been bound to some database file with the **rinit** Command Panel.
- This function is useful when executing a common subroutine that has been passed a file variable. Since the local variable contains data from an unknown file, this function allows you to determine the file name.

Example

filename($file)—returns problem.

See also: **rinit** command panel.

file.position

This function returns information about the current record and record list specified by a file variable.

Format

```
file.position(n)
```

The value of *n* determines what is returned:

n	Returns index into record list of
0	current record
1	last record on qbe list on screen
2	last record on last qbe list so far seen
3	last record seen in file
4	next record after last record seen in file
-1	first record on qbe list on screen
-2	first record on first qbe list so far seen
-3	first record seen in file
-4	record before first record seen in file.

Note: The default value of *n* is **0** (zero).

Example

```
file.position(0)=file.position()=4
```

Factors

This function is only used on the **fdisp** command panel on partial queries and only returns true or false based on the last record in the file.

filesize

Returns the size in bytes of a specified system file of your ServiceCenter implementation.

Format

`$size=filesize(`*`$file.number`*`)`

`$file.number0` returns the size of **scdb.fre**

Number	Returns
1	The size of scdb.asc
2	The size of scdb.lfd
3	The size of scdb.db1 (this is the default if $file.number is omitted)
4	The size of scdb.db2 (if present)
5	The size of scdb.db3 (if present)

Examples

```
$file.number=0
$size=filesize($file.number)
```

After execution, value of **$size** is 512

filesizes

Returns an array of numbers representing the size in bytes of each of the files that make up a ServiceCenter database.

Format

```
$file.size=filesizes()
```

Examples

```
$file.size=filesizes()
```

After execution, the value of `$file.size` is `{512, 1081344, 65536, 32000000}`.

Factors

- The numeric elements in the returned array correspond to the physical files in this order: **scdb.fre, scdb.asc, scdb.lfd, scdb.db1, scdb.db2, ..., scdb.dbn.**
- Use this function to determine programmatically how many data files ServiceCenter is using and their size.

fillchar

This function assigns/retrieves a field input character other than *underscore*.

Format

```
$f=fillchar()
```

Examples

```
$f=fillchar()
$f will contain a '_' (underscore) character
fillchar()=' '
```

All screens now displayed will not show the usual underlined input fields. Instead, input fields will be blank.

Factors

This function is only active in the text mode.

fixed.key

Returns and sets the name of the RAD application that a fixed option key will execute.

Format

```
fixed.key(20)="calc.window"
$a=fixed.key($key.number)
```

Factors

The function can be any compiled RAD application. This application must open a window if it wishes to communicate with the user.

See also: See the **wopen** and **wclose** command panels.

frestore

Restores all the fields in a file to their original (from the database) values.

Format

```
frestore($file)
```

Where $file is the file variable.

Example

```
frestore($file)
```

Assume `$file` is initialized and a record is selected. Changes are made to the contents of `$file` in memory, but the updates have not yet been written to the database.

After execution, `$file` is restored to its original value (its state as selected.)

Factors

- Use only with a full client against a P4 database.
- This function returns no value. It is not necessary to use it in an assignment statement.

genout

Generates a string containing the contents of a record using a specified form. It produces either fixed or variable length output.

Format

```
genout($file.variable, $format.name)
```

Parameters

Parameter	Definition
$file.variable	File variable containing record(s) to process.
$format.name	Name of format to be used.

Factors

- Spaces are substituted for NULL in fixed length output.
- All normal format processing, including input and output routines, occurs. Therefore, data can be automatically reformatted or manipulated as part of the function (e.g., converting a date from *02/28/96 00:00:00* to *February 28, 1996*).

Example

```
$output=genout($operator, "operator.view")
$output={"Name:joes Full Name:Joe Smith", "Printer:sysprint
Email:joes@peregrine.com"}
```

get.base.form.name

Returns the base name of a form, stripping off the form extension for GUI or Web forms. This will strip off either the .g or the .w from a form name.

Format

```
$base.form.name=get.base.form.name ($form.name)
```

Example 1

```
$form.name="operator.g"
$base.form.name=get.base.form.name($form.name)
```

After execution, `$base.form.name` will be `"operator"`.

Example 2

```
$form.name="operator.w"
$base.form.name=get.base.form.name($form.name)
```

After execution, `$base.form.name` will be `"operator"`.

Example 3

```
$form.name="operator"
$base.form.name=get.base.form.name($form.name)
```

After execution, `$base.form.name` will be "operator".

Example 4

```
$form.name="operator.a"
$base.form.name=get.base.form.name($form.name)
```

After execution, `$base.form.name` will be "operator.a".

get.dateformat

Returns the date format your operator ID is using.

Format

```
$date.fmt=get.dateformat()
```

Example

```
$date.fmt=get.dateformat()
```

After execution for a US user, value of `$date.fmt` is **1**

Factors

get.dateformat returns the following values:

Return Value	Format
1	mm/dd/yy
2	dd/mm/yy
3	yy/mm/dd
4	mm/dd/yyyy
5	dd/mm/yyyy
6	yyyy/mm/dd

get.timezoneoffset

Returns the absolute time difference between GMT and the time zone of the operator.

Format

```
$tzoffset=get.timezoneoffset()
```

Example

```
$tzoffset=get.timezoneoffset()
```

After execution for a Pacific Time time zone user, the value of `$tzoffset` is ‘`-08:00:00`’.

gui

Determines whether or not the process is running in GUI mode.

Format

```
$bool=gui()
```

Example

```
$bool=gui()
```

After execution in GUI mode, value of `$bool` is **true.** After execution in text mode, value of `$bool` is **false**

index

Returns the index or position number for a specific element value in an array or character in a string. If the target value is not in the array or string, it returns NULL.

Format

index(*target value*, *$variable*, *starting position #*)

Where **target value** is the value for which you are searching in the array or string, **$variable** is the name of the variable that will be searched, and **starting position** # is the index in the array or position in the string where the search will start. The default is **1**(one).

Factors

- The *index* function operates identically for arrays, regardless of the data type in the array.
- If the array is of structures or arrays, the target variable must exactly match the structure or array, i.e., the function does not allow selecting a field from a structure.
- Use *index* to search for any value by converting both the value and the search variable to the same type. For example, to search for the variable $problem.number on all forms beginning with "problem", the query passed to the select panel would be

Example

index(1,{1,2,3}) returns *1*

index(2,{1,2,3}) returns *2*

index(1,{1,2,3,},2) returns *0*

index(2,{1,2,3},2) returns *2*

index("$problem.number", str(field)) searches for the variable $problem.number on all forms beginning with "problem."

insert

Returns an array with one or more inserted elements.

Format

insert(*$array[, $position[, $number[, $value[, $denull*]]]])

Parameter	Description
$array	Array into which an element is inserted
$position	The position of the first inserted element. If this is zero, it will insert the element at the end of the array. This is an optional parameter and defaults to zero (0).
$number	Number of elements to insert. This is an optional parameter and defaults to 1. If the value is zero (0), this function will return without changing the array.
$value	A value to insert into the new elements. This is an optional parameter and defaults to NULL.
$denull	Logical value determining whether or not the array should be denulled before inserting any elements. This is an optional parameter and defaults to *true*.

Example 1

To insert an element at the beginning of an array:

```
$a={"a", "b", "c", "d"}
$a=insert($a, 1, 1, "z")
```

The value of $a is {"z", "a", "b", "c", "d"}

Example 2

To insert an element in the middle of an array:

```
$a={"a", "b", "c", "d"}
$a=insert($a, 3, 1, "z")
```

The value of $a is {"a", "b", "z", "c", "d"}

Example 3

To insert an element at the end of an array:

```
$a={"a", "b", "c", "d"}
 $a=insert($a, 0, 1, "z")
```

The value of $a is {"a", "b", "c", "d", "z"}

Important: If your source array has null elements at the end, this function will automatically denull your array before inserting any new elements (in any position). To avoid this set the $denull parameter to false.

```
$a={"a", "b", , , "e", "f", , , , }
 $a=insert( $a, 0, 1, "z")
```

The value of $a is {"a", "b", , , "e", "f", "z"}

```
$a={"a", "b", , , "e", "f", , , , }
 $a=insert( $a, 2, 1, "z")
```

The value of $a is {"a", "z", "b", , , "e", "f"}

To avoid denulling the array:

```
$a={"a", "b", , , "e", "f", , , , }
$a=insert( $a, 0, 1, "z", 0)
```

The value of $a is {"a", "b", , , "e", "f", , , , ,"z"}

See also: **delete** function

iscurrent

Determines if the record with which you are working is the most current version in the database.

Format

```
$boolean=iscurrent($file.variable)
```

lng

Returns the number of elements in an array or structure and the number of characters in a string.

Format

```
lng($variable)
```

Where **$variable** is the array, structure, or string.

Factors

- The **$variable** must be an array, structure or a character (string).
- Arrays return the number of elements (NULL or otherwise); strings return the number of characters; and structures return the number of fields.

Example

```
lng("1234567890") returns 10
lng({1,2,3}) returns 3
lng({1,2,3,,,,}) returns 7
lng(denull({1,2,3,,,,})) returns 3
```

Note: Remember to denull arrays to get the correct number of elements, but be careful of denulling arrays of structures.

locks

Returns an array of structures containing information about locks used to lock a database for an update or those set by RAD applications using a locked Command Panel.

Format

```
locks()
```

Factors

Each element of the array contains:

- *lock time*
- *process id*
- *terminal id*
- *operator name*
- *resource name (name of the lock)*
- *number*
- *exclusive*
- *locked*
- *breakable*

Example

```
locks() returns
{{['12/26/96 16:01:07', 20212, "SYSTEM", "marquee", "agent:marquee",
0, false, true, false]}
{[''01/02/97 11:26:08', 22699, "Windows 32", "falcon", "AG/pm.main", 0, true, true,
false]}}
```

logoff

Logs off the current user and terminates the session.

Format

```
logoff()
```

Factors

This function does not return any value; it simply executes a function.

Example

```
logoff()
```

logon

Logs the user on.

Format

```
logon()
```

Factors

This function does not return any value; it simply executes a function.

Example

```
logon()
```

mandant

Allows an application to create a SUBSET of the current Mandanten values. For example, if a new ticket is being entered, and the client (customer) for the ticket is known, then a SUBSET of the Mandanten values could be created to show only the assignments that are valid for that client.

Format

mandant(*n*,*string*)

Where *n* is either `0` (request for a SUBSET of the current mandant values) or `1` (request for original mandant values to be restored), and `string` is the Mandanten value.

Example

```
mandant(0,“MEGACORP”)
```

In this example, the Mandanten values become a SUBSET of the single value provided. This fails if the string value does not match the values currently established.

Example

```
mandant(1,””)
```

In this example, the application restores the original Mandanten values.

max

Returns the largest value in a list of values or arrays.

Format

max(*element1*, *element2*, *element3*, ...)

~or~

max(*$variable*)

Where *$variable* is an array or a list of values.

Factors

- The **max** function returns the maximum value of either times or numbers.
- If you include an array in the list of elements, the system evaluates all the elements in the array as if they were individual values.
- An attempt to use an array of strings as a parameter results in a segmentation fault, which will cause the application to take the **error** exit.

Example

max(17,24,35,73,10) returns 73

max(1,{2,3,4},3) returns 4

messages

Provides logging functions for use in the memory message log. The log is implemented as a wrap-around cache of the *x* most recent messages (error, informational and action) as displayed at the bottom of the screen in the message field.

Each user task (*scenter*) can have one private message log at a time. Message logs cannot directly be accessed by other tasks.

Format

messages(*$function_number*, *$function_parameter*)

Function #	Parameters	Action
0	0 to 500	Open message log with 0 to 500 entries.
1	none	Start or resume logging.
2	none	Stop logging.
3	none	Clear the log.

Function #	Parameters	Action
4	none	Close the log and recover memory.
5	0	Retrieve array of all log entries in most recently displayed order.
5	1	Retrieve array of logged informational messages; excludes error and action messages
5	2	Retrieve array of logged action messages; excludes error and informational messages.
5	3	Retrieves array of error messages; excludes informational and action messages

Factors

- A maximum of 500 logged messages is supported.
- Reading the message log (function 5) does not clear the log.
- Retrieving the log (# 5) always returns an array in last displayed order.
- Using the clear log function (function 3) stops logging, just as if a function 2 was executed.
- The proper order for using the logging functions is:
  - open log (function 0)
  - start logging (function 1)
  - ...run application
  - stop logging (function 2)
  - retrieve log (function 5)
  - clear log (function 3)
  - start logging (function 1)
  - ...run application
  - ...etc.
  - ...
  - ...
  - stop logging (function 2)
  - retrieve log (function 5)
  - close log (function 4)

Examples

Action	Syntax
Opening a log for 200 messages	$throwaway=messages(0,200)
Start logging	$throwaway=messages(1)
Stop logging	$throwaway=messages(2)
Clear log	$throwaway=messages(3)
Close log	$throwaway=messages(4)
Retrieve all logged messages	$array=messages(5,0)
Retrieve all logged informational messages	$array=messages(5,1)
Retrieve all logged action messages	$array=messages(5,2)
Retrieve all logged error messages	$array=messages(5,3)

min

Returns the smallest element in a list of values or arrays.

Format

min(*element1*, *element2*, *element3*, ...)

~or~

min(*$variable*)

Where **$variable** is an array or a list of values.

Factors

- The *min* function returns the maximum value of either times or numbers.
- If you include an array in the list of elements, the ServiceCenter evaluates all the elements in the array as if they were individual values.
- An attempt to use an array of strings as a parameter results in a segmentation fault, which will cause the application to take the **error** exit.

Example

min(17,24,35,73,10) returns *10*

min(4,{1,2,3,},2) returns *1*

modtime

Returns the last modified date/time of a record.

Format

modtime(*$file*)

Factors

- Use this function to determine if a record has been modified before or after a given time.
- Use modtime(*currec()*) to return the last modified date/time for any record. Use this variation in queries to retrieve all records modified after a given point in time.

Example

modtime($file) returns *1/10/96 12:15:17*

A query of modtime(currec)>tod( )- '08:00:00' selects all records in a file that were added or updated in the last eight hours.

Factors

The **modtime** function is not updated on clients when records are updated on the server until those records are reselected.

month

Returns the month of year for a date regardless of the date format.

Format

$ymonth=month(*$date*)

$dateDate/time value.

Example

$ymonth=month('2/15/96')

After execution, value of $ymonth is 2

null

Returns true if its parameter is NULL or is a compound data type which consists entirely of NULL values.

Note: Use of the **null** function should not be confused with the use of the reserved word NULL. The **null** function handles compound data types, and the reserved word NULL is for primitive data types only.

Format

null(*value*)

Factors

- An array or structure is null if it contains all null elements. This function applies itself recursively to nested elements.
- An empty string (" "), **0** (zero), and **00:00** are not null.
- A value of unknown in a boolean field is not null.
- Use the **null** function to check a field or variable to see if it is null.

Example

null(1) returns *false*

null(NULL) returns *true*

null({}) returns *true*

null({1,}) returns *false*

null({{},}) returns *true*

nullsub

Substitutes a null value with the second value given.

Format

nullsub(*value1*,*value2*)

Where *value2* is the value to return if *value1* is null.

Example

nullsub(1,2) returns *1*

nullsub(NULL,2) returns *2*

operator

Sets and returns the name of the current operator ID.

Format

```
operator()
```

Factors

- The operator function can also be used to set the operator name.
- The **operator**() value is set upon login to the login name in the operator record.

Example

```
$a = operator()
operator() = $a
```

option

Returns the number of the last interrupt key. For example, *0* = *enter*, *1* = *first option key*, etc.

Format

```
option()
```

Factors

- The **return** (or <**enter**>) key returns **0**.
- Use caution when `option()` is a criteria on a decision or process panel which can be accessed from several paths. You should set a variable to `option()` when the first panel is accessed.
- You should *not* attempt to account for Fkey remapping. The system will connect the remapped key to its native definition when the function is processed. For example, if a user had F1 remapped to F19, and the user selected F19, the `option()` function would return **1**, not **19**.
- Clicking on a button returns the number defined in the Button ID property.
- Clicking on an item in the Options menu returns the option number associated with the option on the **rio** command panel used to display the Options menu.

Example

option() returns 10

parse

Parses a string into an operator.

Format

parse(*string*,*type*)

Factors

- An operator can be either an expression (e.g., *type*<11) or a statement (e.g., *type*=11).
- The parse property, when enabled on forms, automatically parses the data entered.

Example

```
   $x=parse("1+1",1)
evaluate($x) returns 2

   $x=parse("$x=1",11)
evaluate($x) causes $x to be set to 1
```

See also: **evaluate** function

perf

Evaluates system performance and writes the information to disk. Returns either a 0 (success) or a -1 (error).

Format

perf(*n*)

Where *n* is the number of the option you want.

Option	Result
1	Delete records in the systemtotals and systemperform files where capture=false before writing new information to disk.
2	Add records and do not delete. Nothing is removed before new records are written to systemtotals and systemperform files. Maintenance of these files is the responsibility of the RAD programmer.

Option	Result
3	Delete records in the systemtotals and systemperform files where capture=false and do not write information to disk.
4	Delete all records in the systemtotals and systemperform files, regardless of capture status.

Example

perf(1) returns 0

See also: **prof** function

pfdesc

Returns a string containing the description of the specified option key from the terminal configuration data.

Format

pfdesc(*n*)

Where *n* is the number of the option you want.

Example

pfdesc(1) returns *F1=*

Factors

Pf key descriptions are defined in ServiceCenter's *termtype* record.

pfmap

Returns or sets the option key remapping array.

Format

pfmap()

Factors

The *nth* entry in the array is the number to which the nth option key has been remapped. This value must be from 1 to 24. There are 24 entries in this array.

Example

pfmap() returns *{1,2,3,4,5,6,13,14,9,15,16,12,7,8,10,11,17,18,19,20,21,22,23,24}*

printer

Sets and returns the value of the logged-on printer.

Format

```
printer()
```

Factors

This function can be assigned to set the current printer. When an operator logs on, the login application is executed, assigning `printer()` to be either the printer specified by the user at login or the default printer specified in the user's operator record.

Example

```
$a = printer()
printer() = $a
```

priority

Changes the priority of a task.

Format

```
priority(n)
```

Where *n* is the amount you wish to change the priority for a particular task.

Factors

- Priorities are always relative to the current task priority and may range from 1 to 255. A task starts out at a priority of 255 and may not be increased past that number. The priority may be decreased using the syntax: `priority (-10)` which then sets the priority to 245. To increase the priority back to 255, use the syntax: `priority (10)`. (Unix and MVS.)
- Priority has no functionality in Windows NT.

Example

```
priority(3)
```

processes

Returns an array describing all processes.

Format

```
processes(s)
```

Factors

Each element of the array contains:

- process start time
- process id
- terminal
- process name
- idle time

The argument of *s* determines the types of processes that will be returned:

S	Type of process
null	All
ALL	All
SYSTEM	System (terminal="SYSTEM")
USER	User (not System)
ACTIVE	Active
INACTIVE	Inactive (not active)

Example

processes("SYSTEM") returns
{{['4/20/92 10:25', 10936,"SYSTEM", "alert", '00:00:42']}}

prof

Returns various system performance statistics as it checks system resources used by an application.

Format

prof(*n*)

Where *n* is the value of the option you wish returned. Other parameters are as follows:

Parameter	Description
1	Returns user CPU time (unsupported on MVS)
2	Returns system CPU time (unsupported on MVS)
3	Returns memory allocated (unsupported on MVS)
4	Returns system total memory size.
5	Returns the number of statements evaluated.
6	Returns the number of low level read calls.
7	Returns the number of low level write calls.
8	Returns the number of bytes read.
9	Returns the number of bytes written.
10	Returns the number of screen I/O’s
11	Returns the number of calls to stcopy
12	Returns the user executing priority
13	Returns tbe total records retrieved
20	Returns the symbol table size
21	Returns the stack size

Example

prof(7) returns *10*

recordcopy

Copies a set of fields from one record to another record.

Syntax

```
$junk=recordcopy($source.file, $source.fields, $target.file, $target.fields)
```

Parameter	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$source.file	The source record from which fields are copied.
$source.fields	Array of field names in **$source.file**.
$target.file	The target record to which fields are copied.
$target.fields	Array of field names in **$target.file**.

Example

Assume **$source.file** is an operator record; **$target.file** is a contacts record.

```
name in $source.file="falcon"
phone in $source.file="858-481-5000"
$source.fields={"name", "phone"}
$target.fields={"contact.name", "contact.phone"}
$junk=recordcopy($source.file, $source.fields, $target.file, $target.fields)
```

After execution, the value of **contact.name** in$target.file is "falcon” and **contact.phone** in $target.file is "858-481-5000"

Warning: The recordcopy function does not type check data, so you can inadvertently copy a number field to a string, or a structure to a number.

recordtostring

The **recordtostring** function takes the value of a field from an array and appends the value to a string.

Format

```
recordtostring($string, $file, $arrayofnames, $sep)
```

Parameters

Parameter	Description
$string	String to which the field value is appended
$file	Current file
$arrayofnames	The array from which the field is taken
$sep	A separator character

Example

```
recordtostring("", $file, {"location", "location.name"}, "^")
```

Where $file is a location record in which **location=Acme HQ**, and **location.name=downtown**. After execution, $string will contain **Acme HQ^downtown.**

Factor

Event Services uses ^ as a default separator. You can use the **recordtostring** function to build an eventout string.

replicate

The replicate function replicates a named file from a remote site to the current site.

Format

replicate(*$filename, $sitename*)

Where $filename is the name of the file to replicate, and $sitename is the name of the remote site.

Example

```
$l.void=replicate($L.file, $L.site)
```

Factors

- Data added at the current site is deleted and replaced by a replicated copy of the data from the remote site.
- All changes to the file made at either site is applied to the other site.

round

Rounds up (positive direction) to the nearest specified number of decimal places.

Format

```
round(number,n)
```

Where *number* is the number to round and *n* is the number of decimal digits to be used in the rounding.

Note: This number can be anything that equates to a numeric value. For example, expressions, variables, numbers, etc.

Example

```
$a = round(3.47,1)
$a = 3.5
$a = round(3.53,1)
$a = 3.5
$a = round(3.45,1)
$a = 3.5
$a = round(-3.45,1)
$a = -3.4
$a = round(100/30,2)
$a = 3.33
$a = round('1/19/96 06:19:34',0)
$a = '1/19/96 06:20'
```

rtecall()

The ServiceCenter **rtecall**() functions are in-line statements that are especially useful in the ServiceCenter RAD Debugger. They are also available anywhere in ServiceCenter that supports expressions, statements, or calculations, such as in Format Control calculations, scripts, and displayoptions. The **rtecall**() function is extensible. New capabilities appear in every release of ServiceCenter. All options documented here are available in ServiceCenter release 5.0, but some may not be available in earlier releases.

rtecall("alalnum")

This function checks to make sure a string contains only alphanumeric characters. The *first* character must be alphabetic. The remaining characters must be alphabetic, numeric, or the provided non-alphabetic characters.

Format

```
$L.success.flag= rtecall ($L.fnc.name, $L.return.code, $L.str, $L.chars)
```

Parameters

Parameter	Date Types	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "alalnum"
$L.return.code	Number	Provides a more detailed return code
$L.str	String	The string to be checked
$L.chars	String	An optional comma delimited string of non-alphanumeric characters to allow

Factors

If the $L.9b

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flag=true
$L.return.code=0
$L.str=name in $L.file
$L.chars="., _"
$L.success.flag=rtecall("alalnum", $L.return.code, $L.str, $L.chars)
```

Result:

- When name in $L.file=”my.name” $L.success.flag will be true.
- When name in $L.file=”my_name” $L.success.flag will be true.
- When name in $L.file=”my name” $L.success.flag will be false.

rtecall("alnum")

This function checks to make sure a string contains only alphabetic, numeric, or the provided non-alphabetic characters.

Format

```
$L.success.flag= rtecall ($L.fnc.name, $L.return.code, $L.str, $L.chars)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "alnum"
$L.return.code	Number	Provides a more detailed return code
$L.str	String	The string to be checked
$L.chars	String	An optional comma delimited string of non-numeric characters to allow

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flag=true
$L.return.code=0
$L.str=phone in $L.file
$L.chars=" , (, ), -"
$L.success.flag=rtecall("alnum", $L.return.code, $L.str, $L.chars)
```

Result:

- When phone in $L.file="(508) 362.2701" $L.success.flag will be false.
- When phone in $L.file="(508) 362-2701" $L.success.flag will be true.

rtecall("alpha")

This function will check to make sure a string contains only alphabetic characters, or only alphabetic characters and the provided non-alphabetic characters.

Format

```
$L.success.flag= rtecall ($L.fnc.name, $L.return.code, $L.str, $L.chars)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful.
$L.fnc.name	String	Name of the sub-function to call, in this case "alpha."
$L.return.code	Number	Provides a more detailed return code.
$L.str	String	The string to be checked.
$L.chars	String	An optional comma delimited string of non-alphabetic characters to allow.

