CICS Transaction Server for z/OS
Version 4 Release 2

Business Transaction Services

SC34-7160-00

CICS Transaction Server for z/OS
Version 4 Release 2

Business Transaction Services

SC34-7160-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 239.

This edition applies to Version 4 Release 2 of CICS Transaction Server for z/OS (product number 5655-S97) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1999, 2011.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Dedication vii

Preface ix
What this manual is about ix
Who this manual is for. ix
What you need to know to understand this manual ix
Syntax notation used in this manual ix
A note on terminology x

Changes in CICS Transaction Server for z/OS, Version 4 Release 2 xi

Part 1. Overview of BTS 1

Chapter 1. Why do I need CICS business transaction services? 3
Business transactions and CICS transactions 3
Business transactions 3
CICS transactions. 4
The disadvantages of not using BTS 4

Chapter 2. What are CICS business transaction services? 7
What is a BTS application? 7
Control flow 9
Recovery and restart in BTS 10
Client/server support in BTS 10
Web Interface support in BTS 10
Support for existing code in BTS 10
Sysplex support in BTS 11
Dynamic routing of BTS activities 11
Audit trails 11
Monitoring in BTS 11
The BTS sample application 12

Part 2. BTS application programming 13

Chapter 3. Using the BTS API to write business applications 15
BTS activities and processes 15
Names and identifiers 15
Activation sequences 16
Synchronous and asynchronous activations. . . 17
Lifetime of activities 18
Processing modes 18
User sync points. 20
BTS data-containers 20
BTS timers. 21
BTS events 21
Atomic events 22
Composite events 23
Event pools 24
Deleting events 25
Reattachment events and activity activation . . 26

Chapter 4. The Sale example application 33
Overview of the Sale application 33
Data flows. 33
CICS transactions and programs 34
The initial request 35
Creating the business transaction 36
Starting the business transaction 37
The root activity. 38
Transferring input and output data 44

Chapter 5. Dealing with BTS errors and response codes 49
Checking the response from a synchronous activity 49
Checking the response from an asynchronous activity 51
Getting details of activity ABENDs 51
Trying failed activities again. 52

Chapter 6. Example of running parallel BTS activities 53

Chapter 7. Interacting with BTS processes and activities 61
Acquiring processes and activities 61
Process and Activity rules 62
Using client/server processing 62
The client program 63
The server program 66
Activity processing 69
A user-related example 70
Transferring data to asynchronous activations . . . 83

Chapter 8. Compensation in BTS . . . 85
Implementing compensation. 85
A compensation example 86
Process flow 86
The root activity. 88
Dealing with application locking 96

Chapter 9. Reusing existing 3270 applications in BTS 99
Running a 3270 transaction from BTS. 99
Resource definition 101
Running more complex transactions. 101
Intermediate output messages 101
Conversational transactions. 102
Pseudoconversational transactions 103
Using timers 105

© Copyright IBM Corp. 1999, 2011

Abend processing 105
Sample programs 106

Part 3. Administering CICS business transaction services . . . 109

Chapter 10. System definition for BTS 111
Defining BTS data sets 111
Repository data sets 111
Local request queue data set 113
Naming the routing program 115

Chapter 11. Resource definition for BTS 117
Defining the LRQ file to the CSD 117
Defining repository files to the CSD 118
CEDA DEFINE PROCESSTYPE 119

Chapter 12. Security in BTS 123
Resource security in BTS 123
Process and activity user IDs 123
Attach-time security for processes and activities 124
Command security in BTS 124
CREATE PROCESSTYPE 124
DISCARD PROCESSTYPE 126
INQUIRE PROCESSTYPE 127
SET PROCESSTYPE 130

Chapter 13. BTS operator commands 133
CBAM BTS browser 133
Process-types screen 134
Processes screen 135
Activities screen 136
Activity details screen 137
Containers screen 137
Events screen 138
Timers screen 139
CEMT INQUIRE PROCESSTYPE 140
CEMT INQUIRE TASK 144
CEMT SET PROCESSTYPE 145

Chapter 14. Using BTS in a sysplex 149
The scope of a BTS-set 149
A note about audit logs 149
Dynamic routing of BTS activities 150
Which BTS activities can be dynamically routed? 150
Understanding distributed routing 151
Controlling BTS dynamic routing 154
Creating a BTS-set 154
Naming the routing program 156
Using a CICS distributed routing program . . . 156
How the distributed routing program relates to the dynamic routing program 156
Writing a distributed routing program 157
Using CICSPlex SM with BTS 159
Overview of CICSPlex SM workload management 159
Using CICSPlex SM to route BTS activities . . 159

Part 4. BTS problems and performance 161

Chapter 15. BTS problem determination 163
Dealing with stuck processes 163
Application design errors 163
Restarting stuck processes 164
Dealing with activity abends 166
Dealing with unserviceable requests 167
Unserviceable routing requests 167
Dealing with CICS failures 168

Chapter 16. Creating a BTS audit trail 169
Introduction to BTS audit trails 169
Specifying the level of audit logging. 170
Audit trail constraints when using DASD-only log streams 173
Audit trail examples 173
Process-level auditing 175
Activity-level auditing 175
Full auditing 176
Using the audit trail utility program, DFHATUP 176
Using DFHATUP to read audit logs 176
Sample job stream to run the DFHATUP program 177
Example output from the DFHATUP utility . . 179

Chapter 17. Examining BTS repository records 191
The repository utility program, DFHBARUP . . . 191
Using DFHBARUP 191
Sample job stream to run the DFHBARUP program 192
Example output from the DFHBARUP utility 194

Chapter 18. BTS messages 201

Chapter 19. Setting trace levels for BTS 203
Defining tracing levels at system initialization . . 203
Defining tracing levels when CICS is running . . 204

Chapter 20. Extracting BTS information from a CICS system dump 205

Chapter 21. Tuning BTS performance 207

Part 5. BTS Application Programming Reference 209

Chapter 22. Overview of BTS API commands 211
Process- and activity-related commands 211
Creating, activating, and terminating processes and activities 211

Retrieving information about activities 212
Relating UOWs and activities 212
Container commands 213
Event-related commands 213
Terminology 213
The event-related commands 215
Browsing and inquiry commands. 216
Browsing commands 216
Inquiry commands 217
Tokens and identifiers 217
Commands that take identifiers returned by browse operations 218
Browsing examples 219
BTS system events. 221

Part 6. BTS System Programming Reference 223

Chapter 23. Overview of BTS system programming commands 225
Control commands 225
Inquire command 225

Chapter 24. BTS system programming commands 227
CREATE PROCESSTYPE 227
DISCARD PROCESSTYPE 229
INQUIRE PROCESSTYPE 230
Browsing process-type definitions 232
SET PROCESSTYPE 233

Part 7. Appendixes 235

Appendix. The BTS sample application 237

Notices 239
Programming interface information 240
Trademarks 240

Bibliography 241
CICS books for CICS Transaction Server for z/OS 241
CICSPlex SM books for CICS Transaction Server for z/OS 242
Other CICS publications 242

Accessibility 243

Index 245

Dedication

The CICS® business transaction services features described in this book were inspired and anticipated by a software developer who joined IBM® in April 1974, and who worked in the IBM Hursley Software Development Laboratory. The developers, testers, reviewers, and writer want to dedicate this book to the memory of Peter Lupton.

© Copyright IBM Corp. 1999, 2011

Preface

What this manual is about

This manual documents intended Programming Interfaces that allow the customer to write programs to obtain the services of Version 4 Release 2.

This manual is about CICS business transaction services (BTS) of CICS Transaction Server for z/OS®. CICS business transaction services consist of an application programming interface (API) and support services that allow you to model and manage complex business transactions.

The manual contains introductory, guidance, and reference material.

Who this manual is for

This manual is intended for planners, application programmers, and system programmers.

What you need to know to understand this manual

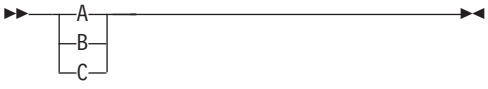
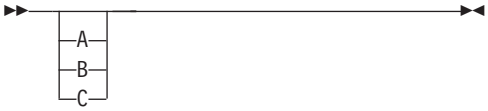
This manual assumes a conceptual understanding of CICS application programming, system definition, resource definition, customization, and security.

Syntax notation used in this manual

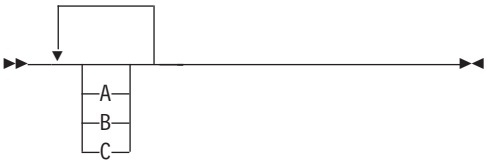
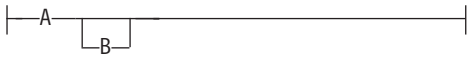
In the reference section of this manual, the syntax of BTS application programming commands is presented in a standard way.

The "EXEC CICS" that always precedes the keyword of each command is not included; nor is the "END_EXEC" statement used in COBOL or the semicolon (;) used in PL/I and C that you must code at the end of each CICS command. In the C language, a null character can be used as an end-of-string marker, but CICS does not recognize this; you must never, therefore, have a comma or period followed by a space (X'40') in the middle of a coding line.

You interpret the syntax by following the arrows from left to right. The conventions are shown in the following table.

Symbol	Action
	A set of alternatives - one of which you ***must*** code.
	A set of alternatives - one of which you ***may*** code.

© Copyright IBM Corp. 1999, 2011

Symbol	Action
	A set of alternatives - any of which you may code.
	Alternatives where **A** is the default.
 Name: 	Use with the named section in place of its name.
Punctuation and uppercase characters	Code exactly as shown.
Lowercase characters	Code your own text, as appropriate (for example, `name`).

$ (the dollar symbol) ...

In the character sets given in this book, the dollar symbol ($) is used as a national currency symbol and is assumed to be assigned the EBCDIC code point X'5B'. In some countries, a different currency symbol - for example, the pound symbol (£) or the yen symbol (¥) - is assigned the same EBCDIC code point. In these countries, the appropriate currency symbol must be used instead of the dollar symbol.

A note on terminology

In this manual, the term “MVS” refers to those services and functions that are provided by z/OS.

Changes in CICS Transaction Server for z/OS, Version 4 Release 2

For information about changes that have been made in this release, please refer to *What's New* in the information center, or the following publications:

- *CICS Transaction Server for z/OS What's New*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.1*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1*

Any technical changes that are made to the text after release are indicated by a vertical bar (|) to the left of each new or changed line of information.

© Copyright IBM Corp. 1999, 2011

Part 1. Overview of BTS

Read this introductory information to learn about about CICS business transaction services (BTS).

To operate BTS, there are no additional requirements beyond the requirements for CICS itself.

Table 1. Concepts road map

If you want to...	Refer to...
Understand the problems that BTS is designed to solve	Chapter 1, "Why do I need CICS business transaction services?," on page 3
Get a high-level view of BTS	Chapter 2, "What are CICS business transaction services?," on page 7

© Copyright IBM Corp. 1999, 2011

Chapter 1. Why do I need CICS business transaction services?

In a complex environment, CICS business transaction services (BTS) bring sophistication to the CICS application programming interface (API), making it better able to model complex business transactions.

CICS provides a robust transaction processing environment. For example, you can create transactions with the ACID properties of atomicity, consistency, isolation, and durability. CICS provides the environment for transactions to continue to run under various conditions.

Much emphasis is placed on the continuous operation and high availability of CICS. Use of sophisticated technologies, such as the Parallel Sysplex®, with resource managers sharing data across the sysplex, has led to improved system availability through the elimination of single points-of-failure. CICS business transaction services complement these technologies.

Related information

ACID properties of transactions

Business transactions and CICS transactions

Business transactions were typically modeled by CICS transactions in ways that now have some shortcomings. CICS business transaction services can overcome these shortcomings.

Business transactions

A *business transaction* is a self-contained business deal. Some business transactions, for example, buying a newspaper, are simple and short-lived. However, many business transactions involve multiple actions that take place over an extended period.

As an example, selling a vacation might involve the travel agent in actions such as:

- Recording customer details
- Booking seats on an aircraft
- Booking a hotel
- Booking a rental car
- Invoicing the customer
- Checking for receipt of payment
- Processing the payment
- Arranging foreign currency.

Both the customer and the travel agent regard the purchase of the vacation as a single business transaction, as indeed it is, because each action only makes sense in the context of the whole. The example illustrates some typical properties of complex business transactions:

- They tend to be made up of a series of logical actions.
- Some actions might be taken days, weeks, or even months after the transaction was started - arranging foreign currency, in this example.

© Copyright IBM Corp. 1999, 2011

- Some of the actions might be optional - not everyone wants to rent a car, for example.
- At any point, an action could fail. For example, a communications failure could mean that it is not possible to book a hotel. In this case, the action must be tried again. Or the customer might fail to meet their final payment; this failure would require a reminder to be sent. If the reminder produces no response, the vacation must be canceled - that is, the actions that have already been taken must be undone.
- Data - for example, a customer account number - must be passed between the individual actions that make up the business transaction.
- Some control logic is required, to "glue" the actions together. For example, there must be logic to deal with the conditional invocation of actions, and with failures.

CICS transactions

The basic building blocks used by CICS applications are the CICS transaction, which in turn is made up of one or more units of work (UOW). Both the transaction and the UOW are typically short-lived.

A UOW is short-lived because it should not hold locks for long periods, thus causing other UOWs to wait on resources and possibly abend. A CICS transaction provides the environment in which its associated UOWs run; for example, the transaction ID, program name, and user ID. A CICS transaction is typically short-lived, because the aim is for it to use CICS resources only while it is doing work; it should not spend long periods waiting for input, for example.

Before CICS Transaction Server for OS/390® Release 3, the largest transaction processing unit that CICS understood was the terminal-related pseudoconversation. A pseudoconversational application appears to a terminal user as a continuous conversation, but consists internally of multiple transactions.

The disadvantages of not using BTS

Typically, application programmers have modeled business transactions using the basic CICS building blocks, transactions, and units of work. However, not using BTS has disadvantages and can impact your application design or recovery and restart.

Here are some of the disadvantages:

Application design

Typically, the individual actions that make up a complex business transaction are mapped on to CICS transactions. Typically, it is not practicable to map a whole business transaction on to a single, long-running CICS transaction, even if the transaction is divided into multiple units of work, because of resource constraints. The locks held by the UOWs would tend to be held for long periods; system performance would suffer, and transaction abends become frequent, due to deadlocks or contention for locked resources.

Mapping each individual action on to a CICS transaction is a more sensible option. However, this approach ignores the overall structure of the business transaction. Typically, the control logic necessary to glue the actions together ends up being spread between the various CICS transactions. Thus, the high-level logic required to control the overall progress of the business transaction and the low-level logic

required to implement a specific business action become blurred. One effect is that the CICS transactions become less easy to reuse, because they are required to do more than implement a particular business action.

An even better option might be to separate the control logic in a single, top-level program that would be invoked again whenever a new stage of the business transaction was ready to run. Each time it was invoked, the top-level program could run a transaction that implements a particular action of the business transaction. This would work similarly to a terminal-related pseudoconversation, in which terminal events cause successive transactions to be invoked. Unfortunately, in CICS releases before CICS Transaction Server for OS/390, Version 1 Release 3 this was not possible. A pseudoconversational application could be used only to simulate a single conversation with a terminal.

Recovery and restart

Long-lived business transactions are much more likely than short-lived transactions to span restarts of CICS, which might or might not be planned. To survive restarts, state data relating to the flow of control of the business transaction must be saved to a recoverable resource. Thought must also be given to how the business transaction is to be restarted after a restart of CICS.

Chapter 2. What are CICS business transaction services?

CICS business transaction services consist of an application programming interface and support services that simplify the development of business transactions. Business transactions are often made up of multiple actions, which might be spread over hours, days, or months.

Terminology: This and the following chapter introduce a number of terms new to CICS. These terms are explained in context, as they occur.

You can use CICS business transaction services to control the execution of complex business transactions. Using BTS, each action that makes up the business transaction is implemented as one or more CICS transactions, using the traditional approach. However, a top-level program is used to control the overall progress of the business transaction. The top-level program manages the inter-relationship, ordering, parallel execution, commit scope, recovery, and restart of the actions that make up the business transaction. This program management brings a number of benefits:

- Management and control are at a business transaction level, as well as at an action level.
- Control logic is separated from business logic. The individual CICS transactions that make up the business transaction no longer need to be concerned with "before and after" actions. This logic separation simplifies the development of such transactions and makes it easier to reuse them.

What is a BTS application?

A BTS application is an application written using the CICS business transaction services API to take advantage of CICS business transaction services.

The components of an application written using the CICS business transaction services API are illustrated, in simplified form, in Figure 1 on page 8. (For brevity, in the rest of this book we refer to an application that uses the CICS business transaction services API as "a BTS application".)

© Copyright IBM Corp. 1999, 2011

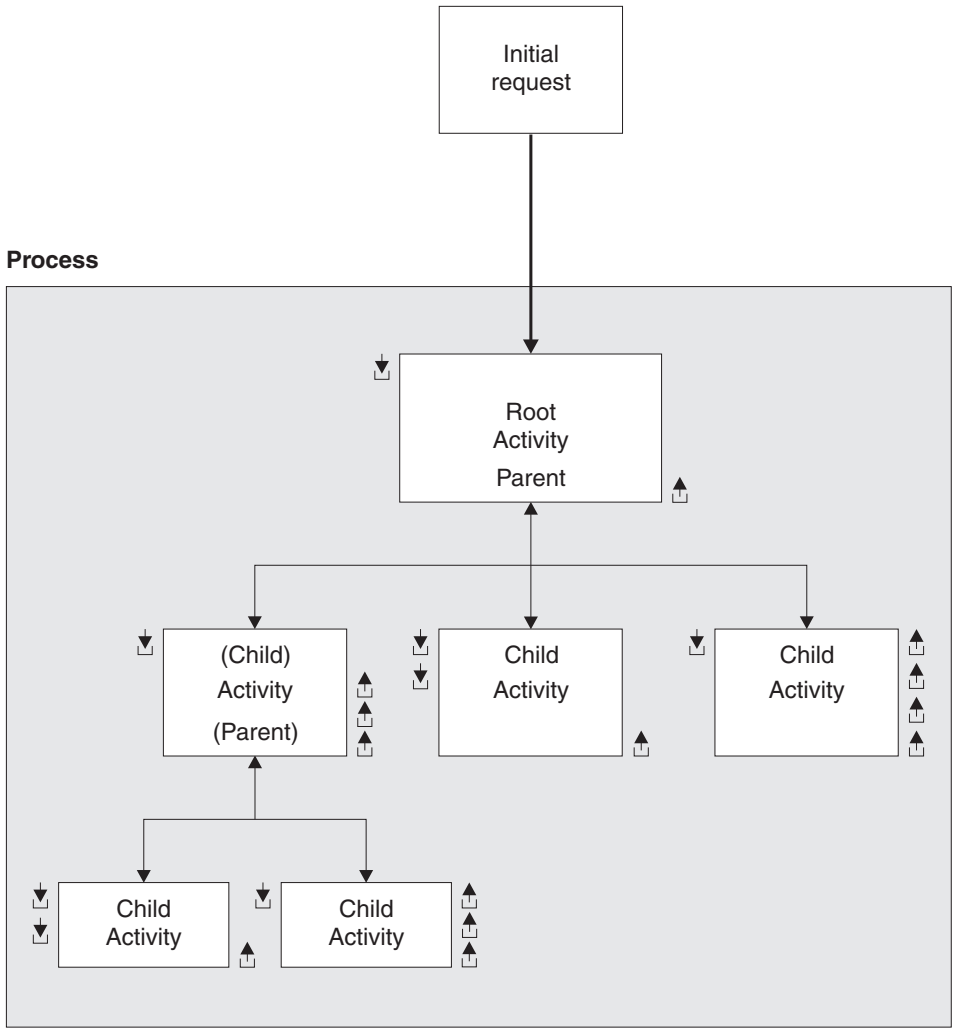


Figure 1. Components of a BTS application

The roles of the components are as follows:

Initial Request
A CICS transaction that starts a CICS business transaction services **process**.

Process
A collection of one or more BTS **activities**. It has a unique name by which it can be referenced and invoked. Typically, a process is an instance of a business transaction.

In the vacation example, an instance of the business transaction might be started to sell Jane Doe a vacation in Florida. To identify this particular transaction as relating to Jane Doe, the process could be given the name of Jane Doe's account number.

Activity
The basic unit of BTS execution. Typically, it represents one of the actions of a business transaction—in the vacation example, renting a car, for instance.

A program that implements an activity differs from a traditional CICS application program only in its being designed to respond to BTS **events**. It can be written in any of the languages supported by CICS.

Activities can be hierarchically organized, in a tree structure. An activity that starts another activity is known as a **parent activity**. An activity that is started by another is known as a **child activity**.

Root activity
The activity at the top of the activity tree - it has no parent activity. A process always contains a root activity. When a process is started, the program that implements its root activity receives control. Typically, a root activity is a parent activity that:

- Creates and controls a set of child activities - that is, it manages their ordering, concurrent execution, and conditional execution
- Controls synchronization, parameter passing and saving of state data.

Data-container
A named area of storage, associated with a particular process or activity, and maintained by BTS. Each process or activity can have any number of data-containers. They are used to hold state data, and inputs and outputs for the activity.

Event (not shown in Figure 1 on page 8)
A BTS event is a means by which CICS business transaction services signal progress in a process. It informs an activity that an action is required or has completed. "Event" is used in its ordinary sense of "something that happens". To define an event recognizable by CICS business transaction services, such a happening is given a name.

Timer (not shown in Figure 1 on page 8)
A BTS object that expires when the system time becomes greater than a specified date and time, or after a specified period has elapsed. Each timer has an event associated with it. The event occurs ("fires") when the timer expires.

You can use a timer to, for example, cause an activity to be invoked at a particular time in the future.

The preceding components are managed by CICS, which:

- Manages many business transactions (processes).
- Records the status of each business transaction.
- Ensures that each activity is invoked at the appropriate times.

The components of a BTS application, and how they relate to each other, are described in more detail in Chapter 3, "Using the BTS API to write business applications," on page 15.

Control flow

Follow the high-level control flow of a typical BTS business transaction.

The high-level control flow of a typical BTS business transaction is as follows:

1. A CICS transaction makes an initial request to start a process.
2. CICS initiates the appropriate root activity.

3. The root activity program, using the BTS API, creates a child activity—or several child activities. It provides the child activity with some input data (by placing the data in a data-container associated with the child), and requests CICS to start the child activity.

   If, as is often the case, the child activity is to run asynchronously with the root activity, the root activity program returns and becomes dormant.
4. The root activity is invoked again when one of its child activities completes. It determines which event caused it to be invoked again - that is, the completion of the activity that it started earlier. It retrieves, from the output data-containers of the completed activity, any return data that the completed activity has placed there.
5. Steps 3 and 4 are repeated until all the child activities that make up the business transaction have completed.
6. CICS terminates the root activity.

   If the business transaction has completed normally, the process is no longer known to CICS.

Recovery and restart in BTS

CICS maintains state data for BTS processes in a recoverable VSAM KSDS. This file can be RLS-enabled. On an emergency restart, CICS automatically restarts any BTS activities that were in flight at the time it failed.

Client/server support in BTS

CICS business transaction services support *client/server* processing. A server process is one that is typically waiting for work. When work arrives, BTS restarts the process, which retrieves any state data that it has previously saved.

Web Interface support in BTS

With the CICS Web Interface, you can run CICS transactions from a web browser. CICS business transaction services extend CICS support for the Internet.

In a typical current scenario, a web-based business transaction might be implemented as a pseudoconversational CICS application. The initial request from the web browser invokes a CICS transaction that does some setup work, returns a page of HTML to the web browser, and ends. Subsequent requests are handled by other CICS transactions (or by further invocations of the same transaction). The CICS application is responsible for maintaining state data between requests.

Using BTS, a web-based business transaction could be implemented as a BTS process. A major advantage of this approach is that state data is now maintained by BTS. This is useful if the business transaction is long-lived.

Support for existing code in BTS

BTS supports the 3270 bridge function. Therefore, BTS applications can be integrated with, and use, existing 3270–based applications.

The 3270 bridge function is described in Bridging to 3270 transactions in the External Interfaces Guide.

Even though BTS activities are not terminal-related, they are never started directly from a terminal; a BTS activity can use a 3270-based program.

Sysplex support in BTS

You can operate BTS in a single CICS region. However, BTS processes are sysplex-enabled, and in a sysplex you can create one or more *BTS-sets*.

A BTS-set is a set of CICS regions across which related BTS processes and activities can run. For example, the activities that constitute a single process might run on several regions.

Dynamic routing of BTS activities

In a BTS-set, you can control the dynamic routing of the CICS transactions that implement your BTS activities across the participating regions.

When an event is signaled, an activity is activated in the most appropriate region in the BTS-set, based on one or more of the following criteria:

- Any workload separation specified by the system programmer
- Any affinities the associated transaction of the activity has with a particular region
- The availability of regions
- The relative workload of regions

You can control the dynamic routing of your BTS activities using one of the following methods:

- Use the CICSPlex® System Manager/ESA (CICSPlex SM) component of CICS Transaction Server for z/OS to:
  - Specify workload separation for your BTS processes.
  - Manage affinities.
  - Control workload routing of the transactions that implement BTS activities.
- Write a CICS distributed routing program.

Dynamic routing of BTS activities is described in Chapter 14, “Using BTS in a sysplex,” on page 149.

Audit trails

You can create an audit trail for the BTS processes and activities that run in your CICS regions. By doing so, you can, for example, track the progress of a complex business transaction across the sysplex.

The CICS code contains BTS audit points in much the same way as it contains trace points. However, because in a sysplex environment different parts of a process might execute on different regions, each audit record contains system, date, and time information. By sharing log streams across regions, you can gather audit information from different regions in the same log.

Monitoring in BTS

CICS maintains monitoring information for both processes and activities. You can request information about use of resources of a business transaction without knowing the identifiers of all its constituent CICS transactions. Information is now available at the business transaction level, as well as at the CICS transaction level.

BTS monitoring is described in Chapter 21, “Tuning BTS performance,” on page 207.

The BTS sample application

CICS supplies a sample BTS application and fragments of example code. The sample is a basic sales application, consisting of order, credit check, stock check, delivery note, invoice, and payment-reminder activities.

For more information, see “The BTS sample application,” on page 237.

Part 2. BTS application programming

BTS provides an application programming interface (API) that you can use to run business transactions in CICS.

Table 2. Application programming road map

If you want to...	Refer to...
Learn about the components of a BTS application	Chapter 3, "Using the BTS API to write business applications," on page 15
See an example of a basic BTS application	Chapter 4, "The Sale example application," on page 33
Learn how to check the state of BTS activities	Chapter 5, "Dealing with BTS errors and response codes," on page 49
Learn how to run several activities simultaneously	Chapter 6, "Example of running parallel BTS activities," on page 53
Learn how to access BTS activities from outside BTS	Chapter 7, "Interacting with BTS processes and activities," on page 61
Learn how to back out BTS activities	Chapter 8, "Compensation in BTS," on page 85
Learn how to use existing code in BTS applications	Chapter 9, "Reusing existing 3270 applications in BTS," on page 99

© Copyright IBM Corp. 1999, 2011

Chapter 3. Using the BTS API to write business applications

This section provides a detailed description of the BTS application components, and explains how you can use them.

The section contains:

- "BTS activities and processes"
- "BTS data-containers" on page 20
- "BTS timers" on page 21
- "BTS events" on page 21.

BTS activities and processes

An *activity* is the BTS unit of execution, which holds the environment for an instance of the BTS equivalent of program execution. A process is the biggest entity recognized by BTS and it consists of a collection of one or more activities. An *activity* is the BTS unit of execution. It holds the environment for an instance of the BTS equivalent of program execution.

The state of a BTS activity is stored on disk and reinstantiated in memory as required. Typically, it represents one of the actions of a business transaction.

Activities can be hierarchically organized, in a tree structure that may be several layers deep. The activity at the top of the hierarchy is called the **root activity**. An activity that starts another activity is known as a *parent activity*. An activity that is started by another is known as a *child activity*. For example, if activity A starts activity B, B is a child of A; A is the parent of B. Notice that—with the exception of the root activity, which has no parent—an activity can be both a parent and a child.

A *process* is the biggest entity recognized by BTS. It consists of a collection of one or more activities. It always contains a root activity. When a process is run, the program that implements its root activity receives control. Typically, a process is an instance of a business transaction.

Processes can be categorized, using the PROCESSTYPE option of the DEFINE PROCESS command. All the activities in a process inherit the same PROCESSTYPE attribute. Categorizing processes makes it easier to find a particular process—the BTS browsing commands allow filtering by process-type.