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flag=true
$L.return.code=0
$L.str=name in $L.file
$L.chars=" "
$L.success.flag=rtecall("alpha", $L.return.code, $L.str, $L.chars)
```

Result:

- When name in $L.file="my name" $L.success.flag will be true.
- When name in $L.file="my first name" $L.success.flag will be true.
- When name in $L.file="my 1st name" $L.success.flag will be false.

rtecall("counter")

This function turns counters on or off for the current session. Other SC users are unaffected.

Format

```
$L.success.flag= rtecall($L.fnc.name, $L.return.code, $L.switch)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "counter"
$L.return.code	Number	Provides a more detailed return code
$L.switch	Number	Either 1 (counters on) or 0 (counters off)

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg=rtecall("counter", $L.return.code, 1) to turn on counters
$L.success.flg=rtecall("counter", $L.return.code, 0) to turn off counters
```

rtecall("datemake")

This function returns a date, in the proper form, based upon a series of numbers passed to it.

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.date, $L.yr, $L.mo, $L.da,
$L.hr, $L.mn, $L.se)
```

Parameter

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case “datemake”
$L.return.code	Number	Provides a more detailed return code
$L.date	Date/Time	The variable in which the date will be returned
$L.yr	Number	The year (2 digit year uses prefix 20 for years up to 50, 19 for years after; e.g., 48 returns 2048, 99 returns 1999)
$L.mo	Number	The month
$L.da	Number	The day (1 through 31)
$L.hr	Number	The hour
$L.mn	Number	The minutes
$L.se	Number	The seconds

Factors

In this version, $L.success.flag and $L.return.code always return *true*, even when the function is unsuccessful.

Example 1

```
$L.cal.date=’01/01/00 00:00:00’
$L.success.flg=rtecall(“datemake”,$L.return.code, $L.cal.date, 0, 5, 31, 17, 15, 22)
```

Returns:

- $L.success.flag=true
- $L.cal.date=’05/31/2000 17:15:22’

Example 2

```
$L.cal.date=’01/01/00 00:00:00’
$L.success.flag= rtecall(“datemake”,$L.return.code, $L.cal.date, 0, 2, 31, 17, 15,
22)
```

Note that 2/31/00 is an invalid date.

Returns:

- `$L.success.flag=true`

- $L.cal.date='01/01/2000 00:00:00'

The result is invalid so the date variable is unchanged. Nonetheless, $L.success.flag returns true.

rtecall("escstr")

This function precedes special characters in a string with an escape character.

Format

```
rtecall($L.fnc.name, $L.rc, $L.str)
```

Parameters

Parameter	Description
$L.func.name	Name of the sub-function to call, in this case "escstr"
$L.rc	ReturnCode (standard ServiceCenter return code values).
$L.str	string to be modified.

Example

If `$L.str` contains the value `c:\dir\sub`

```
$L.rc=0
$L.ret=rtecall("escstr", $L.rc, $L.str)
```

Before the call to "escstr", $L.str contained the following string:

```
c:\dir\sub
```

After the rtecall, $L.str contains the following string:

```
c:\\dir\\sub.
```

The backslash escape character was inserted in front of the existing backslash.

Note: This function is rarely needed. The only time a string needs the escape characters added is when that string will be fed back into ServiceCenter. ServiceCenter treats any data between quotes as a string. If the data itself contains a quote then that quote must be escaped (with a backslash) so that the quote will be treated as data rather than the end of the string. Since the backslash is used as the escape character, any occurrence of a backslash in the data must also be escaped. An example of when this function might be needed is when a RAD program is constructing a query to retrieve a record based on data from some other record. In this case, the contact name in a problem ticket is used to retrieve the contact information from the contacts file. The RAD program might construct a query as follows:

```
$L.query = “name=”+contact.name in $file
```

This query will not work if the contact.name from $file contains a quote or a backslash because these characters will not be properly escaped and will cause the parse of the query to end prematurely. The correct code would be:

```
$L.temp = contact.name in $file
$L.ret = rtecall(“escstr”,$L.rc,$L.temp)
$L.query=”name=”+$L.temp
```

rtecall(“FILLDATE”)

This function places the current date and time in a field in the current record.

Format

```
$L.success.flg= rtecall($L.fnc.name, $L.return.code, $L.file, $L.field.name)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case “FILLDATE”
$L.return.code	Number	Provides a more detailed return code
$L.file	File	The file handle
$L.field	String	The name of the field to be updated

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

$L.file={[, , {}, , , {}, , , {}, , , {}, , , {}, {}, {}, {{[, , ,]}}, , ,]} where the first field is named “date”

$L.field.name=“date”

```
$L.success.flg= rtecall("FILLDATE", $L.return.code, $L.file, $L.field.name)
$L.file returns ={['04/14/2000 08:27:19', , {}, , , {}, , , {}, , , {}, , , {}, {}, {}, {{[, , , ]}}, , ,
]}
```

rtecall("filecopy")

This function copies all of the data in a collection to another file variable. The dbdict for both the source and target files must exist. Records are *added only*.

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.dbdict.source.name,
$L.dbdict.target.name, $L.count, $L.bad)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "filecopy"
$L.return.code	Number	Provides a more detailed return code
$L.dbdict.source.name	String	The name of the source dbdict
$L.dbdict.target.name	String	The name of the target dbdict
$L.count	Number	A count of the number of records successfully moved
$L.bad	Number	A count of the number of errors encountered

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.dbdict.source.name="location"
$L.dbdict.target.name="locationbak"
$L.success.flg=rtecall("filecopy", $L.return.code, $L.dbdict.source.name,
$L.dbdict.target.name, $L.count, $L.bad)
```

Returns the "locationbak" file as an exact copy of the "location" file.

rtecall(“fileinit”)

This function initializes a file variable in $targetfile from $sourcefile.

Format

```
rtecall($L.fnc.name, $errcode, $targetfile, $sourcefile)
```

Example

```
$L.void=rtecall(“fileinit”, $L.errcode, $file.old, $file)
```

Factors

- The file initialized is identified by the $sourcefile variable.
- The current record for $targetfile is the same as the current record in $sourcefile.
- Update and delete operations can not be done against $targetfile.

rtecall(“getnumber”)

Replaces the getnumb RAD application.

Format

```
$L.flg=rtecall($L.fnc.name, $L.return.code, $L.number, $L.class, $L.field)
```

Parameters

Parameter	Description
$L.flg	*True* for success, *false* if there was an error
$L.fnc.name	Name of the sub-function to call, in this case "getnumber"
$l.return.code	Zero (**0**) if everything is correct, **-1** if an error occurs (this is related directly to $L.flg).
$L.number	The number/string returned
$L.class	Class of number for which you are searching (from the `number` file).
$L.field	Reserved for future use

Factors

- Establishes a lock called "getnumb"+$L.class+$L.field and waits until the lock is established.
- If the number class was not found, or if there are duplicates, it will return an error ($L.flg=false)
- Reads the current number.
- Increments/decrements by the step value (step defaults to **1**)
- If incrementing and the number is greater than the reset value, it will use the start number (start defaults to **0**)
- If decrementing and the number is less than the reset value, it will use the start number (start defaults to **0**)
- New field called **string.flg**
- If **string.flg** is *false*, it will remain a numeric type
- If **string.flg** is *true*, it will convert the number to a string
- If **string.flg** is unknown, or the field isn't in the Database Dictionary (not added during an upgrade), the function checks for a prefix, a suffix or length. If any one of these exists, the function converts the value to a string. If none of these exist the function leaves the value as a number
- To convert the value to a string, the function uses the prefix, corrects the length of the number (padded with zeros on the left), and adds the suffix.

- The function eventually saves the new number to the file and returns it (converted to a string if necessary) to the user.
- Finally, it unlocks and returns.

rtecall("getrecord")

This function retrieves the record identified by unique key values in $L.array.

Format

```
$L.void=rtecall($L.fnc.name, $L.errcode, $L.array, $L.file)
```

Example

```
$L.void=rtecall("getrecord", $L.errcode, {"IM1001"}, $L.file)
```

Where `$L.file` has been initialized for the `probsummary` file.

Factors

$L.array is an INPUT variable, typically the key values returned by $L.array in rtecall("getunique").

rtecall("getunique")

Returns an array that contains the unique key values for a current record. These keys can be used later to retrieve the record using the **rtecall("getrecord")** function.

Format

```
$L.void=rtecall($L.fnc.name, $L.errcode, $L.array, $L.file)
```

Example

```
$L.worked = rtecall( "getunique", $L.errcode, $L.keyvalues, $L.file )
```

Variable/Value	Description
$L.worked = true	If the keys values are returned in $L.keyvalues
$L.worked = false	If no key values were returned. For example, if $L.file does not represent a current database record.
$L.errcode = 0	Function worked
$L.errcode = 1	$L.file does not represent a file variable

Variable/Value	Description
$L.errocde = 2	The file identified by $L.file does not have a unique key defined.
$L.file	File variable that has been initialized (RINIT) and contains a current record.

Factors

$L.keyvalues is an OUTPUT variable. It will be an array that contains the unique key values for the record in $L.file. For example if $L.file was for the problem file, an example of the key values returned in $L.keyvalues would be {"IM1001",1}.

rtecall("log")

This function sends a message to the external sc.log file.

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.message)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "log"
$L.return.code	Number	Provides a more detailed return code
$L.message	String	The message to be sent

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg=rtecall("log", $L.return.code, "This is a new message")
```

Generates a line in the sc.log file:

```
 137 04/12/2000 07:03:44 System background scheduler: event started at:
04/11/2000 23:02:48.
```

```
 137 04/12/2000 07:03:45 System background scheduler: availability started at:
04/11/2000 23:02:49.
 137 04/12/2000 07:03:45 System background scheduler: contract started at:
04/11/2000 23:02:50.
 137 04/12/2000 07:03:45 System background scheduler: ocm started at:
04/11/2000 23:02:51.
 137 04/12/2000 07:03:45 System startup completed successfully, elapsed time:
00:00:18.
 137 04/12/2000 07:32:48 24 records from location unloaded to: locations.save,
00:00:01 elapsed.
 306 04/12/2000 09:05:09 This is a new message
```

rtecall("notypecheck")

This function turns typechecking off or on. It is useful when, for example, numbers and strings of numbers are mixed in a field in error.

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.switch)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "notypecheck"
$L.return.code	Number	Provides a more detailed return code
$L.switch	Number	Either 0 (to disable) or 1 (to enable) type checking

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

Function	Result
$L.success.flg=rtecall("notypecheck", $L.return.code, 0)	Type checking is not performed
$L.success.flg=rtecall("notypecheck", $L.return.code, 1) –	Type checking is performed

rtecall("passchange")

This function changes this user's password.

Format

```
$L.success.flag=rtecall($L.fnc.name, $L.return.code, $L.old.pass, $L.new.pass,
$L.confirm.pass)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "passchange"
$L.return.code	Number	Provides a more detailed return code
$L.old.pass	String	The old password
$L.new.pass	String	The new password
$L.confirm.pass	String	A confirmation of the new password

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg= rtecall("passchange", $L.return.code, "oldpassword",
"newpassword", "newpassword")
```

rtecall("policycheck")

This function imposes data policy as defined in the datadict table. If the policy check fails, $L.success.flg is set to *false*.

Format

```
$L.success.flag= rtecall($L.fnc.name, $L.return.code, $L.file)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "policycheck"
$L.return.code	Number	Provides a more detailed return code
$L.file	File	The file handle

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.rc=rtecall("policycheck", $L.errcode, $L.filed)
```

rtecall("qbeform")

This function returns a QBE form, which can be passed into an **rio** or **fdisp** panel. You can insert this function into a format file handle using the **contents**() function (contents($L.format) = $L.qbe.form).

Format

```
$L.success.flg= rtecall($L.fnc.name,$L.return.code,$L.file,$L.qbe.form)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "qbeform"
$L.return.code	Number	Provides a more detailed return code; for this function, it will always be 0
$L.file	File	File handle used to generate the qbe e.g. the "rinit"ed contacts file
$L.qbe.form	Structure	The resulting QBE form, returned as a structure.

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg= rtecall("qbeform",$L.return.code,$L.file,$L.qbe.form)
```

rtecall("radhistory")

This function keeps track of what RAD panels a user has executed when running ServiceCenter applications.

Format

```
$L.flg=rtecall($L.fnc.name, $L.return.code, $L.history, $L.log.flg)
```

Parameters

Parameter	Data Type	Description
$L.fnc.name	String	Name of the sub-function to call, in this case "radhistory"
$L.rc	Number	Standard ServiceCenter return code (always equal to zero (0).
$L.history	Array	A structured array containing the RAD thread ID, application name, panel name, panel type, pre-formatted string containing all the above information
$L.log.flg	Logical	Quick return flag from all rtecall functions, indicating success or failure.

Factors

- The number of panels kept in history is configurable between 10 and 100 when the **radhistory** parameter is used in the sc.ini file or on a command line. The default is 20.
- $L.log.flg defaults to *false* if it is not included.

Example

```
$L.flg=rtecall("radhistory", $L.rc, $L.history, true)
```

This example prints the entire RAD history to the log.

rtecall("recdupl")

This function copies the contents of the current record into the contents of another record.

Format

```
rtecall($L.fnc.name, ercode, $targetfile, $sourcefile)
```

Factors

- Both $targetfile and $sourcefile *must* have identical descriptors. Data is copied by *position* and not field name.
- Both $targetfile and $sourcefile must be initialized file variables.
- Update and delete operations cannot be performed against $targetfile.

Example

```
$L.flg=rtecall("recdupl", $L.return.code, $L.temp, $L.file)
```

rtecall("rfirst")

This function places the pointer at the first record in a record collection (a QBE list).

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.file)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "rfirst"
$L.return.code	Number	Provides a more detailed return code
$L.file	File	File handle that represents the collection

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg=rtecall("rfirst",$L.return.code, $L.file)
```

rtecall("rgoto")

This function places the pointer at the indicated record.id in a record collection (a QBE list).

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.file, $L.record.id)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case "rgoto"
$L.return.code	Number	Provides a more detailed return code
$L.file	File	File handle that represents the collection
$L.record.id	Number	A record number

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg=rtecall("rgoto",$L.return.code, $L.file, $L.record.id)
```

rtecall("rid")

This function returns the record number of the current record (represented by $L.file).

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.file, $L.record.id)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case “rid”
$L.return.code	Number	Provides a more detailed return code
$L.file	File	File handle that represents the record
$L.record.id	Number	The record number

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg=rtecall(“rid”,$L.return.code, $L.file, $L.record.id)
```

rtecall(“sort”)

This sorts a list or a list of lists in ascending or descending order.

Format

```
$L.success.flg=rtecall($L.fnc.name, $L.return.code, $L.grid, $L.index, 0)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case “sort”
$L.return.code	Number	Provides a more detailed return code
$L.grid	Array	The list (or list of lists) to be sorted
$L.index	Number	The index of the array in the grid that should be used as the sort field **Note:** Note: 0 is the first element of the array.
$L.type	Number	Ascending (1) or descending (0)

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.list={{“a”, ”b” ,”d”, ”c”},{1, 3, 4, 2}}
$L.success.flg=rtecall("sort", $L.return.code, $list, 1, 0)
```
Returns: $L.list= {{"a", "c", "b", "d"}, {1, 2, 3, 4}}

```
$L.success.flg=rtecall("sort", $L.return.code, $list, 1, 1)
```
Returns: $L.list={{"a", "c", "b", "d"}, {1, 2, 3, 4}}

```
$L.success.flg=rtecall("sort", $L.return.code, $list, 0, 0)
```
Returns: $L.list={{“a”, “b”, “c”, “d”}, {1, 3, 2, 4}}

```
$L.success.flg=rtecall("sort", $L.return.code, $list, 0, 1)
```
Returns: $L.list={{"d", "c", "b", "a"}, {4, 2, 3, 1}}

rtecall(“transtart”)

This function measures the amount of data transferred, elapsed time and CPU usage of any *transaction*. It is commonly invoked from the GUI debugger and is used in conjunction with *transtop*.

Format

```
$L.success.flag= rtecall($L.fnc.name, $L.return.code, $L.transaction)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case “transtart”
$L.return.code	Number	Provides a more detailed return code
$L.transaction	String	Any user-selected name for the transaction to be measured

Factors

- If the $L.success.flg is *false*, the function failed.
- If $L.success.flg is *true*, the function succeeded.

Example

```
$L.success.flg= rtecall("transtart", $L.return, "problemopen")
```

rtecall(“transtop”)

This function measures the amount of data transferred, elapsed time and CPU usage of any *transaction*. It is commonly invoked from the GUI debugger and is used in conjunction with *transtart.*

- This is supported on both full and express clients.
- The information is from the perspective of where the RAD is running. Therefore, in express mode the CPU time and data transmission number are from the server machine, while in a full client the CPU time and data transmission numbers are from the client machine.
- String copies are a rough indication of the amount of RAD processing.
- You can have multiple transaction timings active at the same time.

Format

```
$L.success.flag= rtecall($L.fnc.name, $L.return.code, $L.transaction, $results)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case “transtop”
$L.return.code	Number	Provides a more detailed return code
$L.transaction	String	Any user-selected name for the transaction to be measured
$results	Array	The results gathered, in the form of an array: {elapsed seconds, cpu seconds, #string copies, bytes sent to client, bytes received from client}

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

```
$L.success.flg=rtecall("transtop", $L.return.code, “problemopen”, $results)
$results={58.164, 0.961, 46840, 26712, 2477}
```

This shows that the “problemopen” transaction took 58.164 seconds of wall clock time and 0.961 CPU seconds, and performed 46,840 string copies. The transaction sent 26,712 bytes to the client and the client returned 2,477 bytes to the server.

rtecall(“trigger”)

This function turns triggers on or off for the current session. Other ServiceCenter users are unaffected.

Format

```
$L.success.flag= rtecall($L.fnc.name, $L.return.code, $L.switch)
```

Parameters

Parameter	Data Type	Description
$L.success.flg	Logical	Indicates if the function was successful
$L.fnc.name	String	Name of the sub-function to call, in this case “trigger”
$L.return.code	Number	Provides a more detailed return code
$L.switch	Number	Either 1 or 0, to turn triggers on or off respectively

Factors

If the $L.success.flg is *false*, the function failed. If it is *true*, the function succeeded.

Example

$L.success.flg=rtecall("trigger", $L.return.code, 1) to turn triggers on

$L.success.flg=rtecall("trigger", $L.return.code, 0) to turn triggers off

same

Compares two values; if the values are found to be equal or both null, the result is TRUE, if not, the result is FALSE.

Format

same(*value1*,*value2*)

Factors

- This function should be used whenever comparing arrays and structures, nulls or empty strings.
- When using same to compare arrays or structures, preface the variable names with denull. e.g., not same(denull($filea), denull($fileb))

Example

same(1,NULL) returns *false*

same(NULL,NULL) returns *true*

same({},{,}) returns *true*

same({1},{1,}) returns *true*

same("",NULL) returns *false*

scmsg

Returns a message of a particular type and number from the **scmessage** file and substitutes text for the variables in that message.

Format

scmsg(*$class, $id, $arrayof values*)

Parameters

Parameter	Description
$class	Message class (from the **scmessage** file)
$id	ID number of the message (message.id) from the **scmessage** file.
$arrayofvalues	An array of substitution text for the %S variables in the message text (from the **scmessage** file)

Example

```
scmsg(“cib”, “21”, {“mail”, “middle,caller.id”, “open”})
```

The text of this message from the **scmessage** file is the following: *Built macro to %S to %S on %S.* When the %S array values from the **scmsg** function in this example are applied, the message is displayed as: *Built macro to mail to middle,caller.id on open*.

Factors

- All the substitution text must be %S, which indicates that a STRING is being used to provide the data. Each entry in the array of substitution text is examined for its type, and if it is not a STRING type, it is converted to a STRING.
- You must create a separate record in the **scmessage** file for the message in each language you want to display.
- The $arrayofvalues defined in the **scmsg** function are applied to the message in the **scmessage** file that corresponds to the language of the client login.

set.timezone

Sets the time zone, the translation from local to internal characters, and the date format. Timezone is established at either user login or process startup time. (Not all started processes are User Processes) At login, the timezone is determined using first the Company record and then the Operator record.

Format

set.timezone(*$tzfile,1*)

Factors

$tzfile represents a record in the ServiceCenter file.

See also: System Administrator's Guide for more information on time zones and date formats.

setsort

The **setsort** function sorts the fields of an array in a file.

Format

setsort($file, $arrayof names, 0): ascending sort

setsort($file, $arrayof names, 1): descending sort

Example

```
$L.void=setsort($L.ocml, $L.sort, 0)
```

share

The share function shares the data in the current record with the same file in several remote sites.

Format

share($filename, $arrayof sites)

Where $filename is the name of file whose data is to be shared, and $arrayofsites are the names of the remote sites to receive the data.

Example

```
if (not null($L.sites)) then ($L.void=share($L.file, $L.sites))
```

shutdown

Shuts down ServiceCenter from inside the system and does not return any values; it simply performs an operation.

Format

```
shutdown()
```

Factors

This function should be placed on the last panel that is to be executed in a shutdown application. Once the function is executed, the operator will see the operating system prompt from which ServiceCenter was started.

str

Converts a non-string data type into a string.

Format

str(*any valid expression*)

Where *any valid expression* is the value to be converted to a string.

Examples

str(1+1) returns “2“

"report run at "+ str(tod()) returns “*report run at 12/10/90 10:10:00*“

See also: **val** function

stradj

Makes a string a specific length by clipping or adding trailing blanks.

Format

$junk=stradj(*$string*, *$size*)

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$string	String being modified.
$size	Desired size of $string.

Example

```
$string="Peregrine Systems"
$size=20
$junk=stradj($string, $size)
```

After execution, value of `$string` is `"Peregrine Systems   "`

```
$string="Peregrine Systems"
$size=15
$junk=stradj($string, $size)
```

After execution, value of `$string` is `"Peregrine Syste"`

strchrcp

Replaces part of a string with copies of another string.

Format

```
$junk=strchrcp($target, $index, $source, $copies)
```

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$target	String being modified.
$index	Position in `$target` to begin copying `$source`.
$source	String to be copied into `$target`.
$copies	Number of copies of `$source` to replicate within `$target`.

Example

```
$target="Peregrine Systems"
$index=4
$source="falcon"
$copies=2
$junk=strchrcp($target, $index, $source, $copies)
```

After execution, value of `$target` is `"Perfalconfalconms"`

Factors

This function never extends the length of `$target`.

strchrin

Inserts copies of a string into another string.

Format

```
$junk=strchrin($target, $index, $source, $copies)
```

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$target	The string being modified.
$index	The position in $target to begin inserting copies of $source.
$source	The string to be inserted into $target.
$copies	The number of copies of $source to insert into $target.

Example

```
$target="Peregrine Systems"
$index=4
$source="falcon"
$copies=2
$junk=strchrcp($target, $index, $source, $copies)
```

After execution, value of `$target` is `"Perfalconfalconegrine Systems"`

strclpl

Clips a number of characters from the beginning of a string.

Format

```
$junk=strclpl($string, $number)
```

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$string	The string being modified.
$number	The number of characters to clip.

Example

```
$string="Peregrine Systems"
$number=6
$junk=strclpl($string, $number)
```

After execution, value of `$string` is `"ine Systems"`

strclpr

Clips a number of characters from the end of a string.

Format

$junk=strclpr(*$string*, *$number*)

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$string	The string being modified.
$number	The number of characters to clip.