Names and identifiers

You can use a program to define a process. The program gives the process a name, known as its *process name*, which is used to reference the process from outside the BTS system.

This user-assigned name, which can be up to 36 characters long, must be unique within the process-type to which the process belongs.

Similarly, when an activity program defines a child activity, it gives the child a name (its *activity name*), which it uses to reference the child. This user-assigned name, which can be up to 16 characters long, only needs to be unique within the

© Copyright IBM Corp. 1999, 2011

set of child activities defined by the parent. For example, it is perfectly valid for several activities within the same process to each define a child called *Invoice*.

Note: A root activity always has the CICS-assigned name DFHROOT.

Besides its name, each activity has a CICS-assigned **activity identifier**. An activity identifier, which is 52 characters long, is a means of uniquely referring to an activity-instance. It is guaranteed to be unique across the sysplex, and its lifetime is the same as the activity it refers to. Activity identifiers are frequently used as arguments on inquiry and browsing commands. Only its parent can refer to a child activity by name; other programs can access the activity with its identifier.

Activation sequences

To complete all its work, an activity might need to run as a sequence of separate processing steps, or *activations*. For example, a parent activity typically needs to run for a while, stop temporarily, and then continue execution when one of its children has completed.

Each activation is triggered by a BTS *event*, and consists of a single transaction. First activation of the activity is triggered by the system event DFHINITIAL, supplied by BTS after the first RUN or LINK command is issued against the activity. (In the case of a root activity, DFHINITIAL occurs after the first **RUN** or **LINK** command is issued against the process. It is possible to issue multiple **RUN** or **LINK** commands against a process - see "Using client/server processing" on page 62.) When the last activation ends, the *activity completion event* is fired, which might, in turn, trigger another activation of the activity. See "BTS events" on page 21.

Figure 2 shows a BTS activity being reattached in a series of activations.

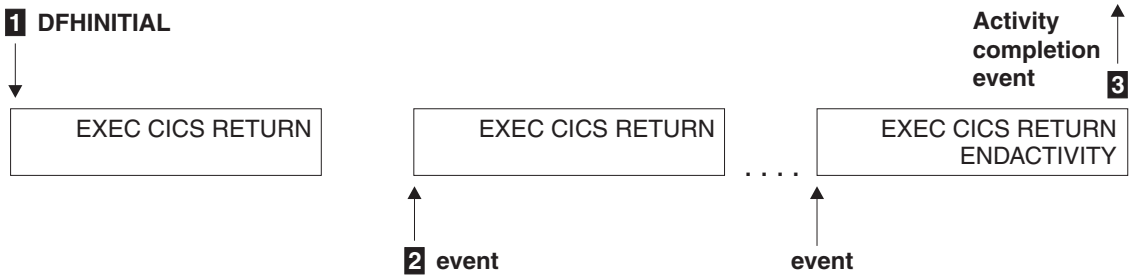


Figure 2. A sequence of activations

1 The first event that "wakes up" the activity is DFHINITIAL. The activity determines that the event which caused it to be activated was DFHINITIAL and therefore performs its first processing step. Typically, this involves defining further events for which it might be activated. The activity program issues an **EXEC CICS RETURN** command to relinquish control. The activity "sleeps".

2 The next event occurs and "wakes up" the activity. The activity program determines which event caused it to be activated and performs the processing step appropriate for that event. It issues an **EXEC CICS RETURN** command to relinquish control.

3 Eventually, no more processing steps are necessary. To confirm that its current activation is the last, and that it is not to be reactivated for any

future events, the activity program issues an **EXEC CICS RETURN ENDACTIVITY** command. The activity completion event is fired.

Note: Root activities do not have completion events.

Figure 3 is a comparison between a terminal-related pseudoconversation and a BTS activity that is activated multiple times.

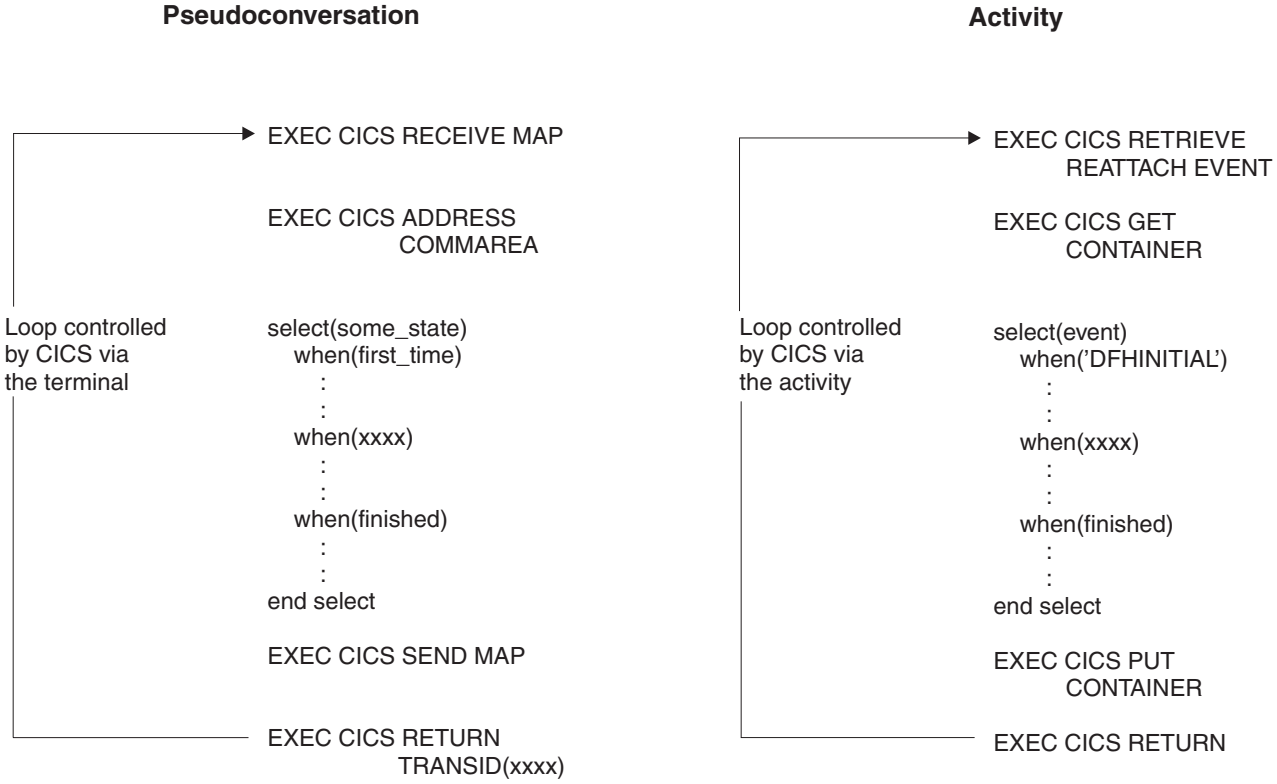


Figure 3. Comparison between a terminal-related pseudoconversation and a BTS activity that is activated multiple times

Note: The **RETRIEVE REATTACH EVENT** command issued by the activity retrieves the name of an event that caused the activity to be reactivated. The **GET** and **PUT CONTAINER** commands retrieve and store input and output data.

Synchronous and asynchronous activations

You can cause an activity or process to be activated in one of two ways, synchronously or asynchronously.

Synchronously
The activity or process is executed synchronously with the requestor. Exactly how it is run varies, depending on which command is used to activate it:

LINK
The activity is included as part of the current unit of work; all locks and resources are shared with the requestor. The activity runs with the transaction attributes of the requestor; any transaction attributes (transaction ID or user ID) specified on its resource definition are ignored. In other words, there is no **context-switch**.

RUN SYNCHRONOUS
The activity is run in a separate unit of work from the unit of work of the

requestor, and with the transaction attributes (transaction ID or user ID) specified on its resource definition. In other words, a **context-switch** takes place.

The two units of work are linked; if the requestor backs out, the activity is backed out also.

Asynchronously
The activity or process is executed asynchronously with the requestor, following a RUN ASYNCHRONOUS command.

The activity is run in a separate unit of work from the unit of work of the requestor, and with the transaction attributes (transaction ID or user ID) specified on its resource definition - that is, a context-switch takes place.

Checking the response from a child activity

After a parent has requested a child activity to be run, it must check the response from the child by issuing a **CHECK ACTIVITY** command to find out whether the child activity succeeded.

The response to the request to run the activity does not contain any information about the success or failure of the child activity itself, only about the success or failure of the request to run it.

Typically, in the case of a synchronous child activity, the **CHECK ACTIVITY** command is issued immediately after the **RUN** command. For an asynchronous child activity, it could be issued:

- When the parent is reattached due to the completion event of the child firing. See "Reattachment events and activity activation" on page 26.
- When the parent is reattached due to the expiry of a timer.

If the child activity needs more than one processing step (transaction) to complete its work, on return from its first activation it is not complete. The **CHECK ACTIVITY** command returns the current completion status.

Following the execution of a **CHECK ACTIVITY** command issued by its parent, if the child activity has completed, its completion event, and its name, is deleted by CICS. The event cannot be deleted in any other way, because it *is* the completion of the activity.

For further information about the uses of the **CHECK ACTIVITY** command, see Chapter 5, "Dealing with BTS errors and response codes," on page 49.

Lifetime of activities

A child activity is created when its parent issues a **DEFINE ACTIVITY** command. It is deleted automatically by CICS, either when its parent completes or if the parent issues a **DELETE ACTIVITY** command against it.

Note: It is not typically necessary to delete an activity explicitly.

Processing modes

Processing states or modes for activities can be active, canceling, complete, dormant, or initial. These modes describe the current state of an activity.

An activity is always in one of the following processing states or **modes**:

ACTIVE
An activation of the activity is running.

CANCELLING
CICS is waiting to cancel the activity. A **CANCEL ACTIVITY** command has been issued, but CICS cannot cancel the activity immediately because one or more of the descendants of the activity are inaccessible. For example, if one of the children of the activity holds a retained lock.

COMPLETE
The activity has completed, either successfully or unsuccessfully. The value returned on the COMPSTATUS option of a **CHECK ACTIVITY** command tells you how it completed.

DORMANT
The activity is waiting for an event to fire its next activation.

INITIAL
No RUN or LINK command has yet been issued against the activity; or the activity has been reset to its initial state with a **RESET ACTIVITY** command.

Figure 4 is a (slightly simplified) view of how the processing modes relate to each other. The BTS commands that cause an activity to move from one mode to another are shown in uppercase.

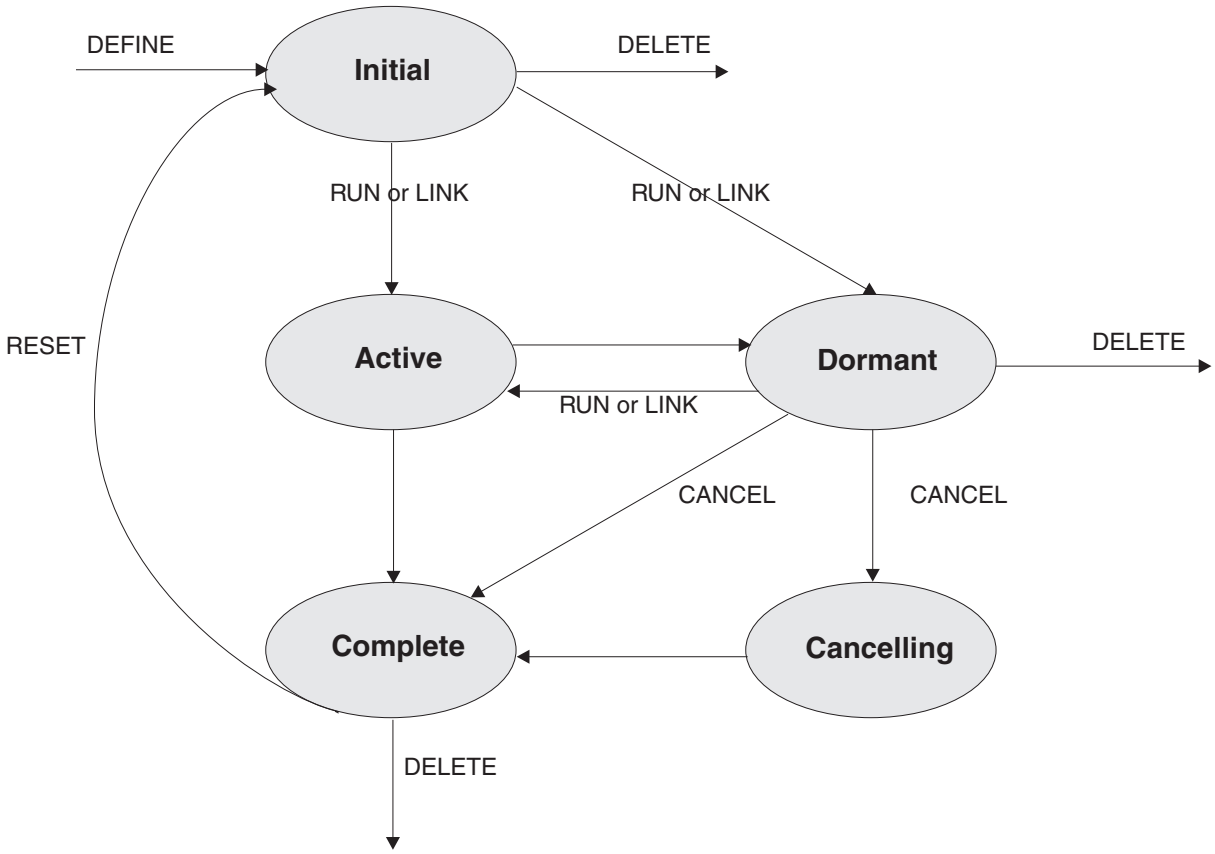


Figure 4. Activity mode transitions. The words in uppercase are the commands which cause the transitions.

Use the **CHECK ACTIVITY** or **INQUIRE ACTIVITYID** command to determine the current mode of an activity.

User sync points

These programs might issue user sync points: a *top-level* transaction that defines and runs a BTS process, a program that runs as the activation of a child activity, or a program executing outside the BTS environment that acquires a process or activity with an **ACQUIRE** command

A program that is running as the activation of a BTS *process* cannot issue user sync points (**EXEC CICS SYNCPOINT** commands).

For more information about acquiring a process or activity with an **ACQUIRE** command, see "Acquiring processes and activities" on page 61.