Example

```
$string="Peregrine Systems"
$index=6
$junk=strclpr($string, $number)
```

After execution, value of `$string` is `"Peregrine S"`

strcpy

Replaces part of a string with a substring.

Format

$junk=strcpy(*$target*, *$tindex*, *$source*, *$sindex*, *$number*)

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$target	The string being modified.
$tindex	The position in `$target` to begin copying a substring from `$source`.

Variable	Description
$source	The string supplying a substring.
$sindex	The beginning position of the substring within $source.
$number	The number of characters to copy from $source.

Example

```
$target="Peregrine Systems"
$tindex=4
$source="falcon"
$sindex=2
$number=3
$junk=strcpy($target, $tindex, $source, $sindex, $number
```

After execution, value of `$target` is `"Peralcine Systems"`

strdel

Deletes a number of characters from a specific location in a string.

Format

`$junk=strdel(`*`$string`*`, `*`$index`*`, `*`$number`*`)`

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$string	The string being modified.
$index	The location in $string from which characters are to be deleted.
$number	The number of characters to delete.

Example

```
$string="Peregrine Systems"
$index=6
$number=8
$junk=strdel($string, $index, $number)
```

After execution, value of `$string` is `"Peregtems"`

strins

Inserts a substring into another string.

Format

$junk=strins(*$target, $tindex, $source, $sindex, $number*)

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$target	The string being modified.
$tindex	The position in $target to begin inserting a substring from $source.
$source	The string supplying a substring.
$sindex	The beginning position of the substring within $source.
$number	The number of characters to insert from $source.

Example

```
$target="Peregrine Systems"
$tindex=4
$source="falcon"
$sindex=2
$number=3
$junk=strins($target, $tindex, $source, $sindex, $number)
```

After execution, value of $target is "Peralcegrine Systems"

strpadl

Pads a string with leading blanks until the string is a certain size.

Format

$junk=strpadl(*$string, $size*)

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).

Variable	Description
$string	The string being modified.
$size	The desired size of `$string`.

Example

```
$string="Peregrine Systems"
$size=20
$junk=strpadl($string, $size)
```

After execution, value of `$string` is `"   Peregrine Systems"`.

strpadr

Pads a string with trailing blanks until the string is a certain size. If the string is longer than the desired size, it is truncated.

Format

```
$junk=strpadr($string, $size)
```

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$string	The string being modified.
$size	The desired size of `$string`.

Example

```
$string="Peregrine Systems"
$size=20
$junk=strpadr($string, $size)A
```

After execution, value of `$string` is `"Peregrine Systems   "`

```
$string="Peregrine Systems"
$size=10
$junk=strpadr($string, $size)
```

After execution, value of `$string` is `"Peregrine "`

strrep

This function returns a string, replacing a specified character sequence with another string.

Format

```
strrep($target,$string1,$string2)
```

Variable	Description
$target	The original string containing characters that will be replaced.
$string1	The string within $target that needs to be replaced.
$string2	The string that is replacing $string1.

Example

```
$s=strrep("Peregrine","egrine","formance")
```

Value of `$s` is `"Performance"`

strtrml

Removes all leading blanks from a string.

Format

```
$junk=strtrml($string)
```

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$string	The string being modified.

Example

```
$string="  Peregrine Systems"
$junk=strtrml($string)
```

After execution, value of `$string` is `"Peregrine Systems"`

strtrmr

Removes all trailing blanks from a string.

Format

```
$junk=strtrmr($string)
```

Variable	Description
$junk=	Syntax requirement only (has no bearing on the outcome of the function).
$string	The string being modified.

Example

```
$string="Peregrine Systems  "
$junk=strtrmr($string)
```

After execution, value of `$string` is `"Peregrine Systems"`

substr

Extracts a substring from a string.

Format

substr(*string*, *beginning position*, *length*)

Where *string* is the variable name of the string from which the sub-string is to be extracted, *beginning position* is the position in the string where the substring is to begin, and *length* is the number of characters in the substring (default is 1). You must provide a starting position (no default), but if the length is omitted, the substring selects from the starting position to the end of the string.

Example

$a=substr("IBM Corporation", 1, 3) returns *IBM.*

Factors

You must use a space after each comma for the substring expression to work properly.

sysinfo.get

Returns various information about the session the user is running.

Format

$info*=sysinfo.get(*$parm)

Parameters ($parm)

Parameter	Returns
“ActiveFloatUsers”	Returns the total number of floating users currently active
“ActiveLicenses”	Returns the total number of licenses currently active
“ActiveNamedUsers”	Returns the total number of named users currently active
"ClientNetAddress"	Returns a character string representation of the ServiceCenter client network address (IP address or APPC LU name) Example: $x.clientNetAddress=sysinfo.get("ClientNetAddress")
ClientOSName	Returns the operating system ServiceCenter is running on. For example Windows 2000, Windows NT, Mac OS.
"ClientVersion"	Returns the version number of the client software as a string in the form of *release.version.level*. For example: *4.0.5*
"ClientPID"	Returns the PID for RAD. For express, *scenter* PID will be the same as the server PID. Example: $x.clientPid=sysinfo.get("ClientPID")
"Display"	■ Returns gui if the user is running a GUI client. ■ Returns text if the user is running a TEXT client. Examples: 1 $display=sysinfo.get("display") After execution, $type will be "gui" if the user is running a Windows session. 2 If sysinfo.get("display")="gui" then (…) 3 If index(sysinfo.get("display"), {"gui","web"})>0 then (…) 4 If sysinfo.get("display")~="text" then (…)
"Environment"	Environment option that returns: ■ *scclient* if RAD is running on a text client ■ *scserver* if RAD is running on the server ■ *scenter* if RAD is running on a scenter process ■ *scguiw32* if RAD is running on an express or full GUI client
“MaxFloatUsers”	Returns the total number of floating users logged on since startup

Parameter	Returns
“MaxLicenses”	Returns the maximum number of licenses used since startup
“Mode”	■ Returns *server* if RAD is running in server mode. ■ Returns *scenter* if RAD is running on a scenter process. ■ Returns *client* if the RAD is running on the client (full client mode) ■ Returns *express* if the RAD is running in **express** mode. *Examples:* 1 $mode=sysinfo.get("mode") If the user is running from a Windows client, **$mode** will be "**client**." 2 If sysinfo.get("mode")="client" then (…) 3 If (index(sysinfo.get("mode"),{"scenter","server", "express"})>0 then (…)
“PrevLabel”	Used for error determination. A RAD program might require the label of the last RAD panel that was executed. The sysinfo.get(“prevlabel”) RTE call returns that label name. Example: $L.lastlabel = sysinfo.get(“prevlabel”) The panel name is the last panel that was executed in the called RAD program. At the start panel for any application, the name is the calling panel name from the parent application
"PrintOption"	Returns printing option information—*server* or *client* depending on how the system is set.
“Quiesce”	Returns *true* if system is quiesced and *false* if quiesce is off
"RecList"	Returns *true* if record list is enabled and *false* if record list is disabled
"ServerNetAddress"	Returns a character string representation of the ServiceCenter client network address (IP address)
"ServerPID"	Returns the numeric process ID (PID) of the server.
“SystemEvents”	Returns System Events information:. For example:

Event Name	Application Name	Returns
Take Call	dde.take.call	“RAD”
Show List	mycfunction	“c”

Parameter	Returns
"Telephony"	Returns *true* if telephony is enabled and *false* if telephony is disabled.
"ThreadID"	Returns the number of the current RAD thread. Example: $x = sysinfo.get ("ThreadID") `$x` will be set to the number ID of the current thread.
“TotalFloatUsers”	Returns the total number of floating users allowed
“TotalLicenses”	Returns the total number licenses allowed
“TotalNamedUsers”	Returns the total number of named users allowed
"TotalProcs"	Returns the total number of executing tasks
“TotalSystemProcs”	Returns the total number of system tasks
“TotalUserProcs”	Returns the total user tasks

time

Returns the time portion of a date/time value.

Format

```
time($date.time.variable)
```

Where *$date.time.variable* is the date/time variable name.

Factors

Note that the time of a given date/time may be different in different time zones.

Example

time($date.time) returns *12:10:15*

See also: set.timezone function

tod

Returns the current system date and time.

Format

```
tod()
```

Factors

The **tod** function returns *mm/dd/yy hh:mm:ss*.

Examples

```
tod()='4/20/96 12:15:16'
$curr.time=tod()
```

tolower

This function returns a string, converting all upper-case characters in the string variable to lower-case. Characters with no lower-case equivalent remain unchanged.

Programming Considerations: Character conversions are based on the value of the language parameter.

Format

tolower(*$string*)

Where $string is the string variable containing upper-case characters.

Example

```
$s=tolower("Copyright 1995")
```

The value of `$s` is `"copyright 1995"`

toupper

This function returns a string, converting all lower-case characters in the string variable to upper-case. Characters with no upper-case equivalent remain unchanged.

Programming Considerations: Character conversions are based on the value of the language parameter.

Format

```
toupper($string)
```

Where $string is the string variable containing lower-case characters.

Example

```
$s=toupper("Copyright 1995")
```

The value of $s is "COPYRIGHT 1995"

translate

Translates from one character set to another, such as from lower to upper case, encoding data, etc.

Format

translate(*source*, *old character set*, *new character set*)

Where *source* is the string to translate, *old character set* is the character set that already exists, and *new character set* is the character set that has been translated.

Factors

- Both old and new character sets must be strings of the same length.
- Both old and new character sets may be literals, variables or fields.
- A one for one correlation must exist between the old and new character sets (i.e., the third character in the new set replaces the third character in the old set).

Example

translate(“123abcdef”,“abc”,“ABC”) returns *123ABCdef.*

See also: **toupper. tolower** functions.

trunc

Truncates the decimal digits of a number to a specified number of digits.

Format

```
trunc(number,n)
```

Where *number* is the number to truncate and *n* is the number of decimal digits desired. The default value of *n* is **0** (zero).

Factors

- The second parameter is not required. By default, the number passed is truncated to a whole number.
- Negative and positive numbers are treated identically.

Examples

Function	Returns
trunc(3.45,1)	3.4
trunc(3.44)	3
trunc(‘13:19:34‘)	‘13:19:00‘

See also: **round** function

type

Returns the numeric type of a datum.

Format

```
$ttype=type($any.value)
```

Where $any.value is any literal or variable.

Example

```
$ttype=type('2/15/96')
```

After execution, value of $ttype is 3.

Factors

Type returns the following numeric values:

Data Type	Return Value
Number	1
Character String	2
Date/Time	3
Logical	4
Label (of a RAD application)	5
File	6
Array	8
Structure	9
Operator	10
Expression	11

val

Converts a datum to a different data type.

Format

`$new.data=val(`*`$data, $new.type`*`)`

Variable	Description
$data	Datum to be converted.
$new.type	Datatype to which the data will be converted: 1 = number (default) 2 = string 3 = date/time 4 = logical

Examples

Variable	Value of $x after execution
$x=val(100)	100
$x=val(100, 2)	"100"
$x=val(100, 3)	’00:01:40’
$x=val(0, 4)	false
$x=val(1, 4)	true
$x=val(2, 4)	unknown
$x=val(3, 4)	NULL
$x=val("Peregrine", 1)	NULL
$x=val("100.3", 1)	100.3
$x=val("Peregrine", 2)	"Peregrine"
$x=val("Peregrine", 3)	NULL
$x=val("2/15/96", 3)	’02/15/96 00:00:00’
$x=val("true", 4)	true
$x=val("0", 4)	NULL
$x=val(’01:00:23’, 1)	3623
$x=val(’02/16/96 08:00:00’)	62992915200
$x=val(’01/01/96’, 2)	"01/01/96 00:00:00"
$x=val(’02/09/96 08:00:00’, 3)	’02/09/96 08:00:00’
$x=val(’01/01/96’, 4)	NULL
$x=val(’12:34:56’, 4)	NULL
$x=val(false)	0
$x=val(true, 1)	1
$x=val(unknown, 1)	2
$x=val(false, 2)	"false"
$x=val(true, 3)	NULL
$x=val(unknown, 4)	unknown

Factors

The following type conversions are supported:

- *Number* can be converted to date/time, string, or logical.
- *String* can be converted to number, date/time, or logical.
- *Date/time* can be converted to number or string.
- *Logical* can be converted to number or string.

variable.send

Sets the value of a RAD variable on the server when using a full client. In a non-client mode this function does not perform any operation.

Format

```
variable.send()
```

Example

```
$L.name = "Bob Jones"
variable.send("$L.name")
```

This example sets the value of the variable $L.name to "Bob Jones" on the server. If the variable does not exist on the server it will be created. Note the importance of the double quotes around the variable name. Without the double quotes the function attempts to set the variable "Bob Jones" which is invalid.

You can use a variable to represent the variable name as follows:

```
$L.name = "Bob Jones"
$L.varname = "$L.name"
variable.send($L.varname)
```

This variation is identical in execution to the first example.

Factors

Use this function to set variables that may be required for database triggers.

version

Returns an array containing the release number of the ServiceCenter executable.

Format

```
version()
```

Example

```
$a=version()
```

Where the value of `$a` is `{"unix",7.0,"4.0.3"}`

- The first element is a string naming the platform:
  - unix
  - mvs
  - mswin
  - winnt
- The second element is 7.0, the RAD release level.
- The third element is a string containing the ServiceCenter version number and the Service Pack number (e.g., `4.0.3` indicates ServiceCenter version 4.0 and Service Pack 3).

year

Returns the full year for a date regardless of the date format.

Format

```
$fyear=year($date)
```

Where `$date` is the date/time value.

Example

```
$fyear=year('2/15/96')
```

After execution, value of `$fyear` is **1996.**

Pseudo Fields

A *pseudo-field* converts input data into output data and can sometimes be assigned. That is, it can be placed on the left hand side of an equals (=) sign. ServiceCenter has pre-defined pseudo-fields.

Note: Pseudo fields are not supported in non-P4, SQL tables.

Month

Accesses the number of months from the beginning of the time base in a date. This function can be used to increment a date by a period of months.

Example

```
Month in $time +=1
```

This increments the current month by one.

Factors

The month function is most useful for scheduling monthly reports, since the number of days to add to each month is calculated automatically.

Name

Accesses the file name in a specific file variable.

Example

```
$filename=Name in $file
```

This sets *$filename* to the name of the file bound to `$file`.

Factors

The name function is analogous to the **filename** function.

Debugging RAD Flows

Two methods for debugging a RAD flow exist within ServiceCenter:

- RAD Debugger
- Command line parameters

RAD Debugger

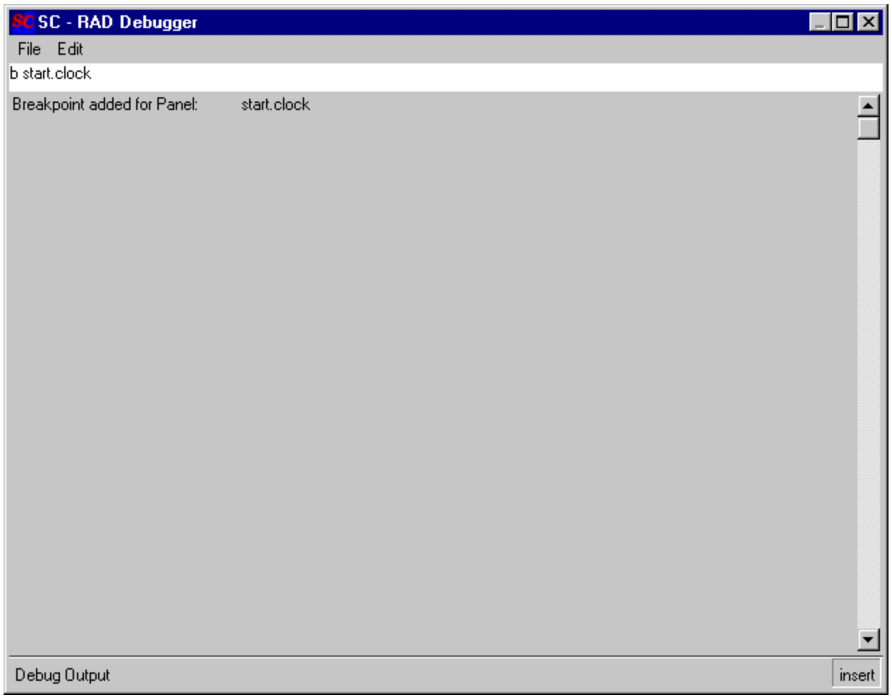
The RAD Debugger allows ServiceCenter users to view and isolate RAD processes for the purpose of troubleshooting an application. Information about the RAD flow is displayed in a separate window containing a scrollable text field.

To use the RAD Debugger:

1 Press `Ctrl-Shift+D+E`.

The RAD Debugger command window is displayed.

2 Enter the desired command followed by a parameter.

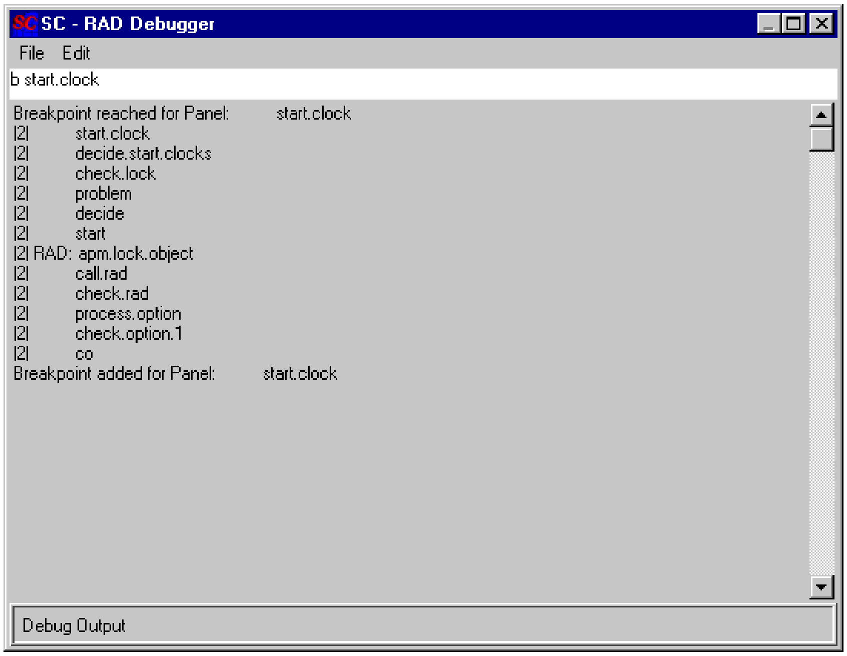


```
b start.clock
Breakpoint added for Panel:        start.clock
```

Figure 13-1: RAD Debugger

3 Press **Enter**.

The requested information is displayed.



```
b start.clock
```

```
Breakpoint reached for Panel:       start.clock
|2|        start.clock
|2|        decide.start.clocks
|2|        check.lock
|2|        problem
|2|        decide
|2|        start
|2| RAD:  apm.lock.object
|2|        call.rad
|2|        check.rad
|2|        process.option
|2|        check.option.1
|2|        co
Breakpoint added for Panel:       start.clock
```

Figure 13-2: RAD Debugger—adding a breakpoint for a named panel

Commands

Command	Function
d (display)	Displays the contents of a variable. A common use of this command is to display the name of the Display application Screen ID attached to the current form. d $L.screen
t (trace)	Turns tracing on or off. Tracing allows you to see every panel that the RAD flow encounters. Use the command by itself to display the status of the trace function. t on t off t

Command	Function
ta	Turns panel tracing on or off for a specific RAD application. To turn application tracing on, include the name of a RAD application. To turn application tracing off, repeat the same command combination a second time. You may have multiple traces engaged at one time. Use the command by itself to display a list of the RAD applications being traced. ta script.execute ta
tar	Removes all RAD specific panel traces. If you are tracing multiple applications, you can save time by using tar rather than using ta to turn off each trace individually.
tt	Turns panel tracing on or off for RAD applications called by a trigger only. Trigger tracing only works with an express client. Use the command by itself to display the status of the panel trace function. tt on tt off tt
b (breakpoint)	Sets a panel breakpoint. When the RAD flow encounters a panel with this label, it halts before executing the named panel and gives you an opportunity to perform debugging procedures such as displaying variables or executing statements. Repeat the command combination a second time to turn off the breakpoint. Execute the command by itself to display a list of all current breakpoints. b init.operator b
ba	Sets a breakpoint on a specific RAD application. A breakpoint is applied each time the RAD flow enters the named application, whether it has run for the first time or when the flow returns to it from a subroutine. ba menu.manager

Command	Function
bv	Sets a breakpoint when a variable changes value. The following variables are affected: ■ Global variables—those beginning with $G., $lo., $CHART., $SYSPUB., $MARQUEE. The RAD flow stops whenever these variables change value. ■ Normal variables—those not beginning with anything special or a parent ($P.) variable. The RAD flow stops if the variable has changed within the same RAD thread. If the RAD flow is in a different thread, the variable being tracked probably has a different purpose and a breakpoint is not desirable. ■ Local variables—those beginning with $L. The RAD flow stops only if the value of the variable changes for the RAD thread and application in which the variable was set. For example, if you are tracking a certain application that uses the local variable $L.qry, you do not want a breakpoint occurring in another application that happens to use the same variable name for another purpose.
bt	Sets a breakpoint on a specific type of RAD panel. When the RAD flow encounters a RAD command panel with this name, the application stops running and returns you to the RAD Debugger. Enter the command combined with the name of a RAD panel to set the breakpoint. Re-enter the same command combination to turn off the breakpoint. bt rinit
rb	Removes all breakpoints of any type
c (continue)	Resumes the execution of the RAD flow after a breakpoint occurs
s (step)	Steps through the RAD flow one panel at a time
gl (globals)	Displays all the global variables—those beginning with $G., $lo., $CHART., $SYSPUB., $MARQUEE.
v (variables)	Displays all the thread variables: those not beginning with a special code
l (locals)	Displays all the local RAD variables: those beginning with $L.
sta (stack)	Displays the current RAD stack
re (relations)	Displays only the variables which are file relations (file handles initialized by the **rinit** command panel).

Command	Function
m (memory	Displays all the variables in memory. This is equivalent to issuing the globals (**g**), variables (**v**), and locals (**l**) commands.
h (help)	Displays a brief summary of the RAD Debugger commands

Command Line Parameters

You can use the command line of a client shortcut to pass debugging parameters from an express client to the server. This features allows a user to obtain debugging information on a per-process basis without modifying the server `sc.ini` file. In addition, the express client can specify the name of a separate log file in which all the resulting information for the task is placed.

Parameters

Parameter	Function
debugstartup	Messages issued as the startup code is executed
debugprocesses	■ Messages issued as the startup code is executed ■ Termination messages
debugtransport	Messages dealing with the exchange of information between the client (full and express) and the server
debugrpc	Performance statistics for full client **rpc** calls or the latest data exchange in express mode
debugrs	Information about resource locks
debugdbquery	Query information
debugfileio	Messages regarding input/output to external files (NOT database input/output) Example: import/export, VSAM/QSAM/JES
debuglog	Names a special log to contain the debugging information. **Specify a file name only and not a path**. The debuglog will be placed in the same directory as the current *sc.log* file. -debuglog:<*logfile name*>
rtm:*n* RTM:*n*	Starts a RAD trace. For a detailed description of the RTM trace function, refer to *Special Parameters* in the ServiceCenter *Technical Reference*.

Example

A user is having an undetermined incident at startup. You want to execute an RTM trace in addition to obtaining the normal debugging information provided by *debugprocesses*, *debugstartup* and *debugtransport*. To simplify the job of deciphering the results, you want to place all the debugging information in a separate log file called support.log. The following command line syntax obtains the desired results:

```
scguiw32.exe -express:myhost.12680 -rtm:3 -debugprocesses
-debugstartup -debugtransport -debuglog:support.log
```

14 CHAPTER

Default Variables in ServiceCenter

The following global and local variables are used in the default ServiceCenter system.