BTS data-containers

A *data-container* is a named area of storage, maintained by BTS. Because data-containers are preserved across multiple activations of the activity, they can be used to hold state data or inputs and outputs for the activity. They are recoverable resources, written to disk as necessary, and restored at system restart.

Each data-container is associated with an activity or process. It is identified by its name and by the activity for which it is a container. An activity can have any number of containers, as long as they all have different names within the scope of the activity. For example, several activities can each have containers named "Input", "Output", and "State".

Data-containers of an activity serve as its working storage. They can be read and updated by the activity itself, by the parent of the activity, or by a program that has "acquired" the activity.

Just like an activity, a process might have a set of data-containers associated with it. These are called *process containers*: every activity in the process can access them, but only the root activity, or a program that has "acquired" the process, can update them.

Remember: A process's containers are not the same as its root activity's containers.

Before running a process, the program that creates it can:

- Create and set the process containers
- Create and set the containers of the root activity.

Alternatively, the root activity can create and set the process containers.

Lifetime of data-containers

Data-containers of the activity have the same lifetime as the activity itself. They are only destroyed when the activity itself is destroyed. While a child activity exists, its data-containers are always accessible to its parent, whatever processing mode the child is in (including complete).

If you issue a DELETE ACTIVITY command against an activity, bear in mind that you destroy the containers of the activity. It is typically best to allow activities to be deleted automatically by CICS. For child activities, this happens when the parent of the activity completes. At this stage, the parent no longer needs access to its children's containers. If the parent is reset and run again, it recreates its child activities.

Related tasks

"Acquiring processes and activities" on page 61
Before a program that runs outside a process can activate an activity in the process, it uses the **ACQUIRE** command to acquire access to the activity.

BTS timers

A *timer* is a BTS object that expires when the system time becomes greater than a specified date and time, or after a specified period has elapsed. You manage timers with TIMER commands; for example, to cause an activity to be activated at a particular time.

Note: A timer that specifies a date and time that has already passed expires immediately. Similarly, if the requested interval is zero, the timer expires immediately.

To define a timer, use the DEFINE TIMER command. When you define a timer, a *timer event* is automatically associated with it. For more information, see "Atomic events" on page 22.

To force a timer to expire before its specified time, use the FORCE TIMER command.

To check whether a timer has expired and, if it has, whether it expired normally or following a FORCE TIMER command, use the CHECK TIMER command.

Timer management tips

- If a piece of processing (for example, *At midnight on 31st December, prepare an annual customer statement*) could result in many timers being set to expire at the same time, put the timers in groups and stagger the expiry times. Staggering the expiry times spreads the load on CICS and improves performance.
- If you shut down CICS at regular times, and know beforehand that at certain times it is unavailable, try not to set many timers to expire at these times. The timer events all fire when CICS is restarted, which could affect CICS startup performance.

BTS events

CICS business transaction services uses BTS events to signal progress in a process. An *event* informs an activity that an action is required or has completed.

"Event" is used in its ordinary sense of "something that happens". To define an event recognizable by CICS business transaction services, such a happening is given a name. An activity program uses such commands as **DEFINE INPUT EVENT**, **DEFINE TIMER**, and the EVENT option of **DEFINE ACTIVITY** to name events about which it wants to be informed.

Named events have Boolean values - FIRED or NOTFIRED. When first defined, an event has the NOTFIRED value. When an event occurs it is said to *fire* (that is, to make the transition from NOTFIRED to FIRED). An activity can, for example:

- Discover the event (or events) whose firing caused it to be reattached (**RETRIEVE REATTACH EVENT**)
- Test whether an event has fired (**TEST EVENT**).

BTS events can be *atomic* or *composite*.

Atomic events

An *atomic event* is a single, low-level occurrence, which might happen under the control of BTS or outside the control of BTS.

There are four types of atomic event:

- Input events
- Activity completion events
- Timer events
- System events.

Atomic events are the basic components out of which composite events can be constructed. For more information, see "Composite events" on page 23.

Input events

Input events inform activities why they are being run. A RUN or LINK ACTIVITY command delivers an input event to an activity, and thus activates the activity. The INPUTEVENT option on the command *names* the input event and thus defines it to the requestor.

The first time an activity is run, CICS always sends it the DFHINITIAL **system event**. DFHINITIAL tells the activity to perform its initial housekeeping. Typically, this involves defining further events for which it might be activated.

An activity must use the RETRIEVE REATTACH EVENT command to discover the event or events that caused it to be activated. On any activation (but typically on its first, when it is started with DFHINITIAL), it might use the DEFINE INPUT EVENT command to define some input events for which it can be activated after.

Note: The RUN command can also be used to activate a *process* multiple times, delivering a different input event on each activation. This is not discussed here - see "Using client/server processing" on page 62.

Activity completion events

The completion of a child activity, but not a root activity, causes the *activity completion event* to fire. The EVENT option on the DEFINE ACTIVITY command names the activity completion event and thus defines it. If EVENT is not specified, the completion event is given the same name as the activity itself.

Timer events

When you define a timer, a *timer event* is automatically associated with it. When the timer expires, its associated event starts.

Note: If you do not specify the EVENT option of the DEFINE TIMER command, the timer event is given the same name as the timer itself.

System events

BTS *system events* are a special input event defined by BTS, unlike *user-defined events*, which are defined by the BTS application programmer.

All the other types of event described in this chapter, including composite events, are referred to as *user-defined events*, because they are defined by the BTS application programmer, using commands such as DEFINE INPUT EVENT, DEFINE TIMER, DEFINE COMPOSITE EVENT, and the EVENT option of DEFINE ACTIVITY.

There is only one type of BTS system event - DFHINITIAL. For more information, see "BTS system events" on page 221.

System events cannot be included in composite events.

Composite events

A *composite event* is a high-level event, formed from zero or more user-defined (that is, nonsystem) atomic events. When included in a composite event, an atomic event is known as a *sub-event*.

The **DEFINE COMPOSITE EVENT** command defines a *predicate*, which is a logical expression typically involving sub-events. At all times, the composite event's fire status reflects the value of the predicate. When the predicate becomes true, the composite event fires; when it becomes false, the composite's fire status reverts to NOTFIRED.

The logical operator that is applied to the composite event's predicate is one of the Boolean operators AND or OR.

When first defined, a composite event contains between zero and eight sub-events. (A composite event that contains zero sub-events is said to be "empty".) The ADD SUBEVENT command can be used to add further sub-events to the composite event.

- A composite event that uses the OR Boolean operator fires when any of its sub-events fires.
- A composite event that uses the AND operator fires when all of its sub-events have fired, or when it is empty.

Figure 5 shows four composite events, C1 through C4. Each composite event contains two sub-events. C1 and C2 use the OR Boolean operator. C3 and C4 use the AND operator. The shaded circles indicate the events that have fired.

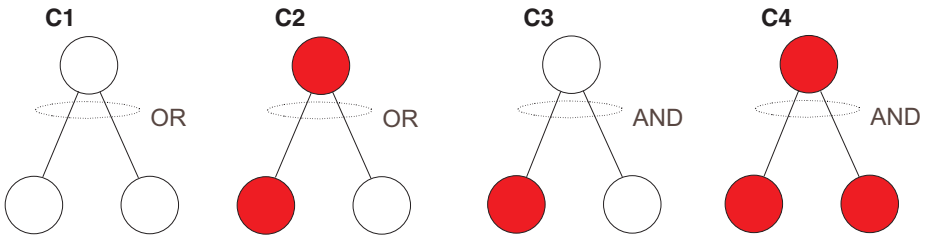


Figure 5. Composite events. An OR composite event fires when any of its sub-events fires. An AND composite event fires when all of its sub-events have fired.

Note:

1. An empty composite event that uses the AND operator is always true (FIRED). An empty composite event that uses the OR operator is always false (NOTFIRED).
2. The following *cannot* be added as sub-events to a composite event:

- Composite events
- Sub-events of other composite events
- System events
- Input events, if the composite uses the AND operator.

The subevent queue

The names of subevents that fire are placed on the subevent queue of the composite event, from where they can be retrieved by issuing one or more **RETRIEVE SUBEVENT** commands.

Each composite event has a subevent queue associated with it. The subevent queue might be empty or contain only the names of those subevents that have fired and not been retrieved.

Figure 6 shows all the events that are recognized by a particular activity. Among them are two composite events, C1 and C2. The sub-event queue for C1 contains the name T1. The sub-event queue for C2 contains the names S1 and S3.

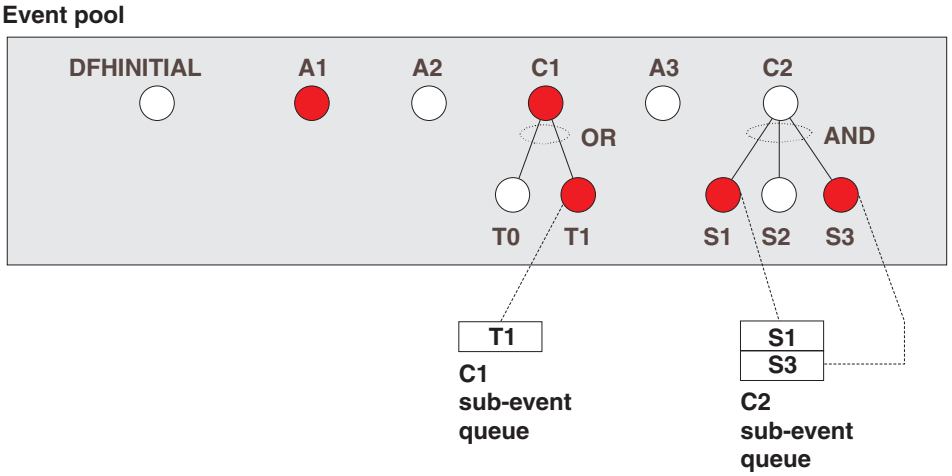


Figure 6. Subevent queues. The subevent queue for composite event C1 contains the name T1. The sub-event queue for composite event C2 contains the names S1 and S3.

Event pools

Events are defined in *event pools*. Each activity has an event pool, which contains the set of events that it recognizes.

The events that an activity recognizes are:

1. Events that have been defined to it with:
   - DEFINE COMPOSITE EVENT
   - DEFINE INPUT EVENT
   - DEFINE TIMER
   - The EVENT option of the DEFINE ACTIVITY command.
2. System events.

An event pool of an activity is initialized when the activity is created, and deleted when the activity is deleted. All the event-related commands described in "Event-related commands" on page 213, except FORCE TIMER, operate on the event pool associated with the *current* activity.

Figure 7 shows an event pool of an activity.

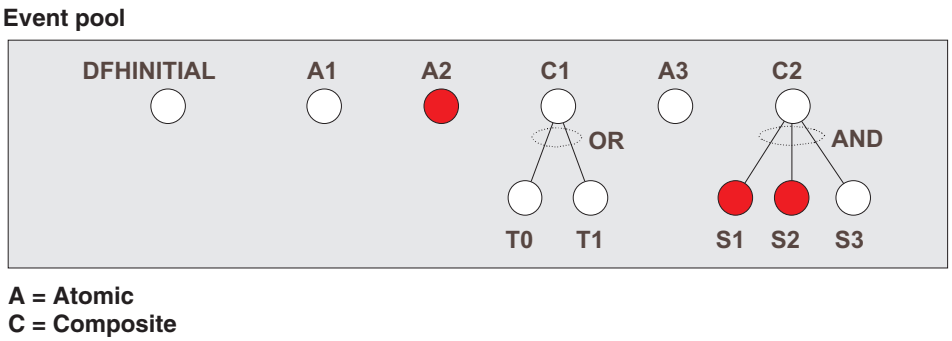


Figure 7. An event pool

Deleting events

You can delete an event by discarding both the event and its name. If the event is a subevent, the value of the composite event is that of its predicate, after the subevent is removed from the Boolean expression of the predicate.

About this task

The command you use to delete an event depends on the type of event to be deleted:

- To delete an input event explicitly, use the **DELETE EVENT** command.
- To delete a composite event explicitly, use the **DELETE EVENT** command. Deleting a composite event does not delete the subevents of the composite event.
- An activity completion event is implicitly deleted when a response from the completed activity has been acknowledged by a **CHECK ACTIVITY** command issued by the parent of the activity; or when a **DELETE ACTIVITY** command is issued.
- A timer event is implicitly deleted if its associated timer has expired and a **CHECK TIMER** command is issued by the activity that owns it; or when a **DELETE TIMER** command is issued.
- You cannot delete system events.
- If an activity program issues a **RETURN ENDACTIVITY** command, CICS automatically deletes all user events other than activity completion events, which must always be deleted with **CHECK ACTIVITY** or **DELETE ACTIVITY** commands in the event pool of the activity. See "Using the ENDACTIVITY option of the RETURN command" on page 31.

Table 3 summarizes the commands that can be used to delete each type of event.

Table 3. Commands used to delete events

Event type	Deletion commands
Activity completion	1. **CHECK ACTIVITY** (if the activity has completed) 2. **DELETE ACTIVITY**
Composite	1. **DELETE EVENT** 2. **RETURN ENDACTIVITY**

Table 3. Commands used to delete events (continued)

Event type	Deletion commands
Input	1. **DELETE EVENT** 2. **RETURN ENDACTIVITY**
System	Cannot be deleted
Timer	1. **CHECK TIMER** (if the timer has expired) 2. **DELETE TIMER** 3. **RETURN ENDACTIVITY**

Before an activity can complete normally, it must have deleted all the activity completion events in its event pool. This means that it must have dealt with all its child activities, see "Activity completion" on page 30.

Reattachment events and activity activation

An activity is reattached (reactivated) on the firing of any event, other than a subevent, that is in its event pool.

In other words, an activity is reattached when either of the following types of event occurs:

- A user-event that has been defined to the activity and not included in a composite event. The user-event might be:
  - An input event
  - The completion event for a child activity
  - A timer event
  - A composite event.
- A system event.

An event that causes an activity to be reactivated is known as a **reattachment event**.

Note: The firing of a subevent never directly causes an activity to be reattached—it is the firing of the associated composite event that does so. Therefore, a subevent can never be a reattachment event.

Handling reattachment events

When an activity is reattached, it uses the RETRIEVE REATTACH EVENT command to discover the event that caused reattachment. The names of reattachment events are placed on the reattachment queue, and from there the activity deals with them.

About this task

If the event that caused it to be reattached is composite, the activity might also need to issue one or more RETRIEVE SUBEVENT commands to discover the subevent or subevents that fired.

At times reattachment might occur because of the firing of more than one event. When reattachment events occur, their names are placed on a queue—the **reattachment queue**—from where they can be retrieved with RETRIEVE REATTACH EVENT commands. Each activity has a reattachment queue, which:

- May be empty
- Contains only the names of those reattachment events that have fired and not been retrieved.

Often, when an activity is reattached there is only one event on the reattachment queue, because activities are reactivated as each reattachment event occurs. However, it is possible for the reattachment queue to contain more than one event—if, for example, the activity has previously been suspended, and reattachment events occurred while it was suspended; or if two or more timer events fire simultaneously.

Figure 8 shows the event pool and reattachment queue for a particular activity. The reattachment queue contains the names A1 and C1.

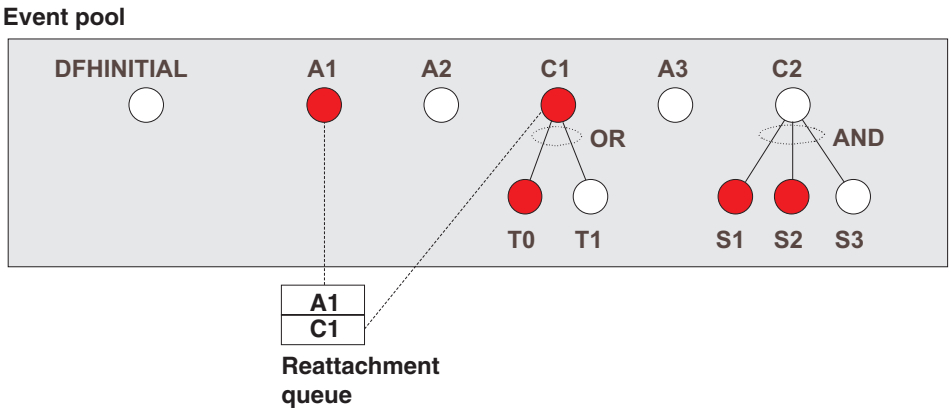


Figure 8. A reattachment queue. The queue contains the names A1 and C1.

Important: With one exception, each time it is activated an activity must deal with at least one reattachment event. That is, it must issue at least one RETRIEVE REATTACH EVENT command, and (if this is not done automatically by CICS) reset the fire status of the retrieved event to NOTFIRED—see "Resetting and deleting reattachment events." Failure to do so results in the activity abending, because it has not progressed, it has not reset any reattachment events and is therefore in danger of getting into an unintentional loop.

The one exception to this general rule is if the activity program issues a RETURN ENDACTIVITY command—in which case, it is not required to have issued a RETRIEVE REATTACH EVENT command in the current activation.

If there are multiple events on its reattachment queue, an activity can, by issuing multiple RETRIEVE REATTACH EVENT commands, deal with several or all of them in a single activation. Alternatively, it can deal with them singly, by issuing only one RETRIEVE REATTACH EVENT command per activation and returning; it is then reactivated to deal with the next event on its reattachment queue. Which approach you choose is a matter of program design. Bear in mind, if you deal with several reattachment events in the same activation, that a sync point does not occur until the activation returns.

Resetting and deleting reattachment events:

Retrieving an atomic event (but not a composite event) from the reattachment queue automatically causes the event's fire status to be reset to NOTFIRED. You can follow this sequence that an activity might use to handle reattachment events.

About this task

Retrieving a composite event from the reattachment queue does *not* reset the event's fire status to NOTFIRED, because a composite event is only reset when its predicate becomes false. Thus, if an activity program retrieves a composite event, it should reset the fire status of the sub-event or sub-events that have fired. (One way of doing this is to issue one or more RETRIEVE SUBEVENT commands.) This in turn causes the fire status of the composite event to be re-evaluated.

If the activity was reattached because of the completion of one of its children, it should issue a CHECK ACTIVITY command to check whether the child activity completed normally. On return from the CHECK ACTIVITY command, CICS deletes the activity completion event from the parent's event pool.

If the activity was reattached because of the expiry of a timer, it can issue a CHECK TIMER command to check whether the timer expired normally. On return from the CHECK TIMER command, CICS deletes the timer event from the activity's event pool.

If the activity wants to delete input and composite events from its event pool, it can issue DELETE EVENT commands. Alternatively, it can rely on a RETURN ENDACTIVITY command, issued on its final activation, to delete them.

Figure 9 on page 29 shows a typical sequence that an activity might use to handle reattachment events. The “Handle atomic event” box is expanded in Figure 10 on page 30.

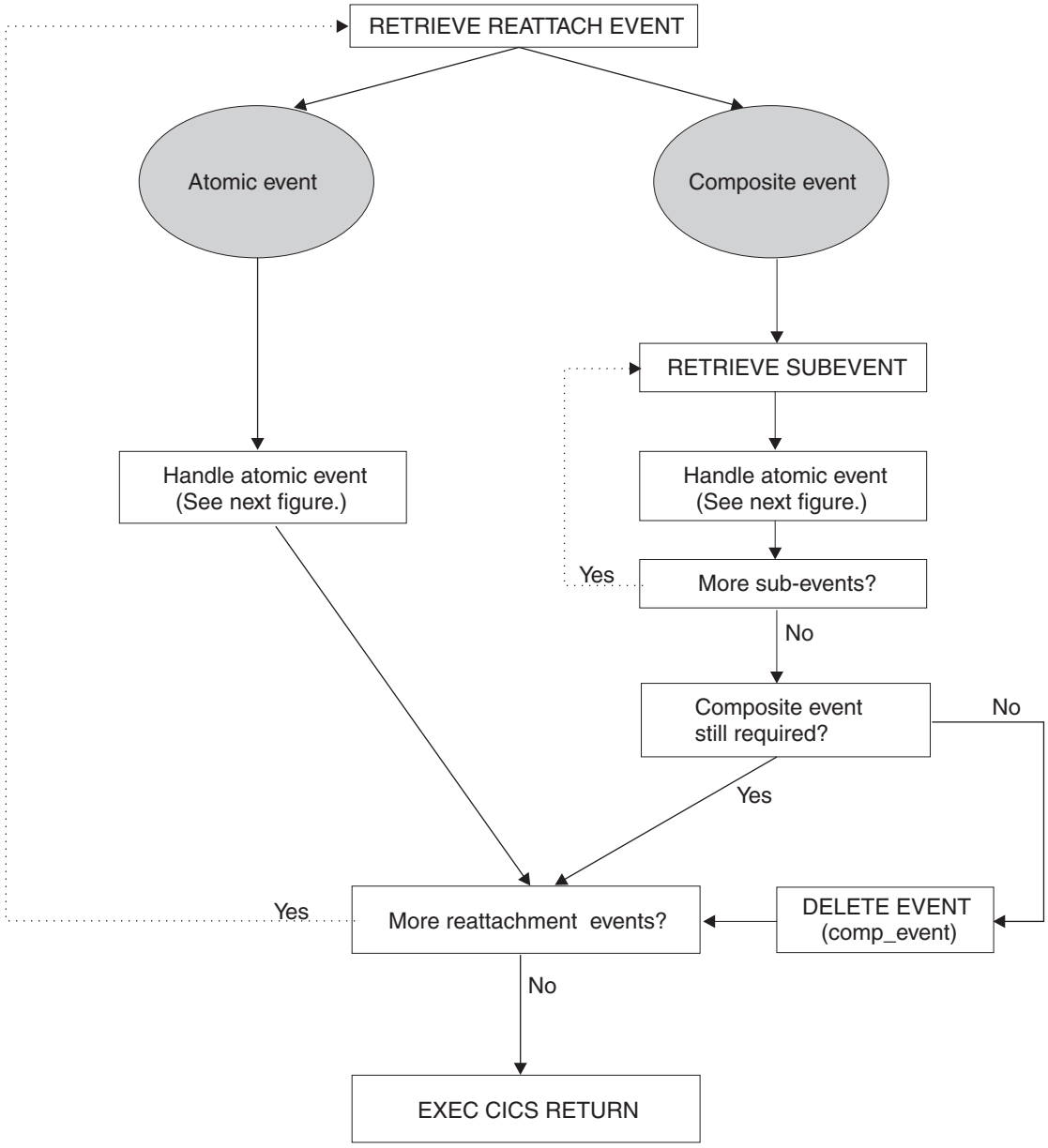


Figure 9. Handling reattachment events. The "Handle atomic event" box is expanded in Figure 10 on page 30. The figure shows multiple reattachment events being handled in a single activation—you may prefer to handle one per activation.

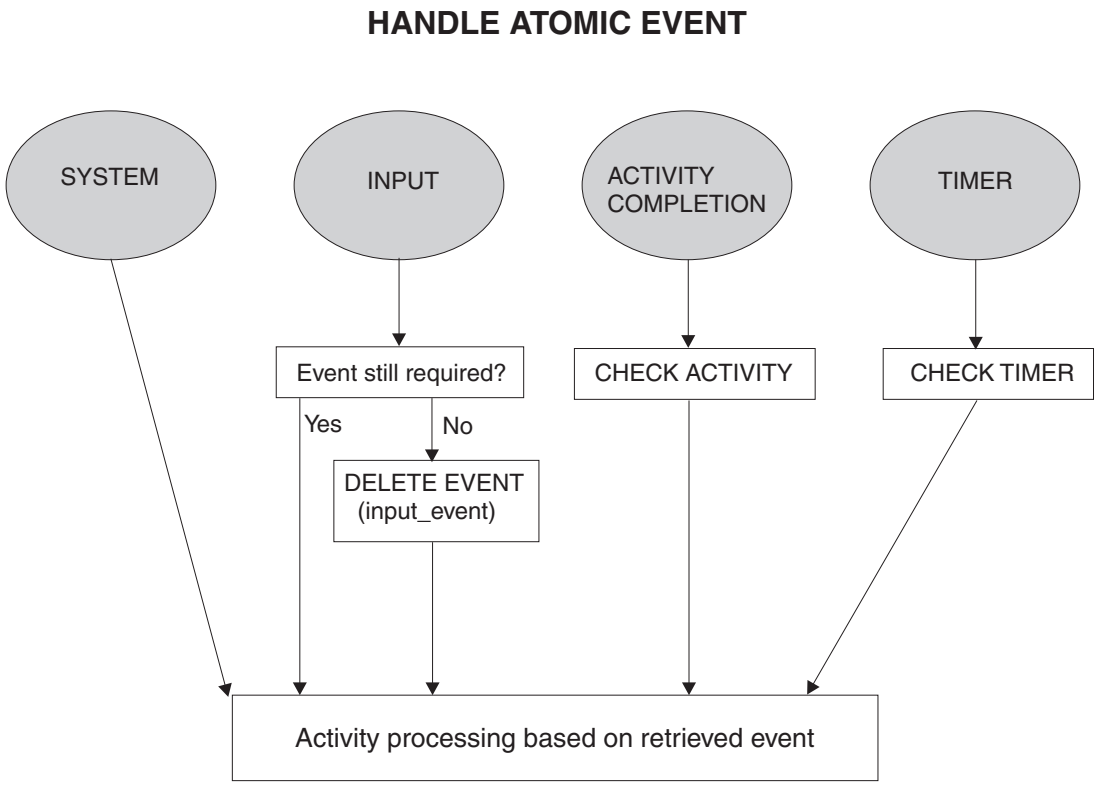


Figure 10. Handling atomic events

Note:

1. Figure 9 on page 29 shows multiple reattachment events being handled in a single activation. This may not always be appropriate. You may want always to retrieve only one reattachment event per activation, even if there is more than one event on the reattachment queue. This could be the case if, for example, you want a syncpoint to be taken between each processing step. (Note especially that a child activity that is run asynchronously is not started until a syncpoint occurs when its parent returns. Dealing with many reattachment events in the same activation could delay the start of the child.)
2. The figures show input and composite events being explicitly deleted by means of DELETE EVENT commands. This is not always strictly necessary—see "Using the ENDACTIVITY option of the RETURN command" on page 31. Similarly, it may not always be necessary to issue CHECK TIMER commands. If you don't, timer events can be deleted by means of a RETURN ENDACTIVITY command issued on the activity's final activation.

Activity completion

An activity completes normally when it returns with no user events in its event pool.

When an activity issues an EXEC CICS RETURN command (without the ENDACTIVITY option):

1. If the activity has correctly dealt with at least one reattachment event during its current activation (see "Handling reattachment events" on page 26):

   If there are events on the reattachment queue
   The activity is immediately reactivated to deal with the fired events.

   If there are no events on the reattachment queue

If there are user events in the event pool
The activity becomes dormant until a reattachment event occurs.

If there are no user events in the event pool
The activity completes normally.

2. If the activity has not correctly dealt with at least one reattachment event during its current activation, it abends.

Using the ENDACTIVITY option of the RETURN command:

You can use the ENDACTIVITY option of the **EXEC CICS RETURN** command to signal that an activity program has completed all of its processing steps and is not to be reactivated.

About this task

Optionally, an activity program can use the ENDACTIVITY option of the **EXEC CICS RETURN** command to signal that it has completed all its processing steps and should not be reactivated. One advantage of using ENDACTIVITY is that the activity program does not have to bother about deleting user events—other than activity completion events—from its event pool before completing; the events are deleted automatically by CICS.

When an activity issues an **EXEC CICS RETURN ENDACTIVITY** command:

If there are no user events in the event pool of the activity
The activity completes normally.

If there are user events, fired or unfired, in the event pool of the activity

- If one or more of the events are activity completion events, the activity abends. Trying to force an activity to complete before it has dealt with one or more of its child activities is a program logic error.
- If none of the events are activity completion events, the events are deleted and the activity completes normally.

It is recommended that you issue a **RETURN ENDACTIVITY** command at the end of the final activation of an activity, as a way of ensuring that the activity completes. For example, if, through a program logic error, an activity returns from what it believes to be its final activation with an unfired event in its event pool, it is possible that the activity could go dormant forever, and never complete. Coding **RETURN ENDACTIVITY** deletes the event and forces the activity to complete.

Chapter 4. The Sale example application

The Sale example application is a set of programs that demonstrates how to use CICS business transaction services to manage business transactions.

This section contains:

- "Overview of the Sale application"
- "The initial request" on page 35
- "The root activity" on page 38
- "Transferring input and output data" on page 44

Overview of the Sale application

Follow this *Sale* example to see the workings of a business transaction that is made up of four basic actions; order entry, delivery, invoice, and payment.