Variable	Definition
$G.profiles	List of personal profiles
$G.pm.global.environment	Incident Management's global environment record
$G.category	List of category forms (*category* file)
$G.assignment	List of Incident Management assignment groups
$G.categories	List of categories
$G.problem.inbox	List of Incident Management inboxes
$G.open.lists	List of all global lists built at login
$G.icm.status	List of Inventory Management statuses
$G.assignment.groups	List of Incident Management assignment groups
$G.prompt.for.save	Global flag (boolean) determining whether or not you are prompted for save when *OK* or *Cancel* are pressed after record is updated.
$G.operators	List of ServiceCenter operators
$G.availability.maps	List of availability maps
$G.pm.environment	The current user's Incident Management personal profile record

Variable	Definition
$G.pm.status	List of Incident Management statuses
$G.groups	List of all PM profile groups in the system
$G.technicians	List of technicians (operator file)
$G.inbox.types	List of inbox types (incident, call)
$G.incident.inboxes	List of Service Management inboxes
$L.filed	This is the current file being displayed by the display application
$L.new	The current file being updated when evaluating macro conditions
$L.old	The pre-updated state of a file undergoing an update while evaluating macro conditions
$lo.appl.name	Application name passed to *menu.manager*
$lo.appl.names	Application parameter names passed
$lo.appl.values	Application parameter values passed
$lo.cm.limit	Partial-key time threshold for Change Management
$lo.company.name	Company name as defined in the Client record (company file)
$lo.copyright	Peregrine copyright (**copyright**())
$lo.date.order	Number representing date order (mdy, dmy, ymd)
$lo.db.limit	Partial-key time threshold for Database Manager
$lo.device	Device ID (current.device())
$lo.es	Vars not used
$lo.groups	List of query group names in operator record
$lo.home	Name of Home menu for GUI
$lo.i	Temporary variable (should be cleaned up in app)
$lo.main	Name of main menu for text
$lo.month.abv	List of abbreviated months ("Jan", "Feb", "Mar", etc.)
$lo.month.ext	List of months ("January", "February", "March", etc.)
$lo.msglog.lvl	Level of messages to be logged
$lo.pm.limit	Partial-key time threshold for Incident Management

Variable	Definition
$lo.printer	Default printer name
$lo.system.startup	Set at login to value of the field called **system.startup.time** in Company Information record.
$lo.time.zone	User's time zone
$lo.uallow.syslog	Determines whether syslog table should be updated when processes start and end in ServiceCenter.
$lo.uallow.timezone	Allows user to modify timezone based on capability
$lo.uapprovals	List of change approval groups
$lo.ucal	not used
$lo.ucalz	not used
$lo.ucapex	List of user's Capability words
$lo.uchgmgr	
$lo.uchgrps	List of change groups
$lo.ucm.print	Flag to indicate Change Management print default
$lo.ufname	User's full name
$lo.ulogin.time	User login time stamp
$lo.ulogoff.parm	* Obsolete *
$lo.upm.print	Flag to indicate Incident Management print default
$lo.user.name	**operator**()

15 CHAPTER Link Maintenance

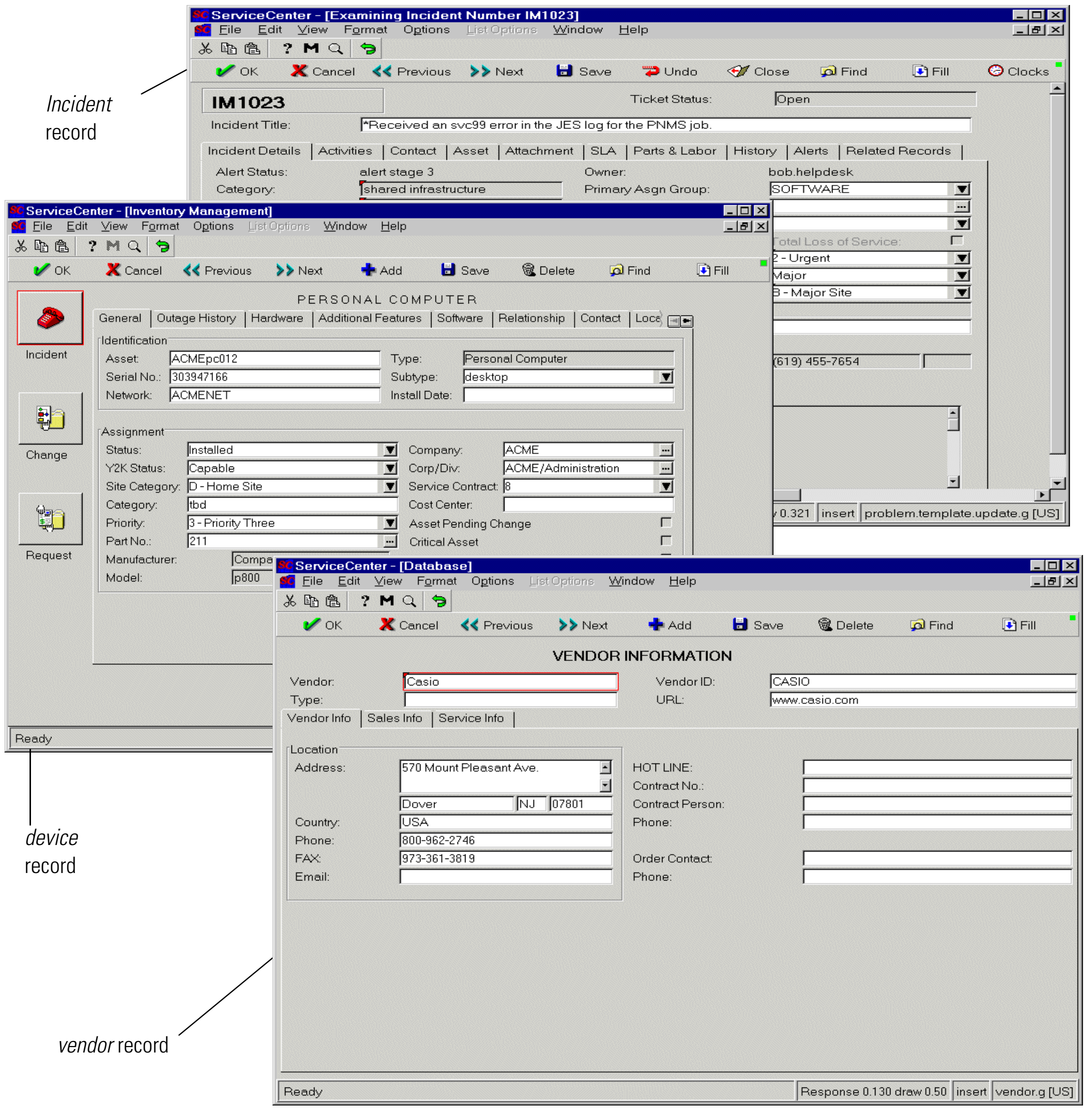
One of the advantages of a relational database is the elimination of redundant information. This is accomplished by storing information about a particular subject in one place, or file, with *links* to other subjects. Links are a combination of data and *link **definitions,*** sets of conditions containing the relationships for linked information. Links are used within ServiceCenter Incident Management (IM) and Inventory and Configuration Management (ICM) environments to relate information in one file to information in another.

Data Relationships and the Link File

Link maintenance involves establishing the relationships between ServiceCenter data so that information residing in one file can form the *link query* that selects and displays or copies information from another file.

A separate link definition can occur for each form (format).

Consider the three records in Figure 15-1 on page 338. The incident document (in this example *problem.network.update*) requires information regarding the serial number, location, and vendor, as well as a description of the incident.


Incident record
device record
vendor record
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Save Undo Close Find Fill Clocks
Add Delete
Incident
Change
Request

ServiceCenter - [Examining Incident Number IM1023]

IM1023 — Ticket Status: Open

Incident Title: *Received an svc99 error in the JES log for the PNMS job.

Incident Details | Activities | Contact | Asset | Attachment | SLA | Parts & Labor | History | Alerts | Related Records

Alert Status: alert stage 3 — Owner: bob.helpdesk

Category: shared infrastructure — Primary Asgn Group: SOFTWARE

Total Loss of Service:

2 - Urgent

Major

3 - Major Site

(619) 455-7654

0.321 | insert | problem.template.update.g [US]

ServiceCenter - [Inventory Management]

PERSONAL COMPUTER

General | Outage History | Hardware | Additional Features | Software | Relationship | Contact | Loca

Identification

Asset:	ACMEpc012	Type:	Personal Computer
Serial No.:	303947166	Subtype:	desktop
Network:	ACMENET	Install Date:	

Assignment

Status:	Installed	Company:	ACME
Y2K Status:	Capable	Corp/Div:	ACME/Administration
Site Category:	D - Home Site	Service Contract:	8
Category:	tbd	Cost Center:	
Priority:	3 - Priority Three	Asset Pending Change	
Part No.:	211	Critical Asset	
Manufacturer:	Compa		
Model:	p800		

Ready

ServiceCenter - [Database]

VENDOR INFORMATION

Vendor:	Casio	Vendor ID:	CASIO
Type:		URL:	www.casio.com

Vendor Info | Sales Info | Service Info

Location

Address:	570 Mount Pleasant Ave.	HOT LINE:	
		Contract No.:	
	Dover NJ 07801	Contract Person:	
Country:	USA	Phone:	
Phone:	800-962-2746		
FAX:	973-361-3819	Order Contact:	
Email:		Phone:	

Ready | Response 0.130 draw 0.50 | insert | vendor.g [US]

Figure 15-1: Database Records

The information required when documenting an incident is often already in the database. To find and display the information with the incident document, you must *tell* the system:

- What value it should use as a *search argument*, and
- Where it should look for the matching information.

This data is stored in the inventory system, referenced according to several categories of information. The *device* record (device type *pc* in Figure 15-1 on page 338) contains the **logical name, contact** and **location** names. The *vendor* record contains the **vendor ID**, **location** and **vendor phone number**.

The information required for the incident document is stored in at least two separate records, so at least two relationships are defined in the link record: one to *device*, one to *vendor*, and one to any other applicable file.

Within each link, there is also an implied order to how these relationships are defined. The *device* record may contain the vendor's name, but not the vendor's phone number, and the contact name is stored in the *location* record but maybe not in the device record.

By defining the first relationship between the incident document and the device record, you can retrieve the data necessary to form the queries which will retrieve data through the second and third relationships.

A link record defines the relationship between the incident document and information in the *device*, *vendor* and other records. Although you may have different link records for each format (form) used in Incident Management, a record named **problem** and one named **probsummary** always exist. These records provide a basis and example for other Incident Management link records.

Note: If you are using IR Expert, consider building links from one of the IR fields (like *action*) to itself. This allows a simple **Find** command to locate relevant incidents before a ticket is opened.

To reach the link record for the problem file:

1 Select **Tools** from the **Utilities** tab in the administrator's home menu.

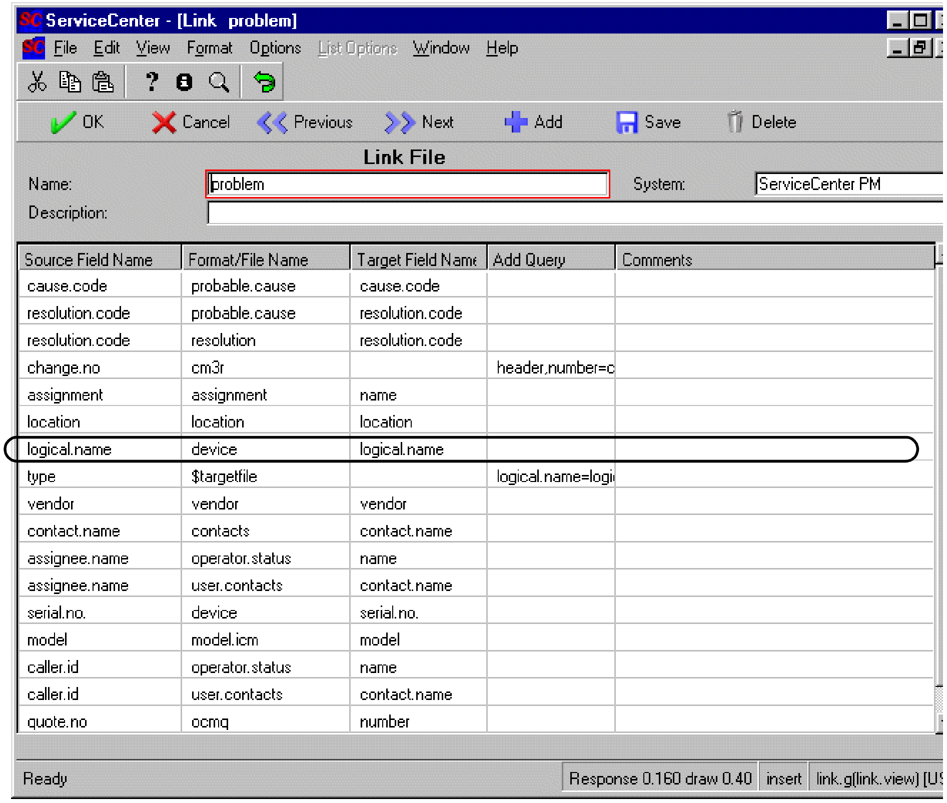
2 Select the **Links** button; this brings up **Link Manager.**

3 Enter problem in the **Form** field.

4 Press **Enter**.

5 Select *problem* from the record list.

(For the **problem.network link** record, click on *problem.network* in the QBE.)


ServiceCenter - [Link problem]
File Edit View Format Options List Options Window Help
OK
Cancel
Previous
Next
Add
Save
Delete

Link File

Name:	problem	System:	ServiceCenter PM
Description:			

Source Field Name	Format/File Name	Target Field Name	Add Query	Comments
cause.code	probable.cause	cause.code		
resolution.code	probable.cause	resolution.code		
resolution.code	resolution	resolution.code		
change.no	cm3r		header.number=c	
assignment	assignment	name		
location	location	location		
logical.name	device	logical.name		
type	$targetfile		logical.name=logi	
vendor	vendor	vendor		
contact.name	contacts	contact.name		
assignee.name	operator.status	name		
assignee.name	user.contacts	contact.name		
serial.no.	device	serial.no.		
model	model.icm	model		
caller.id	operator.status	name		
caller.id	user.contacts	contact.name		
quote.no	ocmq	number		

Ready   Response 0.160 draw 0.40   insert   link.g(link.view) [U

Figure 15-2: Link Maintenance record

The column labeled *Field Name (SOURCE)* contains the names of fields in the current record (in this case *problem*). The columns under the label *Link To And/Or Fill From (TARGET)* contain the file names and corresponding field names that define the relationship. For example, the first entry links the ***logical.name*** field to the logical.name field in the **device record** file. When the link is exercised, the contents of the logical.name field in the **problem** record will be used to search the device file for other information to populate the form. Using the examples on the previous page, the link query `logical.name#“pc010”` is exercised against the *device* file in search of that device’s logical name and other applicable data.

The inventory items are linked to the incident document (*problem.network* in this example) by the field labeled **Asset ID** (*logical.name*) field. The first link to the device record retrieves the *serial number* and *location* names. Subsequent links use that information to form their own relationships, e.g., the value of the field labeled **Contact Name** (*vendor* field) is retrieved from the vendor record, based on the value of the logical name passed to Inventory.

A link to the vendor record from the field labeled **Service Provider** (on any linked Incident Management form) provides the Vendor Phone number for the incident document.

Inventory data stored on *device*, *vendor* and other records can be linked to several forms in Incident Management, and in the same way multiple relationships for the same field can exist. These link relationships are specified in the *link* record. Within the link record, links between data and forms are established in the order in which they appear in the list. A query for a specific field may be linked to more than one inventory record in the process of searching for the necessary data. See Figure 15-3 on page 342.

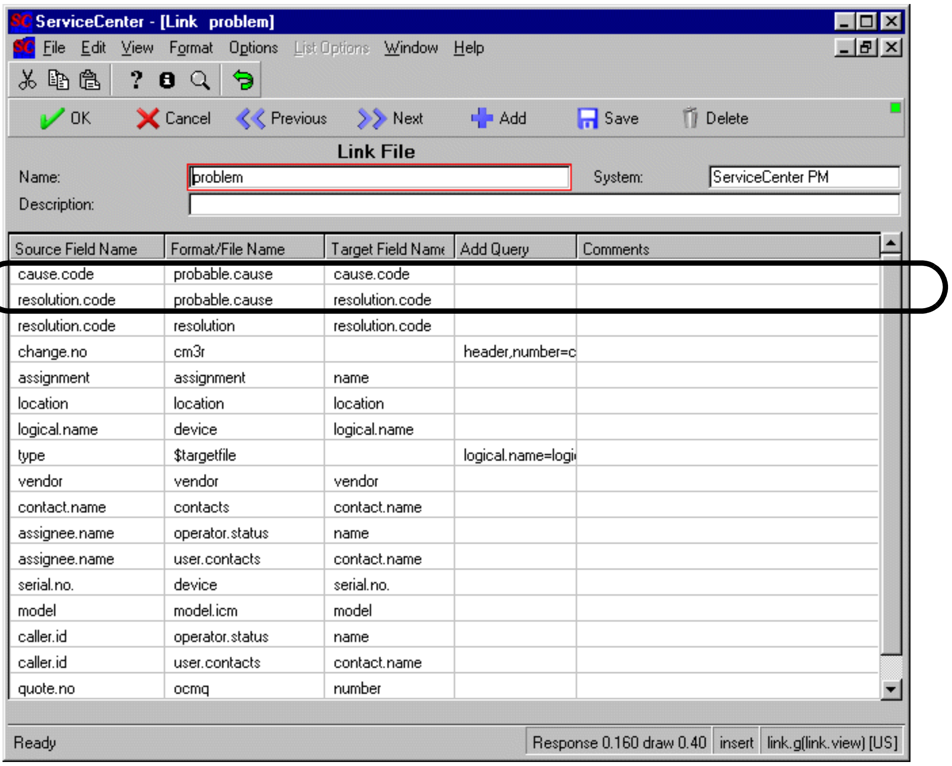
Multiple line links

In Figure 15-3 on page 342, *the* resolution.code field is linked to both the probable.cause file and the resolution file. The order in which the entries appear in the *link* record determines the order in which they are searched by ServiceCenter applications. For example, the *probable.cause* file is searched first. If no records are found to satisfy the link query, the *resolution* file is searched before the search stops. Searching stops when a query is satisfied.

Find, Fill and Virtual Join

Links form the basis for the most powerful features of ServiceCenter Incident Management applications, those of *Find, Fill* and **Virtual Join.**

The Find function uses the value of the field in which the cursor is placed to query for information based on data stored in existing records. Security parameters may restrict a user's access to the requested data, but under most conditions if information is found, a record list of matching inventory records is displayed. If no information is found, a negative message is issued.


ServiceCenter - [Link problem]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete

Link File

Name: problem
System: ServiceCenter PM
Description:

Source Field Name	Format/File Name	Target Field Name	Add Query	Comments
cause.code	probable.cause	cause.code		
resolution.code	probable.cause	resolution.code		
resolution.code	resolution	resolution.code		
change.no	cm3r		header,number=c	
assignment	assignment	name		
location	location	location		
logical.name	device	logical.name		
type	$targetfile		logical.name=logi	
vendor	vendor	vendor		
contact.name	contacts	contact.name		
assignee.name	operator.status	name		
assignee.name	user.contacts	contact.name		
serial.no.	device	serial.no.		
model	model.icm	model		
caller.id	operator.status	name		
caller.id	user.contacts	contact.name		
quote.no	ocmq	number		

Ready | Response 0.160 draw 0.40 | insert | link.g(link.view) [US]

Figure 15-3: Link Record - problem

The Fill function works in a similar fashion, selected information is copied into the current record from existing records. For example, incident documents are populated with device information when a logical name (logical.name) is specified.

Another feature which relies on link definitions is Virtual Join. This tool combines information from many files and presents the results on a single form. At the Help Desk, a vendor's telephone number can be displayed in a incident document without occupying any space in the actual record. Information presented through virtual joins does not reside in long-term memory and is simply another way of looking at current inventory data.

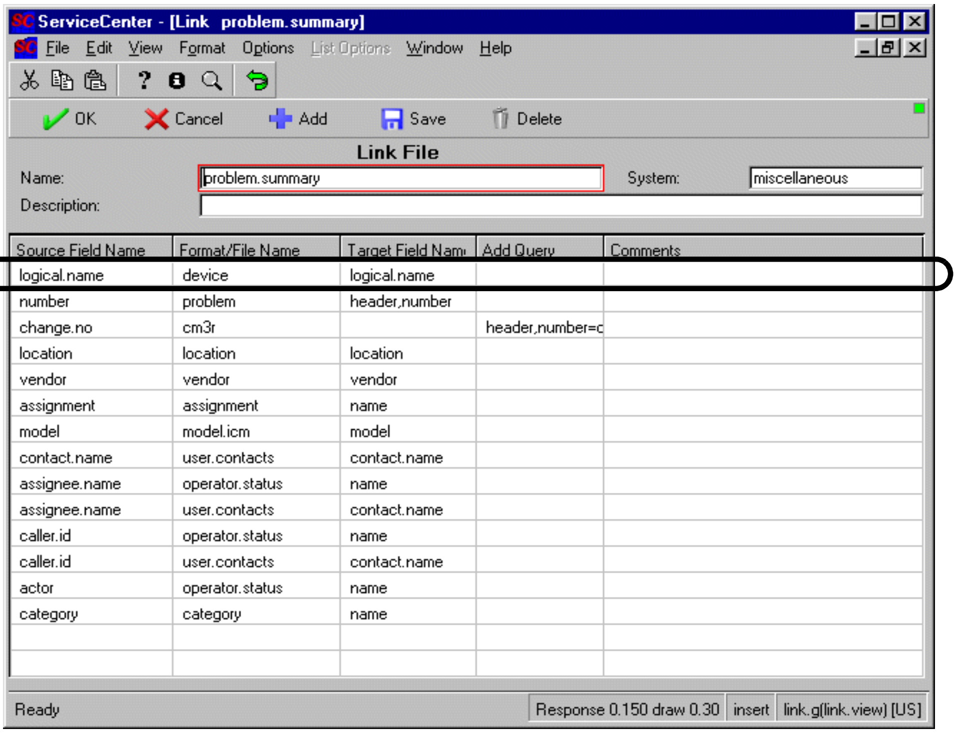
Linking in this method enables data to be current and consistent across all incident documents. A change to the original inventory record automatically updates the incident document whenever it is opened. Each time a incident document is opened or updated, the applicable fields displaying linked data pick up the most recent inventory information.

Important: Queries for Find, Fill and Virtual Joins only perform appropriately when run on fields which have been set up as unique keys in the Database Dictionary record for the target files. These unique key definitions allow data to be stored using certain markers. Link queries need to locate these markers in order to retrieve the appropriate data.

See the Forms Designer chapters for more information on setting up unique keys and other definitions in the Database Dictionary record.

Find Functionality

Using information in the *link* record, **Find** locates and displays information in another record (or records) based on the contents of the current record. (The **find** application has been replaced by the **us.link** application in release **A9802** and later releases. See the section *Us.link* for more information.)


ServiceCenter - [Link problem.summary]
File Edit View Format Options List Options Window Help
OK Cancel Add Save Delete
Link File
Ready
Response 0.150 draw 0.30
insert
link.g(link.view) [US]

Name: problem.summary
System: miscellaneous
Description:

Source Field Name	Format/File Name	Target Field Nam	Add Query	Comments
logical.name	device	logical.name		
number	problem	header,number		
change.no	cm3r		header,number=c	
location	location	location		
vendor	vendor	vendor		
assignment	assignment	name		
model	model.icm	model		
contact.name	user.contacts	contact.name		
assignee.name	operator.status	name		
assignee.name	user.contacts	contact.name		
caller.id	operator.status	name		
caller.id	user.contacts	contact.name		
actor	operator.status	name		
category	category	name		

Figure 15-4: Link Record/File - problem.summary

In the following Incident Management example, the source record contains pc in the field labeled *Asset ID* (*logical.name* field). The link record shown in Figure 15-4 on page 344 relates the *logical.name* field in the SOURCE record to the **logical.name** field in the vendor (TARGET) file.

All device records containing logical.name fields beginning with *pc* will be selected and displayed in a QBE list for the user to select.

Then, the user selects the specific device sought and opens the applicable device record.

Find uses the *field name* of the textbox where the cursor currently is in order to determine what relationship to establish. To select the device records whose names begin with *pc*, the cursor must be resting in the *Asset ID* field, which must contain *pc*.