A Sale business transaction is started by a terminal-user selecting the `Sale` option from a menu of business transactions. This selection causes an instance of the transaction to be created and its root activity to be started. The root activity creates and runs, in sequence, four child activities that implement the four actions of the business transaction:

1. The *Order activity* obtains order data from the user, and validates it.
2. Successful completion of the Order activity causes the *Delivery activity* to be started.
3. Completion of the Delivery activity causes the *Invoice activity* to be started.
4. When payment is received and recorded by the *Payment activity*, the Sale business transaction is complete.

Data flows

Use this flow diagram to follow the data flow in the Sale example application.

Figure 11 on page 34 shows, in simplified form, data flows in the Sale example application.

© Copyright IBM Corp. 1999, 2011

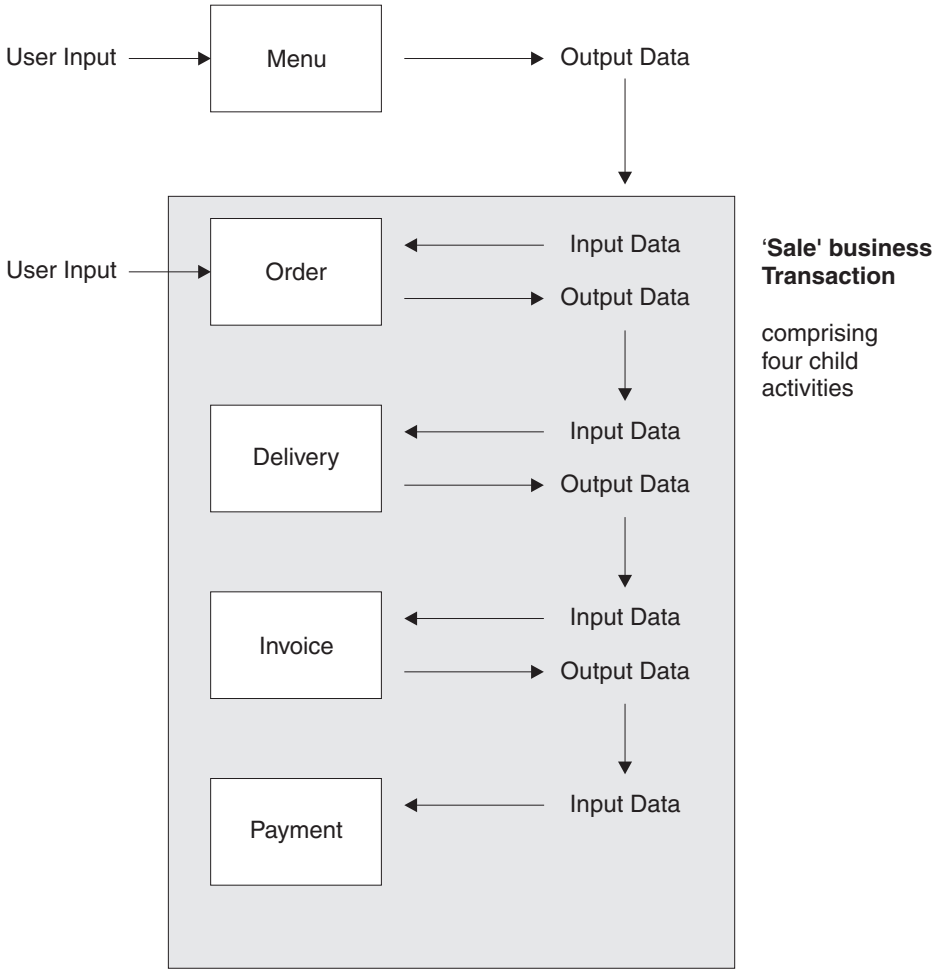


Figure 11. Data flow in the Sale example application. (The root activity is not shown.)

1. Customer data (for example, an account number) collected after the terminal user selects the `Sale` menu option is used as input to the Order activity.
2. Customer data collected by the Order activity is used as input to the Delivery activity.
3. The output data produced by the Delivery activity is used as input to the Invoice activity.
4. The output produced by the Invoice activity is used as input to the Payment activity.

Note: The first activity (Order) requires input from the terminal user. For the purposes of this chapter, subsequent activities (Delivery, Invoice and Payment) are assumed not to require any user involvement and are triggered serially in the background after the Order activity has completed successfully. In later chapters, this assumption is changed to illustrate additional function.

CICS transactions and programs

These CICS transactions and programs make up the basic Sale application.

Table 4 on page 35 shows the CICS transactions and programs that make up the basic Sale application described in this chapter.

Table 4. Transactions and programs in the basic Sale application

Transid	Program	Comments
MENU	MNU001	Menu of business transactions
—	SAL001	Creates and starts the Sale business transaction
SALE	SAL002	BTS root activity, manages the child activities that comprise the Sale business transaction
SORD	ORD001	Order activity
SDEL	DEL001	Delivery activity
SINV	INV001	Invoice activity
SPAY	PAY001	Payment activity

Note:

1. In later chapters, the Sale example application is extended to illustrate more advanced features of BTS, such as:
   - Parallel activities
   - User-related activities
   - Compensation actions.
2. For the sake of clarity, the basic example does not include any error handling code. As explained in "Checking the response from a child activity" on page 18, in a real application, after a parent has requested a child activity to be run it must issue a CHECK ACTIVITY command to:
   - Check the response from the child
   - Check the completion status of the child
   - Delete the child's completion event, if the child has completed.

   The uses of the CHECK ACTIVITY command are fully described later, in Chapter 5, "Dealing with BTS errors and response codes," on page 49.

The initial request

In COBOL pseudocode, DFH0SAL1 creates and starts an instance of the Sale business transaction. The initial request to start a Sale business transaction is handled by the DFH0SAL0 and DFH0SAL1 programs.

When a terminal user selects the `Sale` menu option by typing `SALM`, the menu program DFH0SAL0 links to the DFH0SAL1 program to service the request. The DFH0SAL1 program establishes a unique reference for this instance of the Sale business transaction and starts it.

```
Identification Division.
Program-id. DFH0SAL1.
Environment Division.
Data Division.
Working-Storage Section.
01  Sales-Reference              pic x(36) value low-values.
    .
01  Process-Type                 pic x(8)  value 'Sales'.
    .
Linkage Section.
01  DFHEIBLK.
    .
01  DFHCOMMAREA.
    .
Procedure Division using DFHEIBLK DFHCOMMAREA.
In-The-Beginning.
    .
    .. create unique sales reference ..
    .
    EXEC CICS DEFINE PROCESS(Sales-Reference) PROCESSTYPE(Process-Type)
                TRANSID('SALE')
                PROGRAM('SAL002')
             RESP(data-area) RESP2(data-area) END-EXEC
      .
    EXEC CICS RUN ACQPROCESS
                SYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RETURN END-EXEC
End Program.
```

Figure 12. Pseudocode for the DFH0SAL1 program. DFH0SAL1 creates and starts an instance of the Sale business transaction.

For more information about the sample application, see *The BTS sample application*.

Creating the business transaction

To create an instance of the Sale business transaction, SAL001 issues a DEFINE PROCESS command. Use the options here to specify your business transaction.

The PROGRAM option of DEFINE PROCESS defines a program to run under the control of CICS business transaction services—a root activity program that typically manages the ordering and execution of the child activities that make up a business transaction. In this case, the program is SAL002, which is the root activity program for the Sale business transaction.

The PROCESS option uniquely identifies this business transaction instance from others. (The creation of a unique reference is managed by the user. Typically, you might use a customer reference or account number.)

The PROCESSTYPE option categorizes the business transaction by assigning it a process-type of 'Sales'. Categorizing your processes (business transactions) in this way means that you can browse details of individual processes—and their constituent activities—more easily.

The TRANSID option serves a number of purposes:

Security
: If security is active, CICS performs a security check to see if the requestor has authority to use the specified transaction identifier (transid). Thus, in this example, there would be a check on whether the requestor is authorized to create a new instance of the Sale business transaction.

Externals

When a business transaction is started, its root activity program begins executing, and any external inquiry such as CEMT shows work being done under the root activity's transaction identifier.

In the Sale application, the Sale business transaction is started under the control of the MENU transaction; however, the actual start of an instance of the Sale transaction occurs when control is passed to the root activity program, SAL002. At this point, the transaction identifier changes from MENU to SALE.

Root activity

Later restarts of a root activity may be required to deal with child activities that are executed with the RUN ACTIVITY ASYNCHRONOUS command (the child activities are executed asynchronously with the root activity, are not included in its unit of work, and have different transaction identifiers).

In the Sale application, the SAL002 root activity program is attached under the SALE transaction identifier to deal with the Delivery, Invoice, and Payment activities, that all execute asynchronously, under separate UOW scope, and under different transaction identifiers.

Monitoring and statistics

The transaction identifier can be used to track resource usage for monitoring, statistics, and accounting purposes. It allows monitoring and statistics information to be related to a CICS business transaction services process.

DEFINE PROCESS is a synchronous request and control is returned to the requesting program when BTS has accepted the request and added the process to the set that it is currently managing.

The addition of the process is not committed until the current unit of work has taken a successful syncpoint. If the requesting task abends before the syncpoint is taken, the request to add the process is canceled. (Thus it is not possible to enquire on or to browse the process until the syncpoint has been taken.)

Starting the business transaction

To start this instance of the Sale business transaction, on return from the DEFINE PROCESS request SAL001 issues a RUN ACQPROCESS command.

A program can “acquire” a process in two ways: by defining it, or by issuing an ACQUIRE PROCESS command. Here, SAL001 has acquired a process by defining it; thus the RUN ACQPROCESS causes the SAL002 program specified on the DEFINE PROCESS command to be executed.

Using RUN causes the process to be activated in a separate unit of work from the unit of work of the requesting transaction, under the transaction identifier specified on the TRANSID option of the DEFINE PROCESS command. (A LINK ACQPROCESS command would have caused SAL002 to be executed in the same unit of work as MNU001 and SAL001, and under the same TRANSID, MENU.) The advantages of giving a process a separate TRANSID from the TRANSID of its creator are explained in “Creating the business transaction” on page 36. The SYNCHRONOUS option on the RUN command causes SAL002 to be executed synchronously with SAL001.

Although a RUN ACQPROCESS command causes a process to be activated in a separate unit of work from the unit of work of its requestor, the start and finish of the activation are related to the sync points of the requestor. In the example application, the SAL002 root activity runs its first child activity (Order) synchronously and as part of its own unit of work. If the Order activity is successfully completed (in the business sense as well as the transactional sense), the Sale business transaction is accepted. If not, it is rejected. “Accepted” means committed - this instance of the Sale transaction is ready to start its next activity. “Rejected” means rolled back - this instance of the Sale transaction no longer exists.

The root activity

The SAL001 program starts a new instance of the Sale business transaction by starting the SAL002 program, running under the transaction ID SALE. SAL002 implements a root activity that manages the inter-relationship, ordering, and execution of the child activities that make up the Sale business transaction.

A root activity program such as SAL002 is designed to be reattached by CICS business transaction services when events in which it is interested are triggered. The activity program determines which of the possible events caused it to be attached and what to do as a result. A typical sequence (simplified) is:

1. The root activity requests BTS to run a child activity (possibly several child activities), and to notify it when the child has completed.
2. The root activity “sleeps” while waiting for the child activity to complete.
3. BTS reattaches the root activity because the child activity has completed.
4. The root activity requests the next child activity to run.
5. Steps 1 through 4 are repeated until the business transaction is complete.

Thus, even though the root activity is not initiated from a terminal, you could think of its style as being “pseudoconversational”.

Figure 13 on page 39 shows, in COBOL pseudocode, the Sale root activity program, SAL002.

```
Identification Division.
Program-id. SAL002.
Environment Division.
Data Division.
Working-Storage Section.
01  RC                          pic s9(8) comp.
01  Process-Name                pic x(36).
01  Event-Name                  pic x(16).
    88  DFH-Initial             value 'DFHINITIAL'.
    88  Delivery-Complete       value 'Delivry-Complete'.
    88  Invoice-Complete        value 'Invoice-Complete'.
    88  Payment-Complete        value 'Payment-Complete'.
01  Sale-Container               pic x(16) value 'Sale'.
01  Order-Container              pic x(16) value 'Order'.
01  Order-Buffer                pic x(..).
01  Delivery-Container           pic x(16) value 'Delivery'.
01  Delivery-Buffer              pic x(..).
01  Invoice-Container            pic x(16) value 'Invoice'.
01  Invoice-Buffer               pic x(..).
Linkage Section.
01  DFHEIBLK.
    .
Procedure Division.
Begin-Process.
     .
    EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
              RESP(RC) END-EXEC
      .
    If RC NOT = DFHRESP(NORMAL)
      .
    End-If.
      .
    Evaluate True
      When DFH-Initial
        Perform Initial-Activity
        Perform Order-Activity
        Perform Delivery-Activity
      When Delivery-Complete
        Perform Invoice-Activity
      When Invoice-Complete
        Perform Payment-Activity
      When Payment-Complete
        Perform End-Process
      When Other
        .
    End Evaluate.
        .
    EXEC CICS RETURN END-EXEC
    .
```

Figure 13. Pseudocode for SAL002, the root activity program for the Sale business transaction (Part 1)

```
Initial-Activity.
    .
    EXEC CICS ASSIGN PROCESS(Process-Name)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Order-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Order')
                 TRANSID('SORD')
                 PROGRAM('ORD001')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Sale-Container)
                 ACTIVITY('Order') FROM(Process-Name)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS LINK ACTIVITY('Order')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Delivery-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Delivery')
                 TRANSID('SDEL')
                 EVENT('Delivry-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Order-Container)
                 ACTIVITY('Order') INTO(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Order-Container)
                 ACTIVITY('Delivery') FROM(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Delivery')
                 ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 14. Pseudocode for SAL002, the root activity program for the Sale business transaction (Part 2)

```
Invoice-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Invoice')
                TRANSID('SINV')
                EVENT('Invoice-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Delivery-Container)
                 ACTIVITY('Delivery') INTO(Delivery-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Delivery-Container)
                 ACTIVITY('Invoice') FROM(Delivery-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Invoice')
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Payment-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Payment')
                TRANSID('SPAY')
                EVENT('Payment-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Invoice-Container)
                 ACTIVITY('Invoice') INTO(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Invoice-Container)
                 ACTIVITY('Payment') FROM(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Payment')
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
End-Process.
    .
    EXEC CICS RETURN ENDACTIVITY
             RESP(data-area) RESP2(data-area) END-EXEC
End Program.
```

Figure 15. Pseudocode for SAL002, the root activity program for the Sale business transaction (Part 3)

The following discussion steps through the SAL002 pseudocode shown in Figure 13 on page 39:

1. The root activity determines what event caused it to be attached by issuing the following command:

   ```
   EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
            RESP(data-area) RESP2(data-area) END-EXEC
   ```

   The first time an activity is started during a process, the event returned is the system event DFHINITIAL. This event tells the activity to perform any initial housekeeping.

   In this example, CICS initially invokes the SAL002 root activity as a result of the RUN ACQPROCESS command issued by the SAL001 program. As part of its initial housekeeping, SAL002 uses the EXEC CICS ASSIGN PROCESS command to discover the name of this instance of the business transaction (process). (The name of the process instance was assigned by the DEFINE PROCESS command, and might be, for example, a customer reference or account number.)

2. The root activity creates its first child activity, which in this case is the Order activity:

```
EXEC CICS DEFINE ACTIVITY('Order')
             TRANSID('SORD')
             PROGRAM('ORD001')
          RESP(data-area) RESP2(data-area) END-EXEC
```

   The DEFINE ACTIVITY command requests CICS business transaction services to add an activity to a business transaction (process). In this example, SAL002 adds an activity called *Order* to the Sale business transaction. It is implemented by program ORD001. The TRANSID option specifies that, if the Order activity is run in its own unit of work, it runs under transaction identifier SORD.

3. When the Order activity has been added, SAL002 uses the PUT CONTAINER command to provide it with some input data.