It builds the following query:

```
logical.name field in device file begins with "pc"
```

which translates in ServiceCenter to the following link query which is executed against the **vendor** file:

```
device#"pc"
```

Note: Using advanced features in Link Maintenance, you can apply rules to use a more complex link query for data selection. Refer to the section entitled *Using Advanced Link Editing Features* for more information.

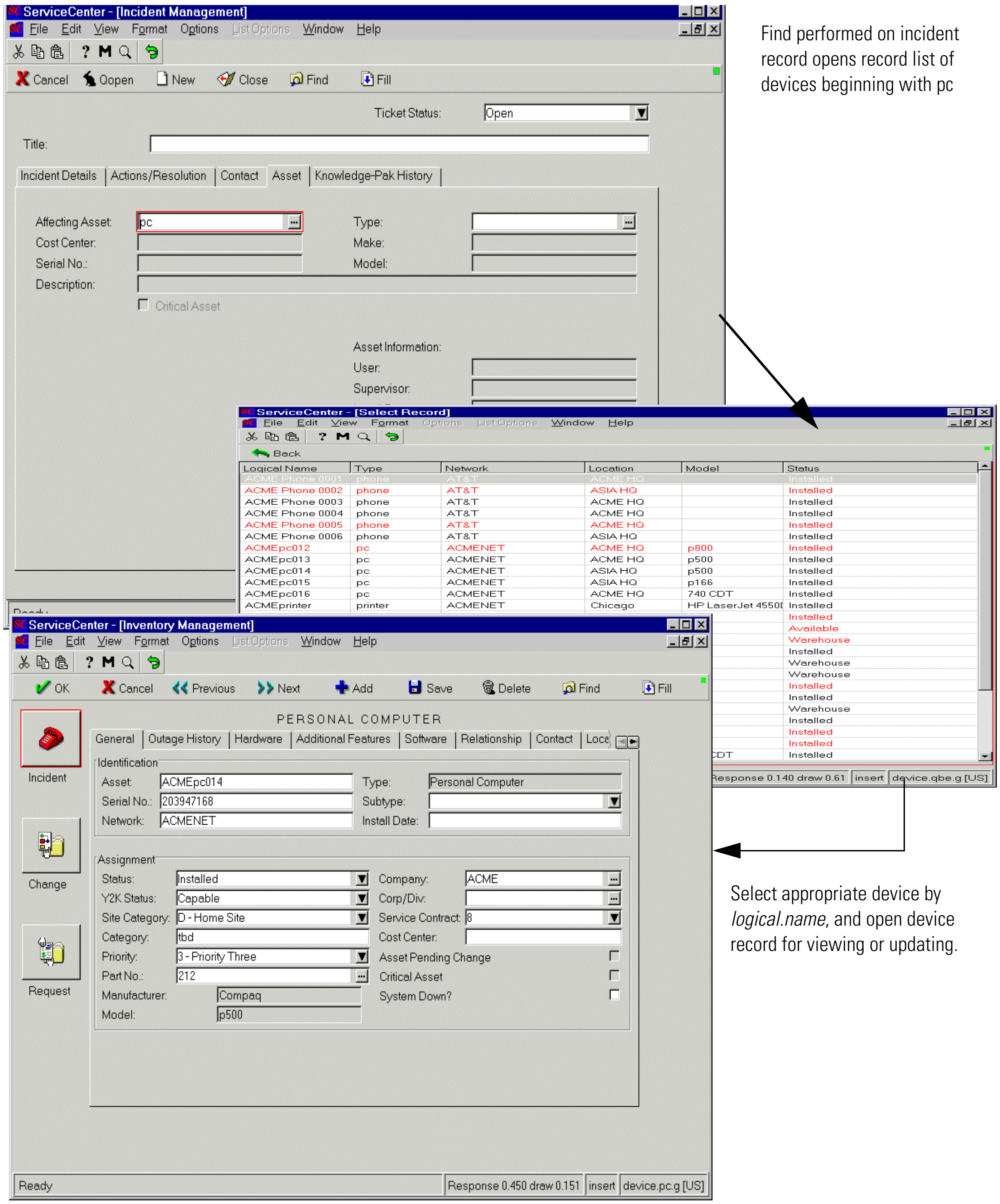
The Find option is available throughout ServiceCenter, subject to security restrictions within the application in use. For example, the Database Manager controls the Find option through the use of format control. In Incident Management applications, the profile determines whether Find is available for the current user and how it behaves.

When Find is used, the selected data records can be manipulated in accordance with security restrictions. Changes made to these records *DO NOT* modify the source record unless the user selects from an available list of options which allow updated information to be posted to the specific inventory record. For more information on posting, see the *Posting* chapter in the **Format Control** topic and *Display Options* in the Display application section.

Important: Remember, **Find** uses normal ServiceCenter data selection rules. Therefore, links to key fields will perform much more efficiently than links to non-keyed fields.

Fill Functionality

Using information in the *link* file, *Fill* locates information in another record and copies it into the current record.



ServiceCenter - [Incident Management]

Ticket Status:	Open
Title:	
Affecting Asset:	pc
Type:	
Cost Center:	
Make:	
Serial No.:	
Model:	
Description:	
Critical Asset	☐
Asset Information:	
User:	
Supervisor:	

Find performed on incident record opens record list of devices beginning with pc

ServiceCenter - [Select Record]

Logical Name	Type	Network	Location	Model	Status
ACME Phone 0001	phone	AT&T	ACME HQ		Installed
ACME Phone 0002	phone	AT&T	ASIA HQ		Installed
ACME Phone 0003	phone	AT&T	ACME HQ		Installed
ACME Phone 0004	phone	AT&T	ACME HQ		Installed
ACME Phone 0005	phone	AT&T	ACME HQ		Installed
ACME Phone 0006	phone	AT&T	ASIA HQ		Installed
ACMEpc012	pc	ACMENET	ACME HQ	p800	Installed
ACMEpc013	pc	ACMENET	ACME HQ	p500	Installed
ACMEpc014	pc	ACMENET	ASIA HQ	p500	Installed
ACMEpc015	pc	ACMENET	ASIA HQ	p166	Installed
ACMEpc016	pc	ACMENET	ACME HQ	740 CDT	Installed
ACMEprinter	printer	ACMENET	Chicago	HP LaserJet 4550	Installed
					Installed
					Available
					Warehouse
					Installed
					Warehouse
					Warehouse
					Installed
					Installed
					Warehouse
					Installed
					Installed
					Installed
				CDT	Installed

Response 0.140 draw 0.61 | insert | device.qbe.g [US]

ServiceCenter - [Inventory Management]

PERSONAL COMPUTER

Identification

Asset:	ACMEpc014	Type:	Personal Computer
Serial No.:	203947168	Subtype:	
Network:	ACMENET	Install Date:	

Assignment

Status:	Installed	Company:	ACME
Y2K Status:	Capable	Corp/Div:	
Site Category:	D - Home Site	Service Contract:	8
Category:	tbd	Cost Center:	
Priority:	3 - Priority Three	Asset Pending Change	☐
Part No.:	212	Critical Asset	☐
Manufacturer:	Compaq	System Down?	☐
Model:	p500		

Ready | Response 0.450 draw 0.151 | insert | device.pc.g [US]

Select appropriate device by *logical.name*, and open device record for viewing or updating.

Figure 15-5: Find Feature

The link record for the incident file is used in Incident Management.

In the following example, Fill is used to populate fields in an incident document based on the value of the field labeled Asset ID.

Rather than selecting the **device** record for display, fields in the **device** (TARGET) file that have the same name as the fields in the **problem** (SOURCE) file are projected into the *source* record.

In this way, fields on the *source* form are populated automatically with data directly from the applicable device record, e.g., *Asset ID* (logical.name) on the incident record is filled with the value of *Asset* (logical.name) from the device record, the field labeled *Type* on the incident form is filled with the value of *Type* from the device record, and the value for the field labeled *Category* is copied from the device record to the incident record.

Through this transfer, the *TARGET* record is unchanged, but the *SOURCE* record is modified to reflect the values of fields in the *TARGET* record.

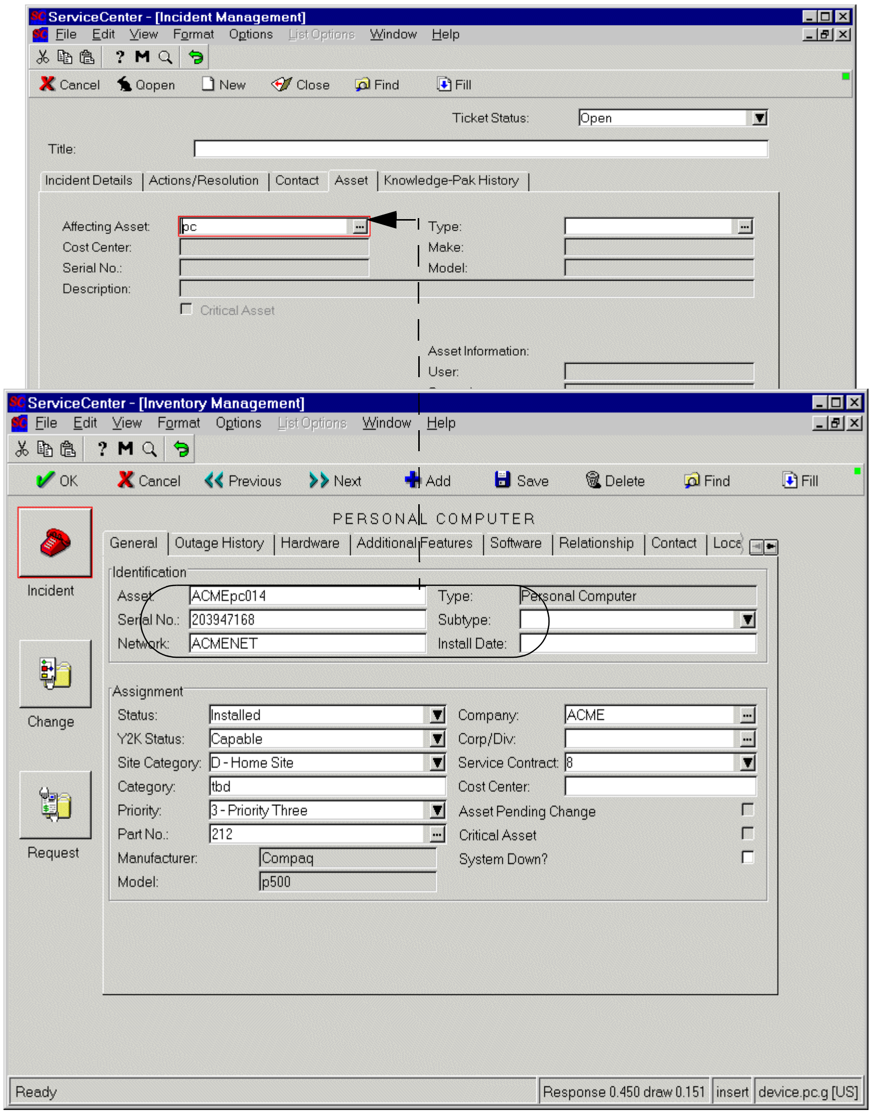
The resulting modified *SOURCE* record is not written to the database until the system is instructed to do so with some action (e.g., **open** or **update** the problem).

The **Fill** option uses the value in the field where the cursor is positioned to determine which link relationship to use. This example assumes that the cursor was in the **Asset ID** *(logical.name)* input field when the *Fill* option was selected.

Fill should be used when it is necessary to *store* information in the *source* record so that it can be changed or used as a link to other information. If you simply need to display the information in a format, use *Virtual Join*, introduced in the following section and described in detail in *Creating Virtual Joins* on page 373.

Virtual Joining Functionality

The *Virtual Join* function allows information from many files to be displayed on a single form. Virtually joined information cannot be modified, but it can be used to link with other files using Find and Fill. (The **find and fill** applications have been replaced by the **us.link** application in release **A9802** and later releases. See the section *Us.link* for more information.)


ServiceCenter - [Incident Management]
File Edit View Format Options List Options Window Help
Cancel Oopen New Close Find Fill
Incident Details
Actions/Resolution
Contact
Asset
Knowledge-Pak History
ServiceCenter - [Inventory Management]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete Find Fill
Incident
Change
Request
General
Outage History
Hardware
Additional Features
Software
Relationship
Contact
Loca

Ticket Status: Open

Title:

Affecting Asset: pc

Cost Center:

Serial No.:

Description:

Critical Asset

Type:

Make:

Model:

Asset Information:

User:

PERSONAL COMPUTER

Identification

Asset:	ACMEpc014	Type:	Personal Computer
Serial No.:	203947168	Subtype:	
Network:	ACMENET	Install Date:	

Assignment

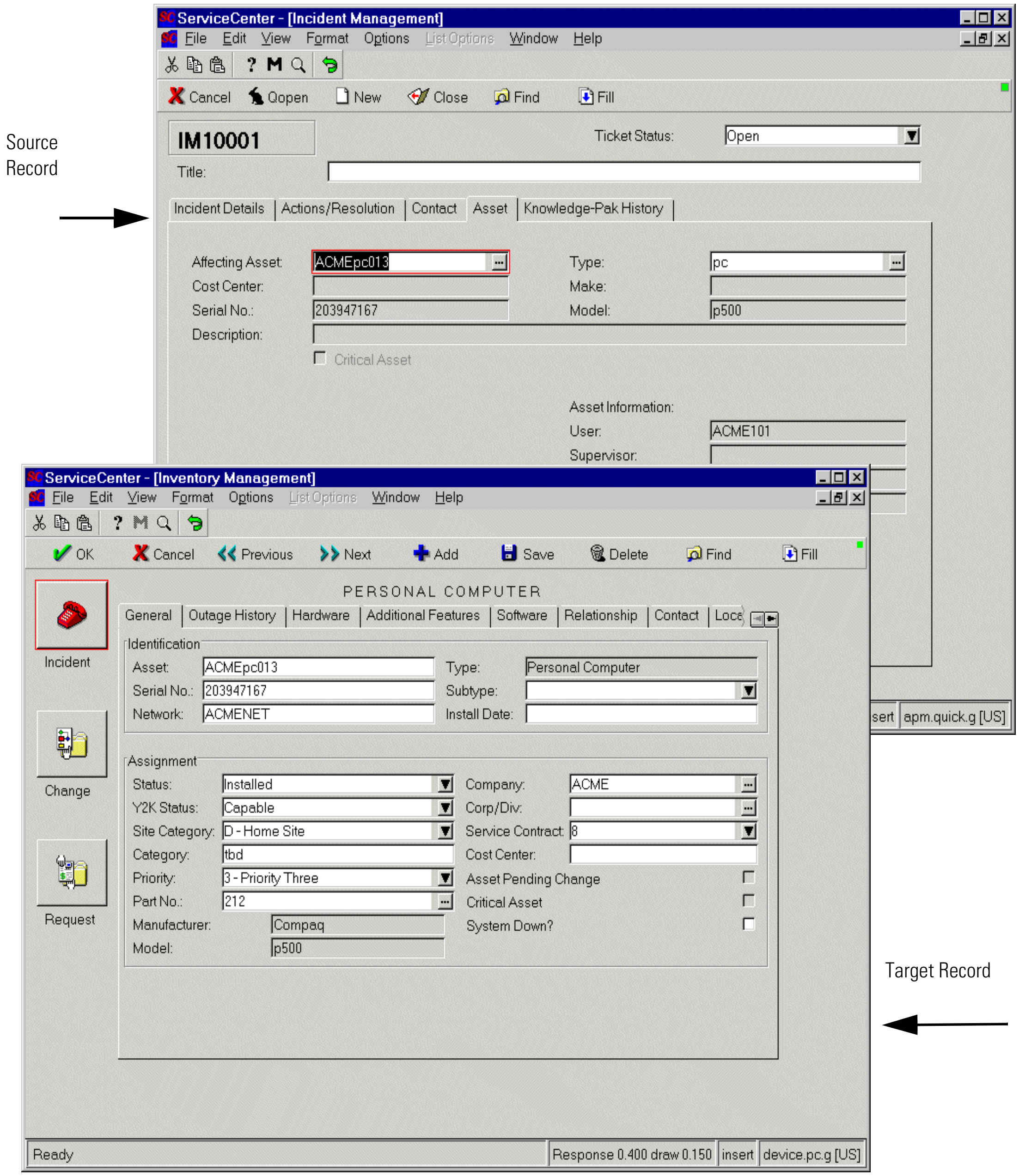
Status:	Installed	Company:	ACME
Y2K Status:	Capable	Corp/Div:	
Site Category:	D - Home Site	Service Contract:	8
Category:	tbd	Cost Center:	
Priority:	3 - Priority Three	Asset Pending Change	☐
Part No.:	212	Critical Asset	☐
Manufacturer:	Compaq	System Down?	☐
Model:	p500		

Ready | Response 0.450 draw 0.151 | insert | device.pc.g [US]

Figure 15-6: Linked Source and Target Records

When specifying link information, there are special requirements for virtual joins:

- The **TARGET Format/File Name** value in the link record must contain a file name, not a format name.
- The target field must be a non-concatenated key in the target file.


Source Record
ServiceCenter - [Incident Management]
File Edit View Format Options List Options Window Help
Cancel Oopen New Close Find Fill
Incident Details Actions/Resolution Contact Asset Knowledge-Pak History
Critical Asset
apm.quick.g [US]
Target Record
ServiceCenter - [Inventory Management]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete Find Fill
Incident
Change
Request
General Outage History Hardware Additional Features Software Relationship Contact Loca
Asset Pending Change
Critical Asset
System Down?

IM10001

Ticket Status: Open

Title:

Affecting Asset: ACMEpc013

Type: pc

Cost Center:

Make:

Serial No.: 203947167

Model: p500

Description:

Asset Information:

User: ACME101

Supervisor:

PERSONAL COMPUTER

Identification

Asset: ACMEpc013

Type: Personal Computer

Serial No.: 203947167

Subtype:

Network: ACMENET

Install Date:

Assignment

Status: Installed

Company: ACME

Y2K Status: Capable

Corp/Div:

Site Category: D - Home Site

Service Contract: 8

Category: tbd

Cost Center:

Priority: 3 - Priority Three

Part No.: 212

Manufacturer: Compaq

Model: p500

Ready | Response 0.400 draw 0.150 | insert | device.pc.g [US]

Figure 15-7: Filled (Populated) Source Record

- The target field must be the first instance of the key in the database dictionary’s key array.

- The target field must be a scalar field (non-array).
- Only simple links can be executed using **Virtual Join**.

While the appearance of virtually joined information is the same as if it existed in the record being displayed, the information is stored in one place and simply displayed on demand to another. It also allows information from many files to be displayed on a single screen.

Note: **Virtual join** can be used in any format except QBE list formats. Virtually displayed information can be used as a **source** field to other information using **Find** or **Fill**, but nested virtual joins are not supported.

Us.link

This section describes the Universal **Services- Link** (**us.link**) application.

The **us.link** application replaces **find**, **fill** and **fill.recurse** for several reasons:

- To take advantage of the current ServiceCenter environment in order to speed up transactions.
- To have a common rule base behind each of these applications.
- To make the find/fill process easier to debug.

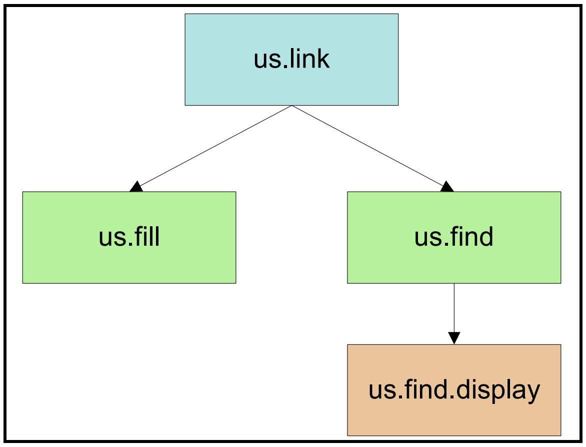
Changes to find, fill, and fill.recurse

The **find**, **fill**, and **fill.recurse** applications are now single panels calling the **us.link** routine. No changes are necessary to existing code that calls one of these routines.

us.link

The **us.link** application is responsible for selecting the correct records from the correct file and then passing control to either **us.find** or **us.fill**.

Application hierarchy



Calling us.link

The **us.link** application can be called using the following parameters.

Note: An effort was made to keep the parameter names as familiar as possible, but this was not always feasible since **fill** and **fill.recurse** use different parameter names for the same fields.

Parameter Name	Description	Default
record	Source Record (required)	None
name	Field name to find from/link to	Current field
string1	Format Name	Current format
second.record	Link Record (optional)	None
prompt	Action (“find” or “fill”)	“find”
boolean1	Background flag	False

The only required parameter for the use of **us.link** is the source record.

Access to $File / Dates

The *$File* variable can now be modified directly using the link expressions, and changes will be saved back into the source record. Additionally, when performing a fill on a date field, the system will first check to see if that date field is in the link record. If it is, the link expressions will be performed rather than just filling the current date and time. In this way it is possible to fill with something other than **tod**() (such as **date**(**tod**()) or **tod**() + '**7 00:00:00**').

Find from / fill to a $ variable

The **us.link** application makes it possible to perform a find from a field which has a *$* variable as an input, and to use fill to move information into a *$* variable field. Using a variable as a source field is accomplished simply by placing the variable name in the **Source Field** column of a link record as you would place a standard field name. To fill to a specific variable, that variable is placed on the **source field**(**fill to/post from**) column on the specific link line.

Note: No changes to the posting routine have been made, therefore posting does not allow use of *$* variables at this time.

Find from / fill to a structured array

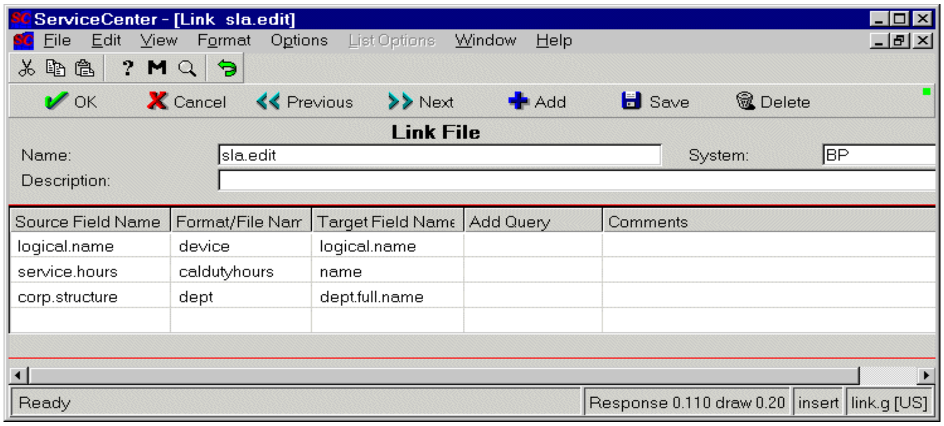
The **us.link** application makes it possible to perform a find from a field which is part of a structured array, and to use fill to move information into that specific element of the structured array. This can only be performed under the following circumstances, however:

- The field name to fill from/post to is part of the structured array.
- The field name to fill from/post to is not used in any other structure of the dbdict.
- The name of the structure must be the same as the name of the array of which it is a part.

To use find/fill on a structured array, it is necessary to set up the variable *$fill.structure* in the expressions of the specific link line. *$fill.structure* is an array of two elements. The first is the index of the field (that is used as the source field) in the structure, and is of type `number`. The second is the name of the structure, and is of type `character`.

When using the fill function on a structured array, only the specific line of the structure being accessed can be modified. Also, the index number of the field within the structure must be used instead of the actual field name. This includes the source field.

Below is an example of filling the **availability** structured array of an SLA record using the logical name column as the source. This link will fill the **logical.name** field (index 1) with the logical name of a device, and the **calendar** field (index 3) with the **table.name** of the device.