```
EXEC CICS PUT CONTAINER(Sale-Container)
         ACTIVITY('Order') FROM(Process-Name)
         RESP(data-area) RESP2(data-area) END-EXEC
```

   The input data is placed in a data-container named *Sale* (the value of the variable *Sale-Container*). The ACTIVITY option of PUT CONTAINER associates the *Sale* data-container with the Order activity.

   Note: An activity can have many data-containers associated with it. A data-container is associated with an activity by being named on a command (such as PUT CONTAINER) that specifies the activity.

   Two or more activities can each have a data-container named, for example, *Order*.

   The data put into the *Sale* data-container is the process name—that is, the unique reference that identifies this instance of the Sale business transaction. The process name in this case is the customer reference or account number specified on the DEFINE PROCESS command in SAL001.

4. SAL002 requests BTS to start the Order activity:

```
EXEC CICS LINK ACTIVITY('Order')
         RESP(data-area) RESP2(data-area) END-EXEC
```

   The LINK ACTIVITY command causes the ORD001 program to be executed synchronously with SAL002 and to be included as part of the current unit of work. The TRANSID option of the DEFINE ACTIVITY command is ignored - LINK ACTIVITY causes the Order activity to run under the transaction identifier of the requestor, SALE.

   The Order activity collects order details from the terminal operator and validates them. The ORD001 program converses with the terminal operator until the order is accepted. It then returns the validated details in an output data-container.

5. When the Order activity completes, SAL002 creates the Delivery activity:

```
EXEC CICS DEFINE ACTIVITY('Delivery')
             TRANSID('SDEL')
             EVENT('Delivry-Complete')
          RESP(data-area) RESP2(data-area) END-EXEC
```

   The Delivery activity is to be executed asynchronously with the root activity. When an activity completes, its completion event fires. The EVENT option *names* the completion event of the Delivery activity as *Delivry-Complete*, and thus defines it. Defining the event allows it to be referenced and checked for.

   CICS reattaches an activity on the firing of any event, other than a subevent, that is in its event pool. (An event pool of an activity contains events that

have been defined to the activity, plus the DFHINITIAL system event.) Thus, the SAL002 root activity is attached again when the completion event of the Delivery activity(*Delivry-Complete*) fires.

Note: All child activities have completion events, that fire when the activities complete. If the EVENT option of DEFINE ACTIVITY is not used, CICS gives the completion event the same name as the activity itself.

For child activities like the *Order* activity, that are always executed *synchronously* with the parent, the EVENT option is not often used. Normally, the firing of a completion event of the synchronous activity does not cause the parent to be reattached, because the event is deleted (by a CHECK ACTIVITY command) during the current activation of the parent. Therefore the event never needs to be tested for by name, among several other possible reattachment events.

The CHECK ACTIVITY command is described in Chapter 5, "Dealing with BTS errors and response codes," on page 49.

6. SAL002 makes the data returned by the Order activity available to the Delivery activity:

```
EXEC CICS GET CONTAINER(Order-Container)
              ACTIVITY('Order') INTO(Order-Buffer)
          RESP(data-area) RESP2(data-area) END-EXEC

EXEC CICS PUT CONTAINER(Order-Container)
              ACTIVITY('Delivery') FROM(Order-Buffer)
          RESP(data-area) RESP2(data-area) END-EXEC
```

   Here, the GET and PUT commands are used to transfer data from the output data-container of the Order activity to the input data-container of the Delivery activity (both of which are named *Order*). These are different data-containers, although they share the same name, they are associated with different activities.

7. SAL002 requests BTS to start the Delivery activity:

```
EXEC CICS RUN ACTIVITY('Delivery')
              ASYNCHRONOUS
          RESP(data-area) RESP2(data-area) END-EXEC
```

   Because RUN rather than LINK is used, the Delivery activity is executed as a separate unit of work, and under the transaction identifier specified on the TRANSID option of the DEFINE ACTIVITY command. (The RUN command always activates the specified process or activity in a new unit of work.) Because the ASYNCHRONOUS option is used, the Delivery activity is executed asynchronously with SAL002, and starts only if the current unit of work completes successfully.

8. SAL002 issues an EXEC CICS RETURN command. Because there is a user event in its event pool—the completion event for the Delivery activity—the root activity does not complete but becomes dormant. Control is returned to SAL001, then to MNU001, and finally to CICS. CICS takes a sync point and commits the following:
   - The creation of a new Sale business transaction
   - Work done by the Order activity, and its input and output data-containers
   - The request to run the Delivery activity, and its input data-container
   - The condition under which the SAL002 root activity is to be reactivated.

   After the CICS sync point, the menu of business transactions is displayed again on the terminal of the user, ready for further selection. The remaining

activities are completed, without reference to the terminal user, under the control of CICS business transaction services. The SAL002 program no longer exists in memory, and the existence of this instance of the Sale business transaction is known only to BTS.

CICS business transaction services start the Delivery activity (SDEL) as requested. (BTS participates as a resource manager for the transaction.) On completion of the Delivery activity, BTS reactivates the Sale root activity—that is, the SAL002 program under the transaction identifier SALE.

9. The SAL002 program is entered at the top again, and so determines what event caused it to be reactivated by issuing the RETRIEVE REATTACH EVENT command. This time, however, the event returned is *Delivry-Complete*. Having established which child activity has completed, SAL002 determines that the next activity to be started is the Invoice activity.

   As with the Delivery activity, SAL002 sets the parameters of the Invoice activity, input data, and execution options before requesting the activity to be run. It then issues an EXEC CICS RETURN command and becomes dormant, waiting to be reactivated for this instance of the Sale business transaction.

10. The pattern implied in step 9 is repeated until the Payment activity completes, at which point the Sale business transaction is complete. SAL002 issues an EXEC CICS RETURN command on which the ENDACTIVITY option is specified. This option indicates to CICS that the processing of the root activity is complete, and that it no longer wants to be reactivated if defined or system events occur. The business transaction ends.

Transferring input and output data

Follow this task to see how to transfer data between a parent and a child activity. The task uses the Delivery activity Sale application as an example.

About this task

The SAL002 root activity creates the Delivery child activity by issuing a **DEFINE ACTIVITY** command.

```
Delivery-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Delivery')
                TRANSID('SDEL')
                EVENT('Delivry-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Order-Container)
                 ACTIVITY('Order') INTO(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC

    EXEC CICS PUT CONTAINER(Order-Container)
                 ACTIVITY('Delivery') FROM(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
```

Figure 16. Creating the Delivery activity

The **GET CONTAINER** command retrieves the data returned by the Order activity, and places it in a storage buffer. The data is retrieved from the output data-container of the Order activity, which is named *Order*.

Note: A child activity's data-containers are accessible to its parent even after the child has completed. An activity's containers are only destroyed when the activity itself is destroyed. An activity is destroyed:

- Automatically by CICS, when its parent completes.
- Before this, if its parent issues a **DELETE ACTIVITY** command against it.

The **PUT CONTAINER** command associates a data-container (also named *Order*) with the Delivery activity, and places the retrieved data in it.

The implementation of the Delivery activity is shown in Figure 17 on page 46.

```
Identification Division.
Program-id. DEL001.
Environment Division.
Data Division.
Working-Storage Section.
01  Event-Name              pic x(16).
    88  DFH-Initial         value 'DFHINITIAL'.
01  Order-Ptr               usage is pointer.
01  Order-Container         pic x(16) value 'Order'.
01  Delivery-Container      pic x(16) value 'Delivery'.
01  Deliver-Data.
    .
Linkage Section.
01  DFHEIBLK.
    .
01  Order-Details.
    05  Order-Number        pic 9(8).
    .
Procedure Division..
Begin-Process.
    .
    EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
             RESP(RC) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
      .
    End-If.
    .
    Evaluate True
      When DFH-Initial
        Perform Delivery-Work
        Perform End-Activity
      When Other
        .
    End Evaluate.
    .
    EXEC CICS RETURN END-EXEC
    .
Delivery-Work.
    .
    EXEC CICS GET CONTAINER(Order-Container) SET(Order-Ptr)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    set address of Order-Details to Order-Ptr.
    .
    EXEC CICS READ FILE .....
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    .  logic to print delivery details
    .
    .
    EXEC CICS PUT CONTAINER(Delivery-Container) FROM(Delivery-Data)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
End-Activity.
    .
    EXEC CICS RETURN ENDACTIVITY
             RESP(data-area) RESP2(data-area) END-EXEC
```

Figure 17. Pseudocode for the Delivery activity

The Delivery activity issues a **GET CONTAINER** command to retrieve data from a data-container named *Order*. Because the command does not specify the ACTIVITY option, it references a data-container associated with the current activity; in other words, it references the same *Order* data-container as that referenced by the **PUT CONTAINER** command in Figure 16 on page 44.

The Delivery activity uses the input data to execute its logic. Then it issues a **PUT CONTAINER** command to store its output in a data-container named *Delivery*. Again, the ACTIVITY option is not specified, so the data-container is associated with the current (Delivery) activity.

See also “Transferring data to asynchronous activations” on page 83.

Chapter 5. Dealing with BTS errors and response codes

Each time one of your applications issues a CICS command, CICS automatically raises a condition to tell it how the command worked. This condition (which is usually NORMAL) is returned by the CICS EXEC interface in the RESP and RESP2 options of the command.

About this task

If something out of the ordinary happens, the application receives an *exceptional condition*, which means a condition other than NORMAL. By testing this condition, it can tell what happened, and possibly why.

The tasks that you can perform include:

- "Checking the response from a synchronous activity"
- "Checking the response from an asynchronous activity" on page 51
- "Getting details of activity ABENDs" on page 51
- "Trying failed activities again" on page 52

Checking the response from a synchronous activity

The Order activity of the Sale application is created and run synchronously with SAL002.

About this task

Figure 18 shows the Order activity of the Sale application being created and run synchronously with SAL002.

```
Order-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Order')
                TRANSID('SORD')
                PROGRAM('ORD001')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Sale-Container)
                ACTIVITY('Order') FROM(Process-Name)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS LINK ACTIVITY('Order')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 18. Requests to create and activate an activity. The conditions returned by the RESP and RESP2 options on the LINK ACTIVITY command do not relate to the processing of the activity itself.

The RESP and RESP2 options on a RUN ACTIVITY or LINK ACTIVITY command return any exceptional condition that is raised during the processing of the command. However, what is processed is a request for BTS to run the activity—that is, for BTS to accept and schedule the activity. Therefore, the RESP and RESP2 options do not return any exceptional condition that might result from processing the activity itself.

© Copyright IBM Corp. 1999, 2011

To check the response from the actual processing of any activity other than a root activity, you must issue one of the following commands:

CHECK ACTIVITY(child_name)
Used to check a child of the current activity.

CHECK ACQACTIVITY
Used to check the activity that the current unit of work has acquired with an ACQUIRE ACTIVITYID command.

For information about acquiring activities, see "Acquiring processes and activities" on page 61.

Root activities are a special case. They are activated automatically by BTS after a RUN ACQPROCESS or LINK ACQPROCESS command is issued; also, they do not have completion events. To check the processing of a process, and therefore of a root activity, use the CHECK ACQPROCESS command.

```
EXEC CICS CHECK ACTIVITY('Order') COMPSTATUS(status)
         RESP(RC) RESP2(data-area) END-EXEC
.
If RC NOT = DFHRESP(NORMAL)
    .
End-If.
.
If status NOT = DFHVALUE(NORMAL)
    .
End-If.
.
```

Figure 19. The Sale root activity. SAL002,. SAL002, checks to see if the Order activity completed successfully or whether an error occurred.

Because Order is one of its child activities, SAL002 uses the CHECK ACTIVITY(child_name) form of the command.

The RESP and RESP2 options on the CHECK ACTIVITY command return a condition that tells you whether the CHECK command is understood by CICS—for example, ACTIVITYERR occurs if an activity named *Order* has not been defined to SAL002.

The COMPSTATUS option returns a CVDA value indicating the completion status of the activity:

- NORMAL is returned if the activity has completed all its processing steps.
- FORCED is returned if the activity was forced to complete with a CANCEL ACTIVITY command.
- INCOMPLETE is returned if the activity needs to be reactivated in order to complete all its processing steps.
- ABEND is returned if the program that implements the activity abended.

If a child activity completes, either successfully or unsuccessfully, and its parent issues a CHECK ACTIVITY command, the execution of the command causes CICS to delete the activity-completion event. Before a parent activity completes, it should ensure that the completion events of all its child activities have been deleted.

Note: If an activity completes and a CHECK ACQACTIVITY command is issued by a program other than its parent, the activity-completion event is not deleted. For example, a program executing outside a BTS process might issue an ACQUIRE

ACTIVITYID command to acquire control of an activity within the process. It might then run the activity, and issue a CHECK ACQACTIVITY command to check the outcome. If the activity has completed, its completion event is not deleted.