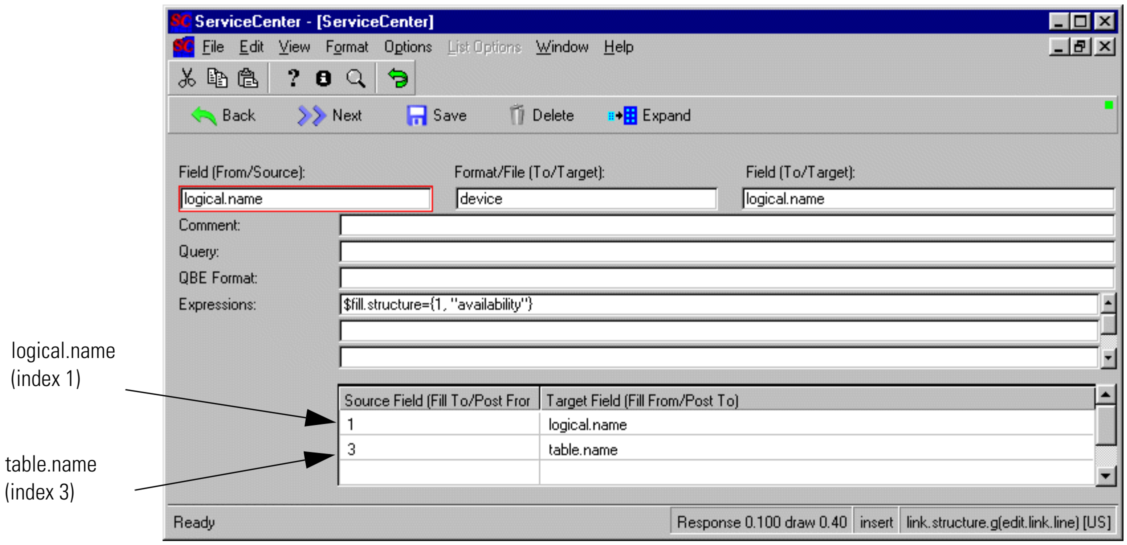

ServiceCenter - [Link sla.edit]
File Edit View Format Options List Options Window Help
OK Cancel Previous Next Add Save Delete
Link File

Name: sla.edit System: BP
Description:

Source Field Name	Format/File Narr	Target Field Name	Add Query	Comments
logical.name	device	logical.name		
service.hours	caldutyhours	name		
corp.structure	dept	dept.full.name		

Ready | Response 0.110 draw 0.20 | insert | link.g [US]

Figure 15-8: Link Record for Service Level Agreement Edit form (sla.edit)


logical.name
(index 1)
table.name
(index 3)
ServiceCenter - [ServiceCenter]
File Edit View Format Options List Options Window Help
Back Next Save Delete Expand

Field (From/Source): logical.name
Format/File (To/Target): device
Field (To/Target): logical.name
Comment:
Query:
QBE Format:
Expressions: $fill.structure={1, "availability"}

Source Field (Fill To/Post Fror	Target Field (Fill From/Post To)
1	logical.name
3	table.name

Ready | Response 0.100 draw 0.40 | insert | link.structure.g(edit.link.line) [US]

Figure 15-9: Link Line Structure (logical.name) Record

Variables used by us.link

The special variables used by **us.link** are the same as those used by the prior applications unless noted.

Variable	Use
$fill.replace	Governs whether or not a field that contains data should be overwritten with new data (including NULL). Only used for the project portion of a fill, not fields specified on the **fill to/fill from** section of the link line.
$project.first	Governs whether or not a project should be done before moving any of the fields defined in the list of **fill to/fill from** fields.
$fill.exact	Feature of the project panel that forces it to adhere strictly to data types and index levels during the project.
$fill.recurse	Determines whether you should move to the next entry in the link record (perform a recursive fill).

$fill.skip	Tells the application to skip the current entry and continue based upon the value of $fill.recurse for that entry.
$fill.option.skip	Enables the skip option to skip the current link entry and move on when displaying the results of a recursive fill.
$fill.option.copy	This option is no longer necessary when using us.link.
$fill.display	Determines whether or not the record you are filling from should be displayed before you copy data from it into your source record.
$fill.display.add	Determines whether or not you may add a record if the current link query returns no records.
$fill.search.format	Specify a search screen when using fill by setting $fill.search.format in a link expression. If this variable is set, and the field which you are filling is NULL, the search screen will be displayed to the user. The search created here continues the standard fill process.

Changes to $fill.display, $fill.display.add

In the **us.link** applications, the *$fill.display* functionality has been modified, while new functionality has been added through *$fill.display.add*.

- When using *$fill.display*, a user can modify the data being presented using the standard rules applying to that database (these rules are defined in the format control). Once the data is modified,
  - Press **OK** to save the record and use the new version of the record to perform the fill function based on the associated link line.
  - Press **Cancel** to keep any updates performed by using the **Save** key, but will not fill information back into the source record.
- When *$fill.display.add* is set to true and the standard link query performed by **us.link** returns no records, the user will be presented with the format defined in the link line and may add a record (if the associated format control allows it). Any information defined in the **fill to/fill from** fields will be copied into this record.
  - Press the **Add** button to add the record while leaving the user on this format. At this point the rules for *$fill.display* will be followed.
  - Press **OK** to add the record and perform the fill function back to the source record using the newly added record.

16 CHAPTER

Understanding Links

This chapter covers linking within ServiceCenter. The following material demonstrates how to:

- Access Link Definitions
- *Add* a new link
- *Modify* an existing link
- Use advanced Link Editing features
- Link dependencies within the Help Desk environment

Accessing Links

1 From the administrator's home menu, click the **Utilities** tab.

2 Click the **Tools** button.

3 Click the **Link** button. The Link file panel appears.

Fields on the Link File Format

Field	Value
Name	The name of the link record. If you got to link format from Forms Design, then ServiceCenter automatically defaults the source filename to the one you specified on the Forms Design screen.
System	The **System** field is used to categorize link records by ServiceCenter application. For example, the link records beginning with *device* are classified as PHD/ICM (Inventory and Configuration Management) links.
Description	Enter a brief description of the link (optional).
Source Field Name	Enter the field name in the source file that relates to a field in the target file.
Format/File Name	Enter the Target File name. The target file contains the information the Source file will access.
Target Field Name	Enter the field name in the Target file that relates to a field in the Source file.
Add Query	If you enter a query or conditional statement in this field, you are overriding the standard link query. See *Advanced Link Editing Features* on page 361.
Comments	This field is optional. Peregrine recommends entering comments for future reference.

Adding a New Link File

1 From the administrator's home menu, click the **Utilities** tab.

2 Click the **Tools** button.

3 Click the **Link** button. The Link file panel appears.

4 Enter the name of the form or file and, if desired, a descriptive name or phrase in the System field at the Link Manager prompt.

5 After entering the Form Name and System fields, select the **New** button from the system tray, or the button displaying the blank page, to create the new *link* record.

The record is displayed for editing. The link record appears without any relationship definitions.

6 Enter the relationships in the appropriate columns

7 Click **Save** to add/update the new link record.

Testing a link

The locations format is used for this example:

1 From the System Administrators home menu, click the **Toolkit** tab.

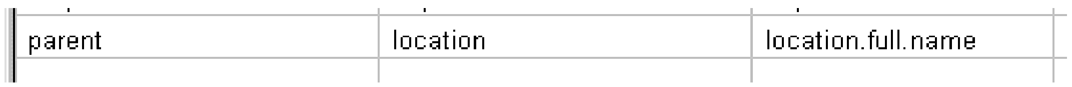
2 Click the **Database Manager** button.

3 Enter location in the form field and Press Enter or click **Search**.
The location file appears.

4 In the **Parent Location** field, enter the letter **P**.

Click *Find*. A list of records appears. The system returns this particular list of records because of the link definition for this file. The **Location Full Name** for each these records starts with PRGN, which is why a capitol P returns the list you see.

If we look at the Link file for location, we can see that the **Parent** field is linked to the **Location Full Name** field.



parent	location	location.full.name

Note: Link records are case-sensitive, which means the case of the values used to search must be the same as the values saved in the *TARGET* file in order for the appropriate data to be retrieved, i.e. *PRGN* does not equal *prgn*.

Modify an Existing Link

1. From the administrator's home menu, click the **Utilities** tab.
2. Click the **Tools** button.
3. Click the **Link** button. The Link file panel appears.
4. Enter the name of a link record and click Search or press **Enter**.

   If you press **Enter** without entering a value, select a link record from the standard QBE.

   When the link record appears, a number of options are available.

Options pull-down menu

Option	Value
Insert Line	Opens a window to prompt for the number of lines to insert, then inserts them above the cursor position.
Delete Line	Opens a window to prompt for the number of lines to delete, then deletes them beginning with the line the cursor is on.
Select Line	Allows advanced link processing. See *Advanced Link Editing Features* on page 361.
Check Field	When the cursor is positioned on a Source Field Name or Target Field Name, prompts for a file name and then checks the database dictionary of the file to determine whether a field of that name exists; if invalid, allows selection of a valid field.

Delete a Link

1. From the administrator's home menu, click the **Utilities** tab.
2. Click the **Tools** button.

3 Click the **Link** button. The Link file panel appears.

4 Enter a link name, or press Enter at the Link Maintenance prompt to return a list of all link records on your system.

5 After selecting the appropriate link record, use the **Delete** key in the system tray.

Delete *always* prompts for confirmation.

6 Select **OK** on the confirmation prompt to complete the delete operation.

You are returned to the link record prompt screen, link.prompt.

Advanced Link Editing Features

Simple links define the relationship between a specific field in a source record and a specific field in a target record. The Find and Fill options perform in a straightforward manner: a link query is built based upon a search argument which contains the value of a field in the *source* file and the name of a field in the *target* file. These fields need to be unique key fields for simple linking.

ServiceCenter provides extended flexibility to define complex link expressions. For example, you can:

- Define a specific query that uses more than one field to form the link query.
- Specify the QBE format that should be used when the link query finds more than one record.
- Define a variable, rather than a specific name, to be used as the target file.
- Manipulate the value of fields by using link expressions.
- Specify that particular fields are copied from the target file to specific fields in the source file during the Fill operation without a requirement for identical field names.
- Include multiple non-keyed fields on the link line structure of a keyed field, enabling those fields to be copied to the *source* form along with the data retrieved from the keyed field query.

Note: Advanced link features are not available when using Virtual Join.

Use the following steps to access Advanced Link Maintenance:

1 From the administrator's home menu, click the **Utilities** tab.

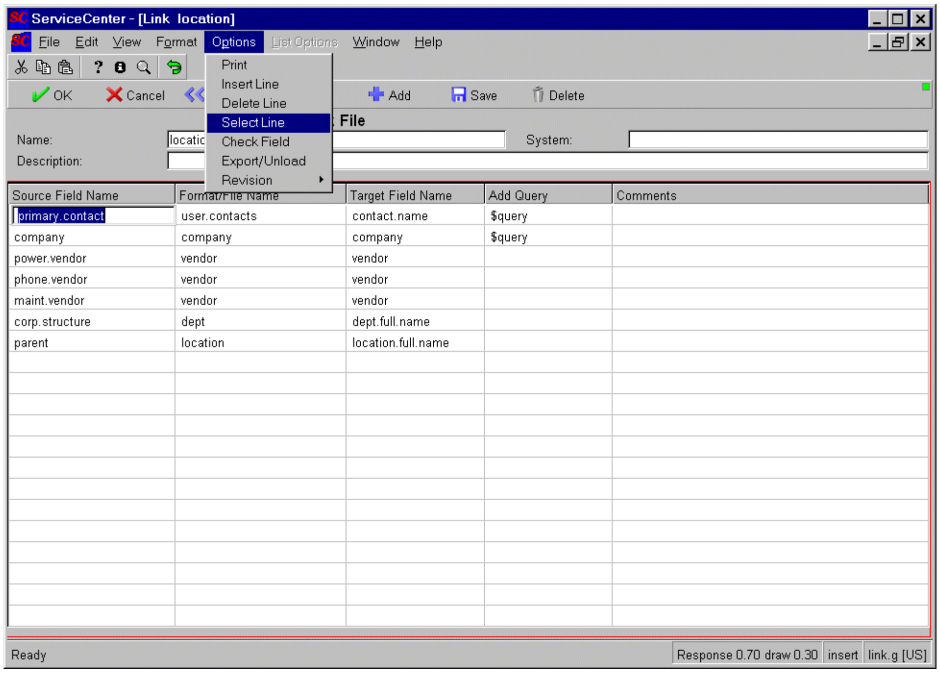
2 Click the **Tools** button.

3 Click the **Link** button. The Link file panel appears.

4 Enter a link name, or press Enter at the Link Maintenance prompt to return a list of all link records on your system.

5 Select the link to edit.

For example, select the locations link record.



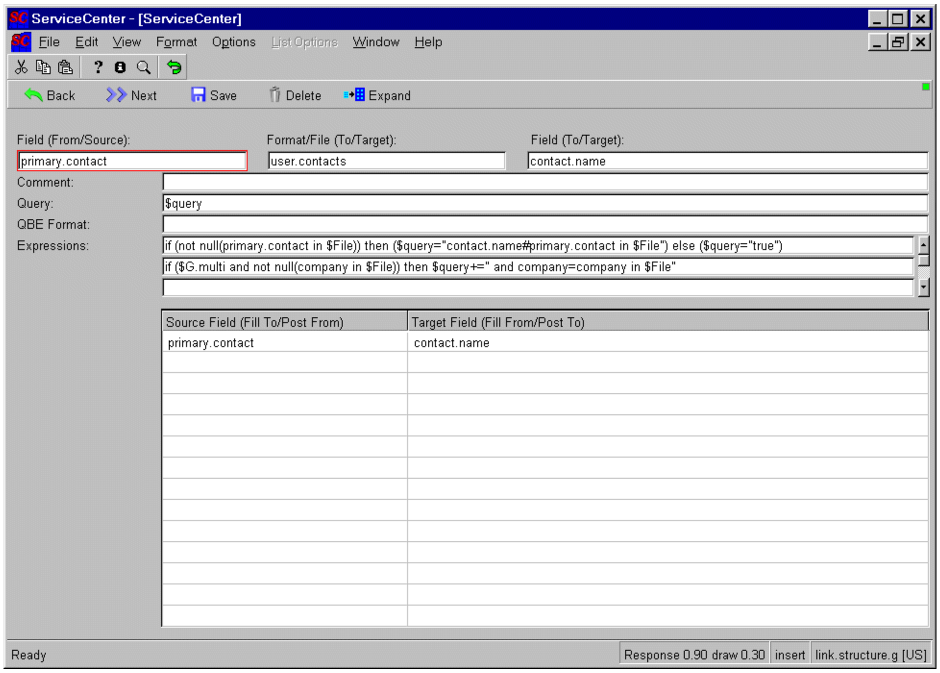
Source Field Name	Format/File Name	Target Field Name	Add Query	Comments
primary.contact	user.contacts	contact.name	$query	
company	company	company	$query	
power.vendor	vendor	vendor		
phone.vendor	vendor	vendor		
maint.vendor	vendor	vendor		
corp.structure	dept	dept.full.name		
parent	location	location.full.name		

Figure 16-1: locations Link Record

6 Position the cursor on the line to be edited

7 Select Options -> Select Line from the menu bar.

The Source Field and Target Format/File Names are copied to the *link structure* form.


ServiceCenter - [ServiceCenter]
File Edit View Format Options List Options Window Help
Back Next Save Delete Expand

Field (From/Source):	primary.contact	Format/File (To/Target):	user.contacts
Field (To/Target):	contact.name		
Comment:			
Query:	$query		
QBE Format:			
Expressions:	if (not null(primary.contact in $File)) then ($query="contact.name#primary.contact in $File") else ($query="true")		
	if ($G.multi and not null(company in $File)) then $query+=" and company=company in $File"		

Source Field (Fill To/Post From)	Target Field (Fill From/Post To)
primary.contact	contact.name

Ready | Response 0.90 draw 0.30 | insert | link.structure.g [US]

Figure 16-2: Location Link Structure

New fields are available, and new option keys are enabled

Fields on the Link Structure Format

Field	Value
Comment	This field contains optional fields; describes use of advanced link features.
Query	This field contains a specific link query that overrides the standard link query. The general rule for specific link queries is: `target field=source field in $File` For example: `vendor=vendor in $File and city=city in $File` The file variable **$File** (with a mandatory capital *F*) is used for all references to the *source* file in *Fill* and *Find* operations.
QBE Format	Specify the name of the QBE format to use if more than one record is selected in *Find* or *Fill*. This field is optional. If you do not specify a QBE format, the system will use the default.
Expressions	For example, you may want to modify the *target* file name depending on a value in the *source* record. In such a case, an expression would set the value of the file variable: `if type in $File="terminal" then $tfile="workstation"`

Field	Value
Source Field (Fill To/Post From)	This list of field names is used by the Fill function as an alternate to project (which copies only identically named fields). If this list contains field names, then Fill will copy the values contained in fields in the corresponding Target Field (Fill From) entries without respect to field name. Although field names do not have to match, there must be a one-to-one correspondence between entries in the Source Field array and the Target Field array.
Target Field (Fill From/Post To)	This list of field names is used by the Fill function as an alternate to project (which copies only identically named fields). If this list contains field names, then Fill will copy the values in these fields to fields in the corresponding Source Field (Fill To) entries without respect to field name. Although field names do not have to match, there must be a one-to-one correspondence between entries in the Target Field array and the Source Field array.

Note: In Find and Fill operations, expressions are evaluated first, then the link query is built and executed, and data is copied last (in the case of Fill). The link records used in data conversions for PM and ICM, and in building the probsummary record in PM, are controlled by a different function.

Link Structure Options Menu

The Option pull-down menu keys are similar to those available in the primary Link Maintenance screen.

Option	Value
Insert Line	Opens a window to prompt for the number of lines to insert, then inserts them above the cursor position.
Delete Line	Opens a window to prompt for the number of lines to delete, then deletes them beginning with the line the cursor is on.
Check Field Name	When the cursor is positioned on a Source Field (Fill To) or Target Field (Fill From), prompts for a file name and then checks the database dictionary of the file to determine whether a field of that name exists; if invalid, allows selection of a valid field.
Check Field Name	When the cursor is positioned on a **Source Field** (Fill To) or **Target Field** (Fill From), prompts for a file name and then checks the database dictionary of the file to determine whether a field of that name exists; if invalid, allows selection of a valid field.

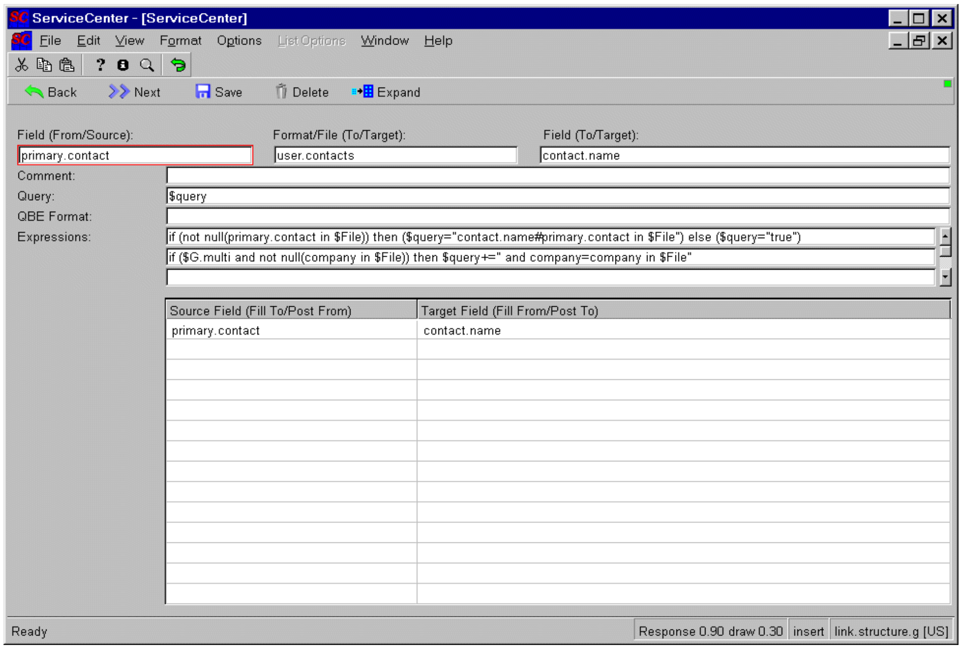
Note: Confirmation of changes to a line entry does not modify the link record in the database; exiting from the *link* record will prompt for confirmation, and it is this confirmation that updates the record in the database.

Specifying a Link Query

If you want to control the search criteria used by *Find* and *Fill*, you can define a specific link query.

There can be instances when the link is dependent on more than the simple value of fields in the records. For example, links from an incident document could be dependent on the device type. Link Maintenance allows specification of expressions, which are evaluated by the *Fill* and *Find* operations before the link query is built.

Whenever the link relationship varies according to data in the *source* record, using expressions and a variable for the *target* file is the most efficient method for establishing a link.


ServiceCenter - [ServiceCenter]
File Edit View Format Options List Options Window Help
Back Next Save Delete Expand

Field (From/Source):	primary.contact
Format/File (To/Target):	user.contacts
Field (To/Target):	contact.name
Comment:	
Query:	$query
QBE Format:	

Expressions:

```
if (not null(primary.contact in $File)) then ($query="contact.name#primary.contact in $File") else ($query="true")
if ($G.multi and not null(company in $File)) then $query+=" and company=company in $File"
```

Source Field (Fill To/Post From)	Target Field (Fill From/Post To)
primary.contact	contact.name

```
Ready    Response 0.90 draw 0.30    insert    link.structure.g [US]
```

Figure 16-3: Link Structure - primary contact field of the location link record

Figure 16-3 on page 367 shows the expression that defines *$query* for this link record. The first query in the figure states that if there is something in the primary contact field, then the system will search for contact names that start with what is in the field. If the field is blank, then the system will return a QBE list of all names. Expressions are processed in the order they are placed on the format.

Copying Fields by Name During Fill Operations

Normally, the Fill operation copies fields of the same name from the target file to the source file using a ServiceCenter function called **project**.

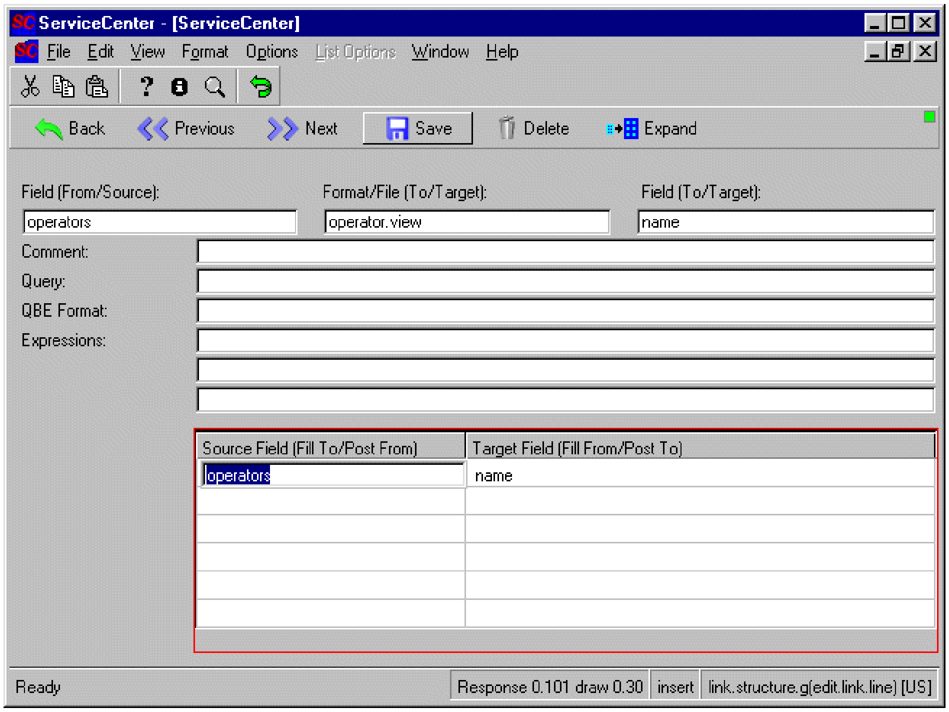
For example, if you were to open an incident document and place the cursor on the *logical.name* input field and then choose the Fill option, the system would search the device file for a record with the same logical.name.The system would then copy fields with the same name from the device file into the incident document.

The incident document would contain information about the device that was copied from like-named fields in the device record.

As long as the field names are the same, and the information being copied is from an equal or higher level, the **project** function will perform normally. When the field names are dissimilar, or if the information to be copied is from structure fields to scalar fields, a different method must be used.

To copy information from structure fields to scalar fields:

1. Open the link record you want to work with
2. Place your cursor on the line containing the link
3. Click Options>Select line
4. Switch the fields by entering the structure field in Source Field (Fill To) and the scalar field in Target Field (Fill From) copies the contents of.


ServiceCenter - [ServiceCenter]
File Edit View Format Options List Options Window Help
Back Previous Next Save Delete Expand

Field (From/Source): operators

Format/File (To/Target): operator.view

Field (To/Target): name

Comment:

Query:

QBE Format:

Expressions:

Source Field (Fill To/Post From)	Target Field (Fill From/Post To)
operators	name

Ready | Response 0.101 draw 0.30 | insert | link.structure.g(edit.link.line) [US]

Figure 16-4: Link query setup for assignment field

Scalar/NonScalar field links

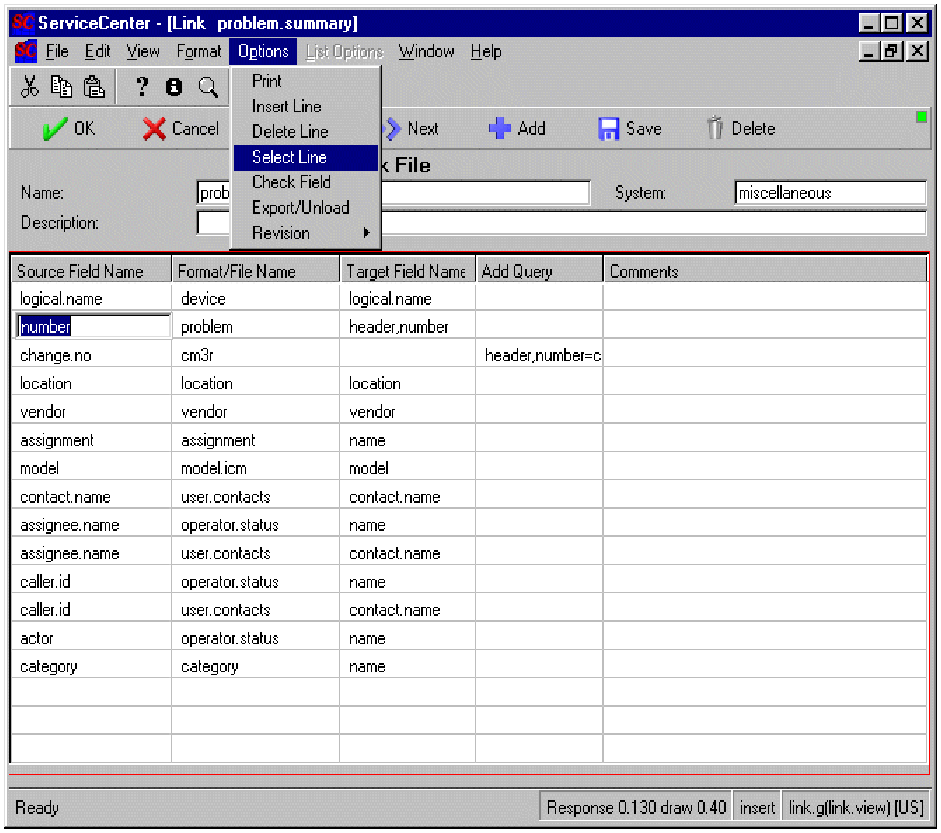
A simple link cannot copy information from a **non-scalar** field (composed of more than one data element of the same type, i.e. an array) to a **scalar** field (composed of a single data element). For example, information from a record in the incident file cannot be projected into a **probsummary** record. Even though the field names are the same, the fields in the incident record are not **scalar** fields, but rather exist in one of the three structures that make up the **problem** ***database descriptor.*** In fact, the fully-qualified name of the assignment field in the incident record is really *header,assignment*.

ServiceCenter can project the logical.name value in the incident file into the incident record's logical.name field. ServiceCenter cannot project the incident file's non-scalar middle,logical.name field into the scalar logical.name field in the incident file. To copy non-scalar information to a scalar field, the Fill operation copies each field using instructions in the link record.

Using the link record for the problem.summary format, (Figure 16-5 on page 370), we can see how Fill copies data on a field by field basis:

1 Place your cursor in the line that links number in the incident file to header,number.

2 Click Options -> Select Line



Name: prob

Description:

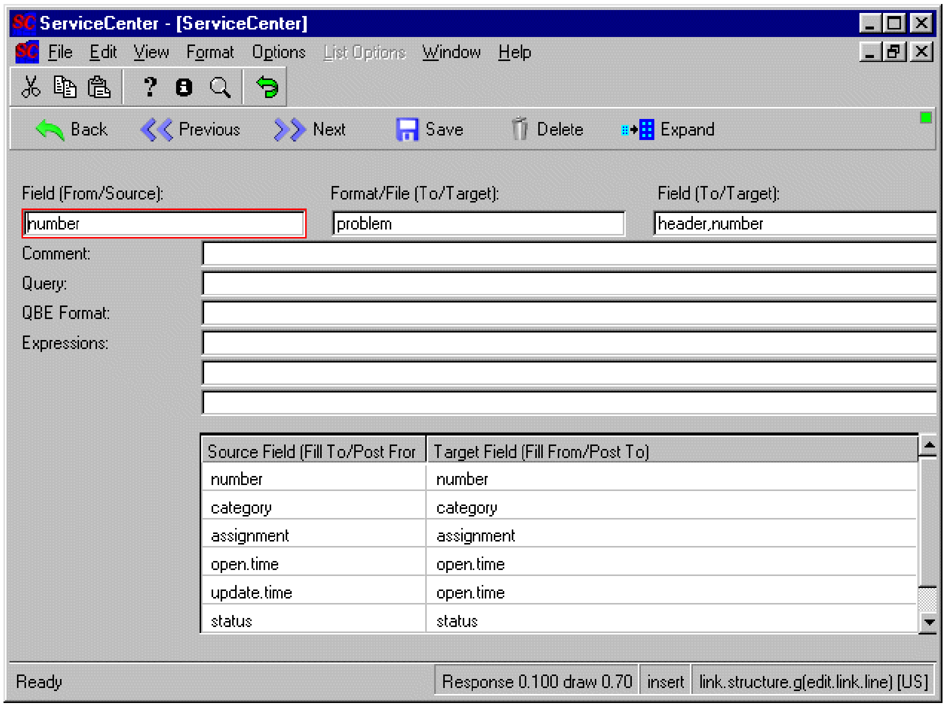
System: miscellaneous

Source Field Name	Format/File Name	Target Field Name	Add Query	Comments
logical.name	device	logical.name		
number	problem	header,number		
change.no	cm3r		header,number=c	
location	location	location		
vendor	vendor	vendor		
assignment	assignment	name		
model	model.icm	model		
contact.name	user.contacts	contact.name		
assignee.name	operator.status	name		
assignee.name	user.contacts	contact.name		
caller.id	operator.status	name		
caller.id	user.contacts	contact.name		
actor	operator.status	name		
category	category	name		

Ready | Response 0.130 draw 0.40 | insert | link.g(link.view) [US]

Figure 16-5: Problem.summary Link Record

Figure 16-6 on page 371 shows the fields listed in the Target Field (Fill From) column will be copied from the incident record to the corresponding fields in the probsummary record.



ServiceCenter - [ServiceCenter]

Field (From/Source): number

Format/File (To/Target): problem

Field (To/Target): header,number

Comment:

Query:

QBE Format:

Expressions:

Source Field (Fill To/Post Fror	Target Field (Fill From/Post To)
number	number
category	category
assignment	assignment
open.time	open.time
update.time	open.time
status	status

Ready | Response 0.100 draw 0.70 | insert | link.structure.g(edit.link.line) [US]

Figure 16-6: Link query setup for number field

This method of copying information is also especially useful when you need to copy only a few of the commonly named fields from one record to another.

Keeping Changes

There is no Update option when editing a link line entry. Rather, a copy of the link line entry is made when it is selected, and that copy is compared to the current line entry when you exit using Back, Close Application, Next Entry or Previous Entry.

Remember that when a link line entry is modified, the change is not written to the database until you exit from the *link* record and confirm the update action.

Link Dependencies within the Help Desk

Two special *link* records are used within the Help Desk applications. They perform the conversion of problem records to *probsummary* records and the conversion of ServiceCenter inventory files to the *entity* and *attribute* files.

The *build.inventory.files link* record converts information in the various ServiceCenter *inventory* files to the Inventory and Configuration Management device file and their associated *attribute* files. It cannot be used for any other purpose since the *$file* variable has special meaning in the conversion application.

A second special *link* record, *build.problem.summary*, is used to copy information from the problem record to the *probsummary* record whenever the problem is updated. The *$pfile* variable is used in place of the normal *$File* variable when building expressions, and the expressions are processed **after** the fields are copied rather than before as in normal link processing.

Document Engine Master Link Record

For each object accessed through the document engine, a master link record is combined with the form-dependent link record. The master link record is valid for the entire file, and always has the same name as the Object/filename. For example, if the filename is *contacts*, the master link record is the *contacts* link record.

17 Virtual Joins

CHAPTER

Virtual Joins allow you to display data fields from several different database files on a single form. Virtual Joins do not allow data entry into the joined fields; they only display information from other files into a format for reference purposes.

A Virtual Join can be used on any format. Virtual Joins are established when data from a record is displayed without the need to use the **Find** and **Fill** function keys of the Link Utility. You can also use Virtual Joins in **Report Writer** instead of secondary file queries through Format Control. (See the *Report Writer Guide* for more information.)

Creating Virtual Joins

The following illustrates the steps involved in retrieving data from a **file** called *sales* and displaying data in a **form** called *orders*.

The following sections will walk through building the forms and subformats needed for the examples.

The formats and subformats will be built are:

- sales
- sales1 subformat
- orders

The first form needed is the *sales* form with at least one record. The file in the database associated with this format is the target file from which data will be retrieved to display in the *order* file.

Subformats allow the creation of multi-part forms, where sections of the form can be reproduced in other forms. These modular forms enable the simultaneous display of data from numerous database records via Virtual Joins. Subformats are also useful in constructing multiple views of the same subformat data (e.g., Incident Management Views).

Note: **code** must be a single unique key (not concatenated) in the target file. Also, the source and target fields must be **scalar** (non-array).

Understanding Subformats

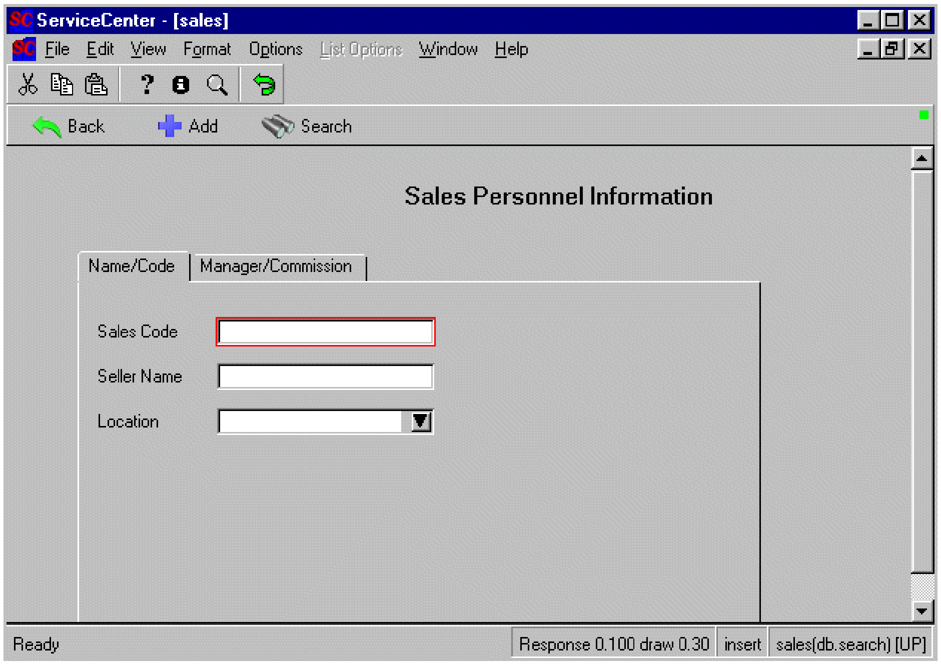
Subformats are constructed in the same way as standard formats, with the exception that they are created to appear within another, larger, format. Subformats can be micro versions of larger forms, containing specific key information that is valuable for display on other formats (e.g., inventory data subformats which appear on incident ticket formats).

A subformat can be used on any format as long as it is properly placed using Forms Designer. Data from a record can be displayed in a subformat automatically or through the use of the **Find** and **Fill** function keys of the Link Utility.

How to Create a Subformat

The following screens illustrate the steps involved in creating a format called *sales* to enter data into a file called *sales*, and a subformat called *sales1*, which can also be used to enter and retrieve data from the *sales* file. The field names within the two forms will be the same, because they are referencing the same data (file), but may be organized to look different. The subformat will be used to create a virtual join on another form.

The following steps will lead you through the construction of the form and file called *sales*, the subformat *sales1*, and the `orders` format.



Sales Personnel Information

Name/Code | Manager/Commission

Sales Code

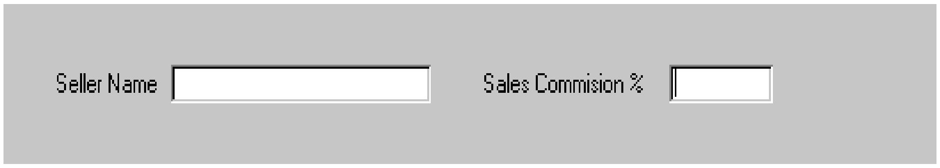
Seller Name

Location

Ready | Response 0.100 draw 0.30 | insert | sales(db.search) [UP]

Figure 17-1: Sales format

The first format, *sales*, will be used to input sales personnel data, which will be retrieved and displayed by the subformat, *sales1*, wherever it appears or is virtually joined on other formats.



Seller Name

Sales Commision %

Figure 17-2: Sales1 subformat

Building the *sales* Form

1 Access Forms Designer.

2 Enter sales in the Form field.

3 Click **New**.

4 Click **Design** to open the designer tool.

- Decline the Wizard tool when prompted.

5 Start constructing the main format by adding a title across the top, e.g., *Sales Personnel Information*, using the Label tool.

6 Create two tabs by selecting the **Notebook** button.

7 Name the front tab *Name/Code* and the back tab *Manager/Commission*.

8 Add a textbox and label to the *Name/Code* tab.

a Place a label called **Sales Code**.

b Place a textbox next to the label with the input value *code*.

9 Add a textbox and label to the *Name/Code* tab.

a Place a label called **Seller Name**

b Place a textbox next to the label with the input value *name*.

10 Add a Combo Box and label to the *Name/Code* tab.

a Place a label called **Location**.

b Place a Combo Box next to the label with the input value *location*.

c In the Properties box, add a list of at least four city names in both the **Displaylist** and **Valuelist** fields.

11 Click on the *Manager/Commission* tab.

12 Add a textbox and label to the *Manager/Commission* tab.

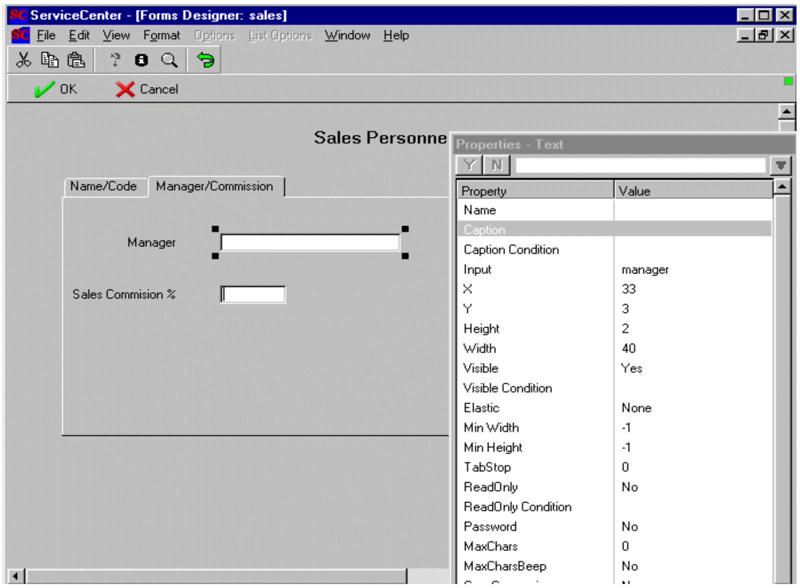
a Place a label called **Manager**.

b Place a textbox next to the label with the input value *manager*.

13 Add a textbox and label to the *Manager/Commission* tab.

a Place a label called **Sales Commission %**.

b Place a textbox next to the label with the input value *commission*.


ServiceCenter - [Forms Designer: sales]
File Edit View Format Options List Options Window Help
OK
Cancel
Sales Personne
Name/Code
Manager/Commission
Manager
Sales Commision %
Properties - Text

Property	Value
Name	
Caption	
Caption Condition	
Input	manager
X	33
Y	3
Height	2
Width	40
Visible	Yes
Visible Condition	
Elastic	None
Min Width	-1
Min Height	-1
TabStop	0
ReadOnly	No
ReadOnly Condition	
Password	No
MaxChars	0
MaxCharsBeep	No

Figure 17-3: field settings on the Sales form

14 Click **OK** to confirm the new form and **OK** again to exit the Forms Designer and save your new form.

The main form *sales* is now complete and should appear as in Figure 17-1 on page 375.

Now you must create a Database Dictionary file so data can be entered through the form and recorded to the database.

Creating the *sales* File

Return to Forms Designer for the following procedures.

1 Pull down the **Options** menu and select *Create File.*

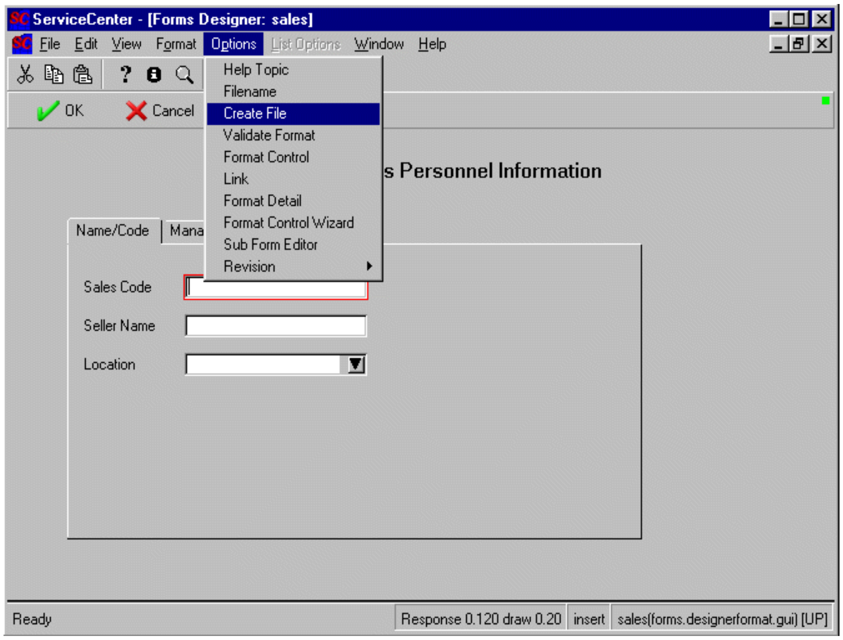


Figure 17-4: Creating the sales Database Dictionary

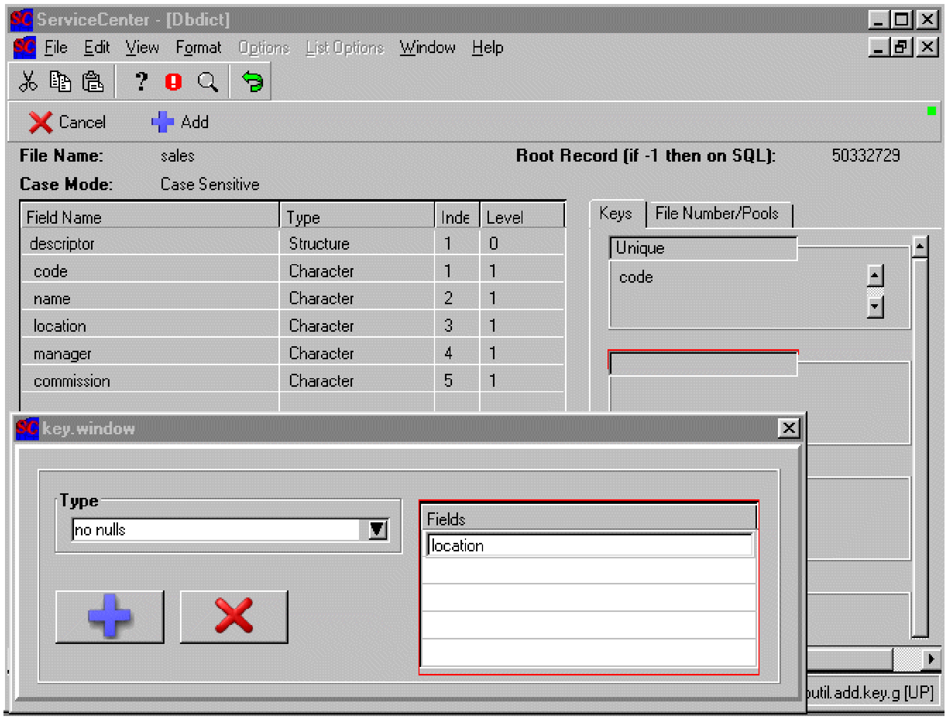
2 Select **OK** at the prompt to accept the default name for the file, *sales*.

This will create a Database Dictionary record and file, which will become the target file for your subformat when it appears on other forms.

3 The Database Dictionary file is generated automatically, with the exception of the *no nulls* key for the location field.

4 To add the **location** key, click the next empty slot on the Keys tab. (See Figure 17-5 on page 379.)

5 Click **New.**



File Name: sales
Root Record (if -1 then on SQL): 50332729
Case Mode: Case Sensitive

Field Name	Type	Inde	Level
descriptor	Structure	1	0
code	Character	1	1
name	Character	2	1
location	Character	3	1
manager	Character	4	1
commission	Character	5	1

Unique
code

Type: no nulls
Fields: location

Figure 17-5: Database Dictionary Record

6 Enter *location* and select *no nulls* for the key type.

7 Click **Add**.

8 Click **OK**, to confirm the addition and save the Database Dictionary entry.

9 Click **OK** to regenerate the file.

The main *sales* data form and file are now complete.

Creating the Sales QBE

Now create a QBE list for the *Sales* format.

To create a QBE list:

1 Open the Forms Design utility

2 Enter *sales.qbe* in the **Form** field, and click **New**. Select **No** at the Forms Wizard prompt.

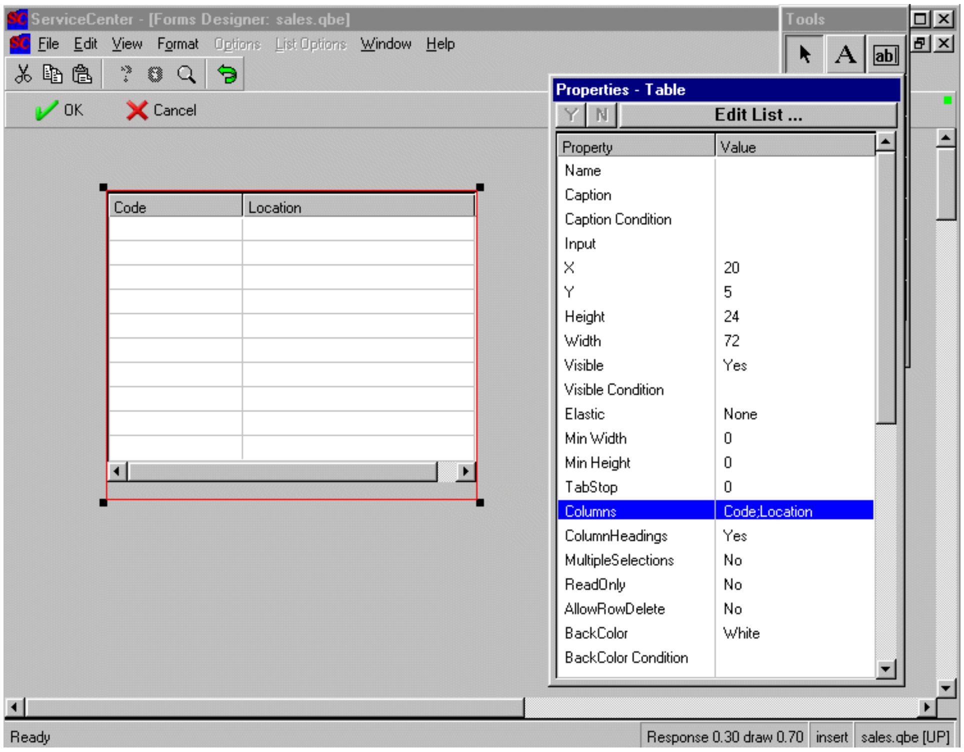
3 Draw a table on the design space

4 Enter **Code** and **Location** in the Columns field of the Table properties window. These are the two dbdict keys for the **sales** file.

5 Select the Code column and enter *code* in the input field.

6 Select the Location column and enter *location* in the input field.

7 Click **OK** to finish.



Property	Value
Name	
Caption	
Caption Condition	
Input	
X	20
Y	5
Height	24
Width	72
Visible	Yes
Visible Condition	
Elastic	None
Min Width	0
Min Height	0
TabStop	0
Columns	Code;Location
ColumnHeadings	Yes
MultipleSelections	No
ReadOnly	No
AllowRowDelete	No
BackColor	White
BackColor Condition	

Figure 17-6: Creating the sales.qbe form

Adding Data to the *sales* File

Now you need to add a few sample records to test the form and provide information for the subformat, which you create in the next section, to retrieve.

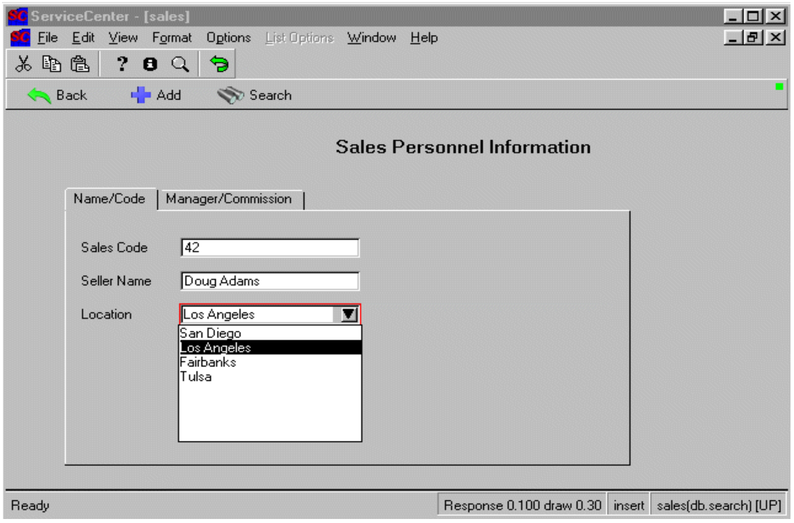
To add data to the sales file:

1 Open the Forms Designer.

2 Enter **sales** in the **Form** field and press Enter.

3 Click the **Options** menu and select ***Database Manager***.

4 Enter at least two new sales personnel records. Make sure to fill in the fields on both tabs.


ServiceCenter - [sales]
File Edit View Format Options List Options Window Help
Back
Add
Search
Sales Personnel Information
Name/Code
Manager/Commission

Sales Code: 42
Seller Name: Doug Adams
Location: Los Angeles

San Diego
Los Angeles
Fairbanks
Tulsa

Ready | Response 0.100 draw 0.30 | insert | sales(db.search) [UP]

Figure 17-7: Sample sales data record

5 Click **OK** to return to **Forms Designer** after you finish adding the data records.

Creating the *sales1* Subformat

1 Return to Forms Designer and the *sales* form.

2 Click the **Options** menu and select **Copy/Rename**.

3 Make a copy of the *sales* form, named *sales1*.

4 Click **OK** to confirm the creation of the duplicate form.

5 Cut both the **Seller Name** label and field and the **Sales Commission %** label and field from the notebook tabs and paste them somewhere on the form.

6 Remove everything else from the form. (title, and the tabs structure and all fields (except **Seller Name** and **Sales Commission %**)

7 You should be left with two fields and two labels, Seller name and Seller Commission.

8 Click **OK** to confirm changes to *sales1*.

The subformat is now completed and, since it is a copied and modified version of the original form (*sales*), it is already associated with the *sales* dbdict file.

To verify that the *sales1* form can bring up sales personnel records:

1 Open **Database Manager**.

2 Open the Sales1 form.

3 Press **Search** and select a record from the QBE list. (If a list does not appear, return to the previous steps and check to make sure you have satisfied each step.)

The first format, *sales*, will be used to input sales personnel data, which will be retrieved through the subformat, *sales1*, virtually joined in the *orders* form.

Creating the Orders form

Design the form to look similar to Figure 17-8 on page 383.

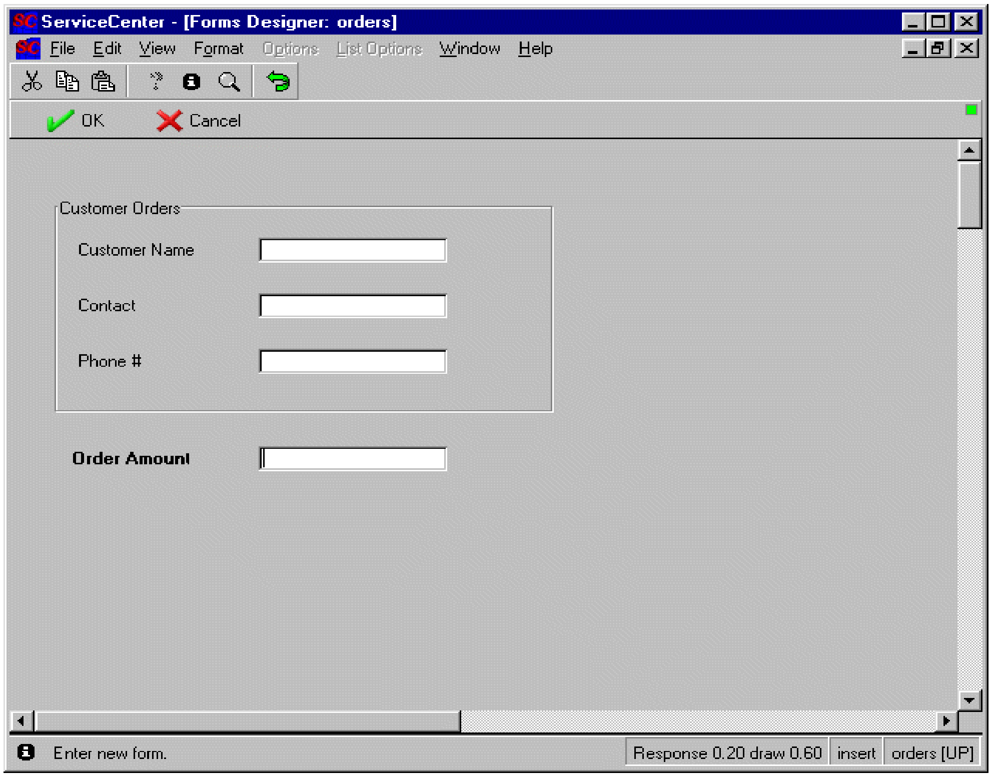
To create the format and file called *orders*:

1 Open Forms Designer.

2 Enter the format name *orders*.

3 Press **Enter or the New** button.

4 If you have another form named *orders*, return to the Forms Designer prompt and rename this new form. Select **No** at the Forms Wizard prompt.



Customer Orders

Customer Name

Contact

Phone #

Order Amount

Enter new form.

Response 0.20 draw 0.60 | insert | orders [UP]

Figure 17-8: Designing the Order Form

5 Select the Frame tool and draw a frame on the screen for the Customer Order fields. Add fields and label the form in a manner similar to the example in Figure 17-8 on page 383. The inputs for the text box fields are:

- customer.name
- contact
- phone.number
- order.amount.

Building The Virtual Join Into The Form

1 Open the orders form (created in the previous section) in the Forms Designer utility.

2 Select the Frame tool and create a frame on the screen where you want the seller data to appear. Give it the caption/title **Seller Data**.

3 Select the Label tool and create a label reading **Seller Code**.

4 Select the Fill box tool and create a fill-able input field. This will allow you to choose from a list of seller codes.

   In the Properties box, enter *code* in the input value and **9** (fill) in the ButtonID field, then click **Y** to confirm the addition.

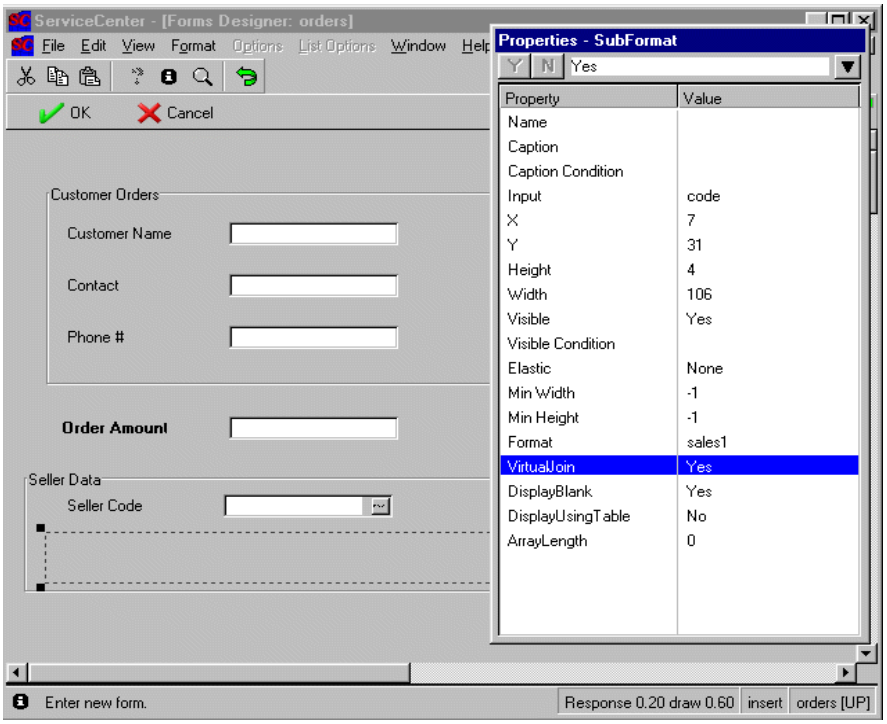
5 Press the **OK** button to save changes to the form.

Subformat Placement

1 Select the Subformat tool.

2 Position the cursor on the screen where you want to begin your virtual join.

3 Create a square where you want the virtually joined subformat to appear.



Property	Value
Name	
Caption	
Caption Condition	
Input	code
X	7
Y	31
Height	4
Width	106
Visible	Yes
Visible Condition	
Elastic	None
Min Width	-1
Min Height	-1
Format	sales1
VirtualJoin	Yes
DisplayBlank	Yes
DisplayUsingTable	No
ArrayLength	0

```
Enter new form.    Response 0.20 draw 0.60    insert    orders [UP]
```

Figure 17-9: Adding the Subformat

4 In the Properties box for the subformat, enter *code* in the **Input** field.

The input field value (`code`) is the same as the previous input field value because a link needs to be established with the target file. The Virtual Join tool uses the link with the *code* key on the *sales* file to pull in the information requested by the **sales1** form, namely values for the *name and commission* fields.

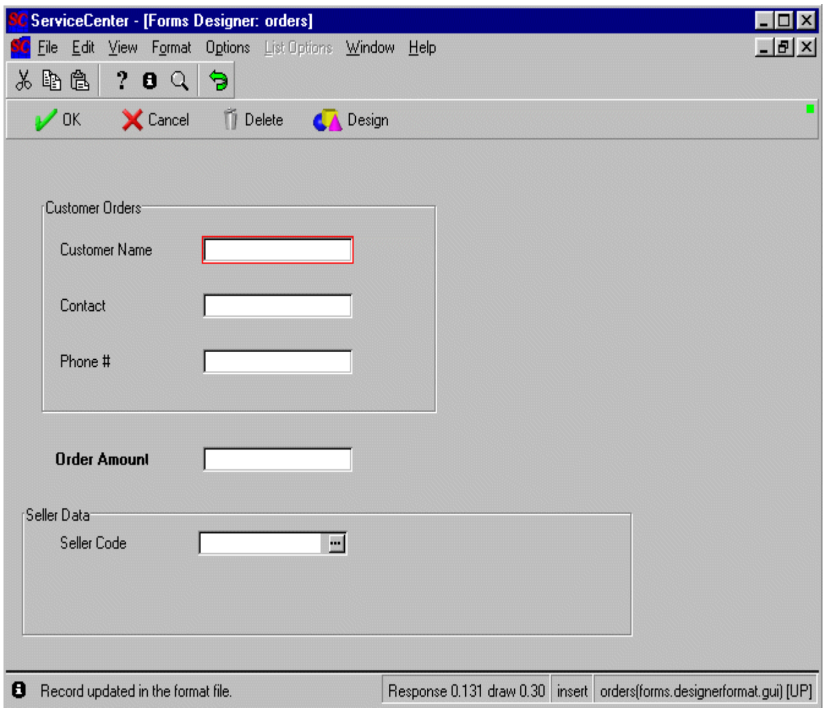
5 Enter *sales1* in the **Format** field to indicate which form should appear in the subformat area.

6 Select Yes in the **Virtual Join** field of the Properties box to activate the virtual join functions on the subformat.

7 Click the **OK** button to save changes to the *orders* form.

Note: The completed form will appear as in Figure 17-10 on page 386 while you are still in Forms Designer. You cannot see the subformat at this point. The subformat data fields will not appear until you have entered data through the Database Manager.

Note you cannot see the Virtual Join until you add data.



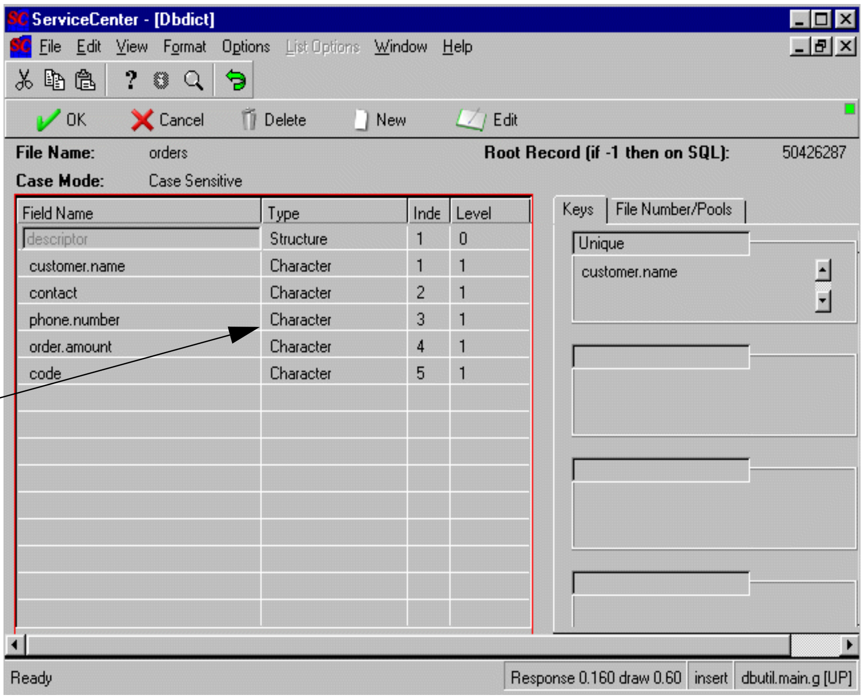
Record updated in the format file. | Response 0.131 draw 0.30 | insert | orders(forms.designerformat.gui) [UP]

Figure 17-10: Order form with Subformat included

8 From the **orders** format, pull down the **Options** menu.

9 Select **Create File.** The Database Dictionary Utility opens.

10 Select **OK** to create the *orders* file.



File Name: orders

Root Record (if -1 then on SQL): 50426287

Case Mode: Case Sensitive

Field Name	Type	Inde	Level
descriptor	Structure	1	0
customer.name	Character	1	1
contact	Character	2	1
phone.number	Character	3	1
order.amount	Character	4	1
code	Character	5	1

Unique

customer.name

Ready | Response 0.160 draw 0.60 | insert | dbutil.main.g [UP]

Note the phone.number field is a character type field.

Figure 17-11: orders file with code field included

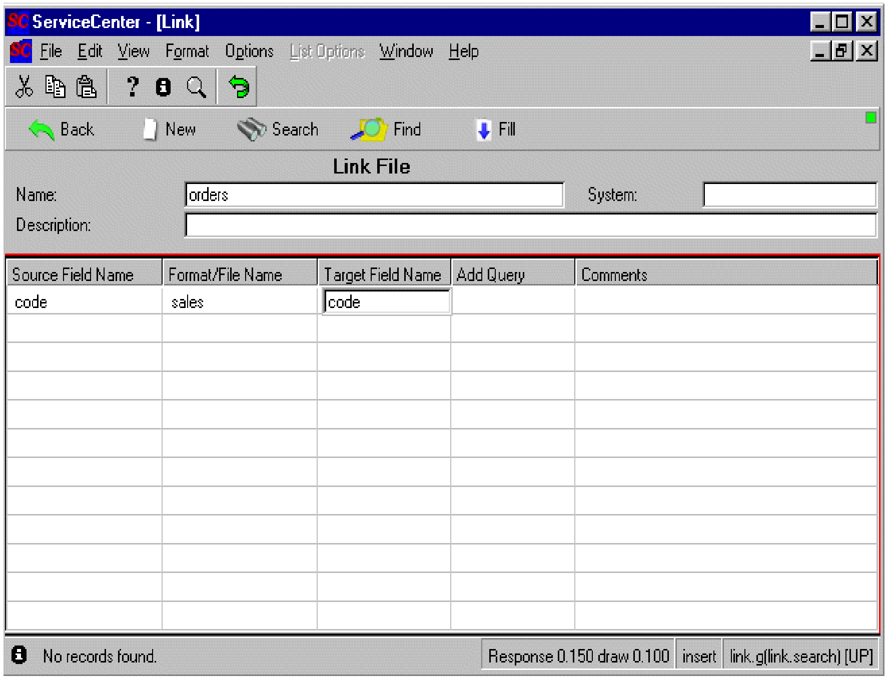
11 In the *orders* file, make sure there is a line for the **code** field at the bottom of the list.

If there is not, click on the **New** button and create a new character field called **code** in the main structure.

Change the phone.number Type to character, if necessary.

12 Click **OK** to save the changes.

13 Click **OK** to leave the Database Dictionary file and return to Forms Designer.



Link File

Name:	orders	System:	
Description:			

Source Field Name	Format/File Name	Target Field Name	Add Query	Comments
code	sales	code		

No records found. Response 0.150 draw 0.100 insert link.g(link.search) [UP]

Figure 17-12: Orders file link record

Building the Link

1 Return to the Forms Designer Utility.

2 Type **orders** in the Form field and click Search or press Enter.
The **orders** form appears.

3 Click **Options** and select **Link**. The link record for the **orders** file appears.

4 Create a link for the **code** field. Link the (Seller Code) **code** field on the **order** file to the (Seller Code) **code** field on the *sales* file.

5 Click **Save**.

6 Click **Back** to exit the link record, returning you to the Forms Designer.

7 Once back in the Forms Design Utility, click **Options**.

8 Select **Database Manager**.

9 You could also just open the *orders* format in the **Database Manager.**

Link Record Field Definitions

The fields for this format are:

Name	ServiceCenter automatically defaults the source filename to the one you specified on the previous screen.
Description	This field is optional. Enter a description of the link, if desired.
Source Field Name	Enter the field name in the source file that relates to a field in the target file.
Format/File Name	Enter the target file name.
Target Field Name	Enter the field name in the target file that relates to a field in the source file.

Important: Virtual Join ignores the query line of a link, returning all information that matches the field value indicated for the query. For instance, with a virtual join you cannot display only open incidents and those closed in the past seven days for a given location, where that location matches the location of the current caller. Rather, all records are displayed for the location.

Using the Virtual Join

1 Create a new *orders* record using the **orders** form.

- Enter the name of a customer in the **Customer Name** field.
- Enter a contact for that customer in the **Contact** field.
- Enter a number for the **Phone** field.
- Enter any number for the **Order Amount** value.

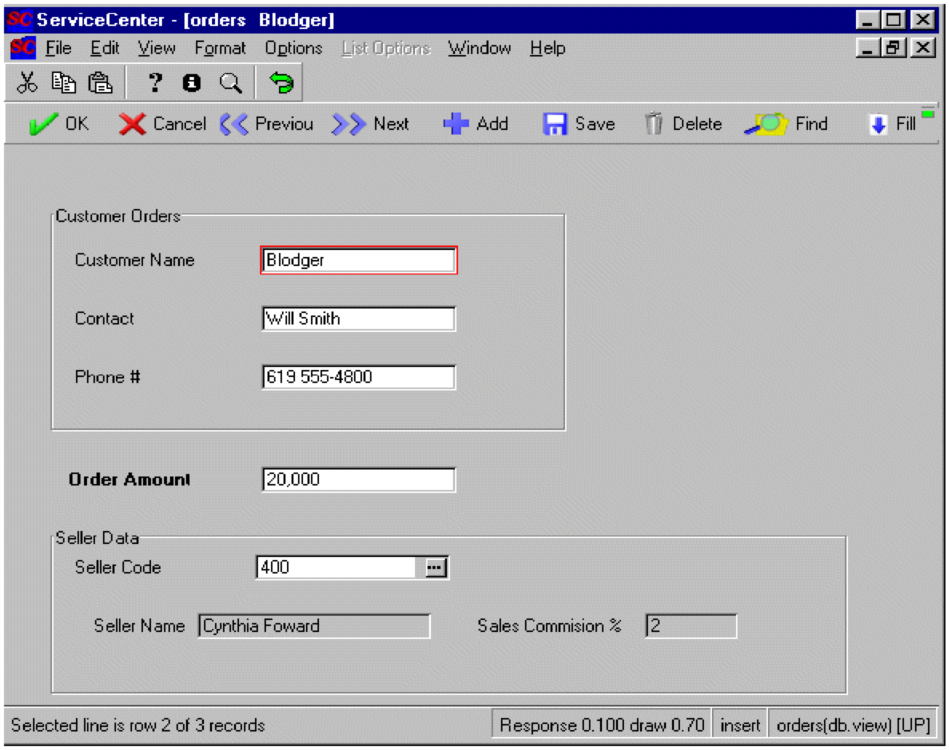
2 In the **Seller Code** field, select the ellipsis (...) button to bring up a QBE list of seller codes.

3 From this list select a valid seller code by clicking on it.

4 Press the **Enter** key.

5 Click the **Add** button to save the record.

The new orders record will appear with the *sales* file data virtually joined in the subformat at the bottom.


ServiceCenter - [orders Blodger]
File Edit View Format Options List Options Window Help
OK Cancel Previou Next Add Save Delete Find Fill

Customer Orders

Customer Name	Blodger
Contact	Will Smith
Phone #	619 555-4800

Order Amount 20,000

Seller Data

Seller Code	400		
Seller Name	Cynthia Foward	Sales Commision %	2

Selected line is row 2 of 3 records | Response 0.100 draw 0.70 | insert | orders(db.view) [UP]

Figure 17-13: Orders form with Virtual Join data

The **Seller name** and **commission** fields are not stored with the record in the *order* file, but are only referenced, and are non-editable. To make changes or updates to the seller data, use the *sales* form.

Note: If you delete this record, **Seller name** and (Seller) **commission** will not be deleted because they are stored in the *sales* file.

Once the record is saved, you will see virtual join information whenever the record is selected, added, or updated.

Index

May 28, 2003