The firing of the completion event causes the parent of the activity to be activated. Only if the parent issues a CHECK ACTIVITY command does CICS delete the completion event.

For an explanation of why a program executing outside a process might want to acquire an activity within the process, see Chapter 7, "Interacting with BTS processes and activities," on page 61. For an example of the use of the ACQUIRE ACTIVITYID and CHECK ACTIVITYID commands, see "Activity processing" on page 69.

Checking the response from an asynchronous activity

Asynchronous activities are treated almost identically to synchronous activities, the only difference being in the point at which the CHECK ACTIVITY command is issued.

About this task

Typically, for a synchronous activity, the CHECK ACTIVITY command is issued immediately after the RUN or LINK command. For an asynchronous activity, it might, for example, be issued:

- When the parent is reattached due to the firing of the activity's completion event.
- When the requestor is reattached due to the expiry of a timer. This could occur if the requestor expects the activity to return without completing; the requestor may then reactivate the activity by sending it an input event.

Getting details of activity ABENDs

If a CHECK ACTIVITY command returns a completion status (COMPSTATUS) of ABEND, you can use the INQUIRE ACTIVITYID command to obtain further information about how the activity abended.

Example

This example returns the name of the program in which the abend occurred, together with the corresponding CICS abend code.

```
If status = DFHVALUE(ABEND)
    .
    To get the activity-identifier of the failed child,
    start a browse of child activities
    EXEC CICS STARTBROWSE ACTIVITY
                  BROWSETOKEN(root-token)
          RESP(data-area) RESP2(data-area) END-EXEC
    .
     EXEC CICS GETNEXT ACTIVITY(child-name)
                   BROWSETOKEN(root-token)
                   ACTIVITYID(child-id)
          RESP(data-area) RESP2(data-area) END-EXEC
    .
     loop until the failed child is found by name
        EXEC CICS GETNEXT ACTIVITY(child-name)
                      BROWSETOKEN(root-token)
```

```
                    ACTIVITYID(child-id)
              RESP(data-area) RESP2(data-area) END-EXEC
   .
end child activity browse loop
Inquire on the failed child, using its activity-identifier
EXEC CICS INQUIRE ACTIVITYID(child-id)
                 ABCODE(data-area)
                 ABPROGRAM(data-area)
        RESP(data-area) RESP2(data-area) END-EXEC
```

Tip: A simpler way of obtaining the activity-identifier of the failed child activity (used on the EXEC CICS INQUIRE ACTIVITYID command) would be to code the ACTIVITYID option of the DEFINE ACTIVITY command used to define the child, and to store the returned value.

Trying failed activities again

If a child activity fails, you can try it again. The parent issues a CHECK ACTIVITY command, if it has not already done so, to check the current completion status of the child activity.

About this task

Procedure

1. Issue a RESET ACTIVITY command. The child activity is reset to its initial state: its completion event is added to the event pool of the parent, with the status set to NOTFIRED; any children of the child activity are deleted. The data-containers of the child activity are not disturbed.
2. Issue a RUN ACTIVITY command. The child activity is invoked with a DFHINITIAL event.

Chapter 6. Example of running parallel BTS activities

Many business transactions include activities that can run in parallel with one another. To illustrate parallel activities, the following example extends the Sale business transaction to support multiple delivery activities.

The logic of the Sale business transaction is changed so that an order can include multiple items, each potentially requiring delivery to a separate location. Each delivery request (activity) can run in parallel, but the customer is not invoiced until all the items have been delivered.

Data flow

Follow this example to see how data flows in the Sale example application when parallel activities are included. The root activity is not shown. Changes from the basic Sale example described in Chapter 4, "The Sale example application," on page 33 are shown in bold.

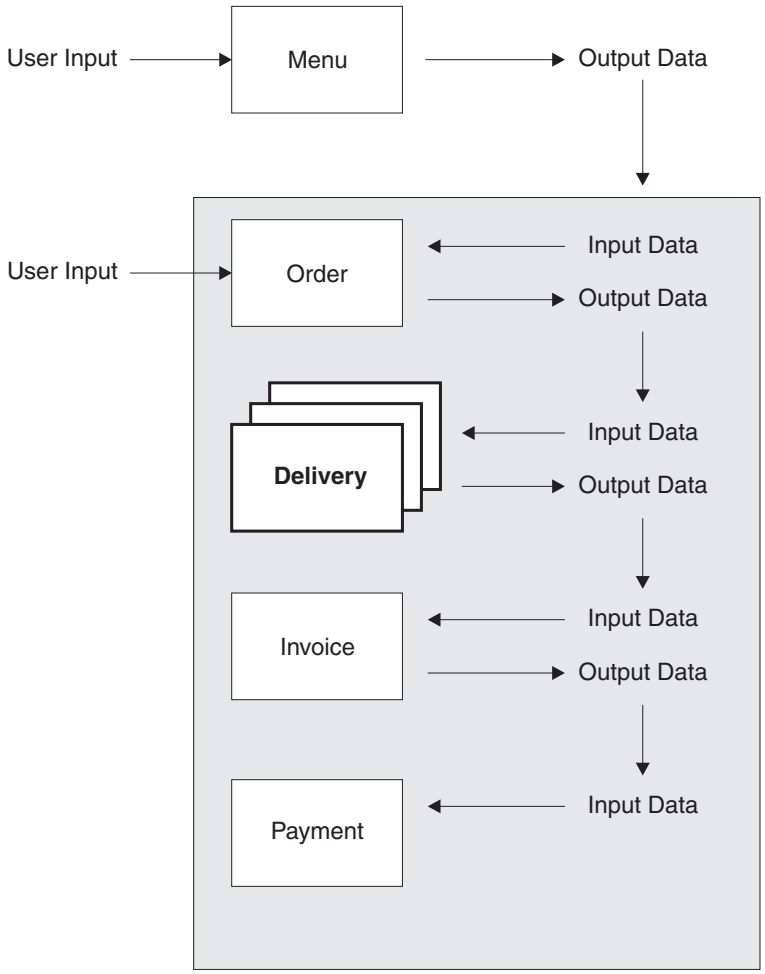


Figure 20. Data flow for parallel activities

© Copyright IBM Corp. 1999, 2011

- User data (an account number) collected after the user selects the `Sale` menu option is used as input to the Order activity.
- User data collected by the Order activity is used as input to multiple Delivery activities.
- The output data produced by the Delivery activities is used as input to the Invoice activity.
- The output produced by the Invoice activity is used as input to the Payment activity.

Root activity

Figure 21 shows, in COBOL pseudocode, the Sale root activity with modifications for parallel activities. **CHECK ACTIVITY** commands have also been added, to check the response from each child activity and to delete its completion event. The changes are in bold text.

```
Identification Division.
Program-id. SAL002.
Environment Division.
Data Division.
Working-Storage Section.
01  Switches.
    05  No-More-Events              pic x value space.
        88  No-More-Events               value 'y'.
01  Switch-Off                      Pic x value 'n'.
01  RC                              pic s9(8) comp.
01  Process-Name                    pic x(36).
01  Event-Name                      pic x(16).
    88  DFH-Initial                 value 'DFHINITIAL'.
    88  Delivery-Complete           value 'Delivry-Complete'.
    88  Invoice-Complete            value 'Invoice-Complete'.
    88  Payment-Complete            value 'Payment-Complete'.
01  Sale-Container                   pic x(16) value 'Sale'.
01  Order-Container                  pic x(16) value 'Order'.
01  Order-Buffer.
    05  Order-Count                 Pic 9(2).
    05  Order-Item occurs 1 to 20 times
        Depending on Order-Count     Pic X(10).
01  Delivery-Container              pic x(16) value 'Delivery'.
01  Delivery-Buffer.
    05  Delivery-Count              pic 9(2).
    05  Delivery-Item occurs 1 to 20 times
        Depending on Delivery-Count  pic x(30).
01  Invoice-Container               pic x(16) value 'Invoice'.
01  Invoice-Buffer                  Pic x(..).
01  Work-Activity.
    05  Work-Name                   Pic x(8) value 'Delivery'.
    05  Filler                      pic x(6) value '-Item-'.
    05  Work-Count                  pic 9(2) value zero.
01  Work-Event.
    05  Event-Name                  pic x(8) value 'Del-Comp'.
    05  Filler                      pic x(6) value '-Item-'.
    05  Event-Count                 pic x(2) value zero.
Linkage Section.
01  DFHEIBLK.
    .
```

Figure 21. The SAL002 root activity program, with modifications for parallel activities highlighted (Part 1)

```
Procedure Division.
Begin-Process.
     .
    EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
              RESP(RC) END-EXEC
     .
    If RC NOT = DFHRESP(NORMAL)
      .
    End-If.
        .
    Evaluate True
      When DFH-Initial
        Perform Initial-Activity
        Perform Order-Activity
        Perform Order-Response
        Perform Delivery-Activity
      When Delivery-Complete
        Perform Delivery-Response
        Perform Invoice-Activity
      When Invoice-Complete
        Perform Invoice-Response
        Perform Payment-Activity
      When Payment-Complete
        Perform Payment-Response
        Perform End-Process
      When Other
        .
    End Evaluate.

    EXEC CICS RETURN END-EXEC
        .
Initial-Activity.
    .
    EXEC CICS ASSIGN PROCESS(Process-Name)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
Order-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Order')
                 TRANSID('SORD')
                 PROGRAM('ORD001')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Sale-Container)
                 ACTIVITY('Order') FROM(Process-Name)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS LINK ACTIVITY('Order')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 22. The SAL002 root activity program, with modifications for parallel activities highlighted (Part 2)

```
Order-Response.
    .
    EXEC CICS CHECK ACTIVITY('Order') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
Delivery-Activity.
    .
    EXEC CICS GET CONTAINER(Order-Container)
                ACTIVITY('Order') INTO(Order-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE COMPOSITE EVENT('Delivry-Complete') AND
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    Perform Delivery-Work varying Work-Count from 1 by 1
      until Work-Count greater than Order-Count.
    .
Delivery-Work.
    .
    Move Work-Count to Event-Count
    .
    EXEC CICS DEFINE ACTIVITY(Work-Activity)
                TRANSID('SDEL')
                PROGRAM('DEL001')
                EVENT(Work-Event)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS ADD SUBEVENT(Work-Event) EVENT('Delivry-Complete')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Order-Container)
                ACTIVITY(Work-Activity) FROM(Order-Item(Work-Count))
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY(Work-Activity)
                ASYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
    .
Delivery-Response.
    .
    Move zeros to Delivery-Count
    Move Switch-Off to No-More-Events
    .
    Perform until No-More-Events
    EXEC CICS RETRIEVE SUBEVENT(Work-Event) EVENT('Delivry-Complete')
              RESP(data-area) RESP2(data-area) END-EXEC
```

Figure 23. The SAL002 root activity program, with modifications for parallel activities highlighted (Part 3)

```
    If RC NOT = DFHRESP(NORMAL)
       .
       If RC = DFHRESP(END)
          Set No-More-Events to TRUE
          EXEC CICS DELETE EVENT('Delivry-Complete')
       Else
          .
       End-If
    Else
       Move Event-Count to Work-Count
       Add 1 to Delivery-Count
       .
       EXEC CICS CHECK ACTIVITY(Work-Activity) COMPSTATUS(status)
                RESP(RC) RESP2(data-area) END-EXEC
       .
       If RC NOT = DFHRESP(NORMAL)
           .
       End-If.
       .
       If status NOT = DFHVALUE(NORMAL)
           .
       End-If.
       .
       EXEC CICS GET CONTAINER(Delivery-Container)
                     ACTIVITY(Work-Activity)
                     INTO(Delivery-Item(Work-Count))
                  RESP(data-area) RESP2(data-area) END-EXEC
    .
    End-If
    End-Perform
    .
Invoice-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Invoice')
                 TRANSID('SINV')
                 EVENT('Invoice-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Delivery-Container)
                 ACTIVITY('Invoice') FROM(Delivery-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Invoice')
                 ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Invoice-Response.
    .
    EXEC CICS CHECK ACTIVITY('Invoice') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
```

Figure 24. The SAL002 root activity program, with modifications for parallel activities highlighted (Part 4)

```
Payment-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Payment')
                TRANSID('SPAY')
                EVENT('Payment-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Invoice-Container)
                ACTIVITY('Invoice') INTO(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Invoice-Container)
                ACTIVITY('Payment') FROM(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Payment')
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Payment-Response.
    .
    EXEC CICS CHECK ACTIVITY('Payment') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
End-Process.
    .
    EXEC CICS RETURN ENDACTIVITY
             RESP(data-area) RESP2(data-area) END-EXEC
End Program.
```

Figure 25. The SAL002 root activity program, with modifications for parallel activities highlighted (Part 5)

The output from the Order activity (retrieved into the variable *Order-Buffer*) is now an array of order items. There can be 1 - 20 items in an order. Having first defined a composite event (*Delivry-Complete*), SAL002 requests a delivery activity to be run for each item ordered:

```
    EXEC CICS DEFINE COMPOSITE EVENT('Delivry-Complete') AND
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    Perform Delivery-Work varying Work-Count from 1 by 1
      until Work-Count greater than Order-Count.
```

All the delivery activities run in parallel. The following set of requests are made for each order item:

```
Delivery-Work.
    .
    Move Work-Count to Event-Count
    .
    EXEC CICS DEFINE ACTIVITY(Work-Activity)
                TRANSID('SDEL')
                PROGRAM('DEL001')
                EVENT(Work-Event)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS ADD SUBEVENT(Work-Event) EVENT('Delivry-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
```

```
    .
    EXEC CICS PUT CONTAINER(Order-Container)
                ACTIVITY(Work-Activity) FROM(Order-Item(Work-Count))
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY(Work-Activity)
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
```

Note that:

- The delivery activity for each order item is given a unique name (*Delivery-Item-n*—the value of *Work-Activity*—where *n* is the 1-20 item number).
- Each delivery activity is provided with an input data-container named *Order*, which contains one of the order items from the *Order-Buffer* array.
- The completion event for each delivery activity is given a unique name (*Del-Comp-Item-n*, the value of *Work-Event*). The **ADD SUBEVENT** command is used to add the completion event for each delivery activity to the composite event *Delivry-Complete*.

  The completion of an individual delivery activity does *not* cause SAL002 to be reattached—because the completion events of the delivery activities have been specified as subevents of the composite event *Delivry-Complete*. Instead, SAL002 is reattached when *Delivry-Complete* fires. Because *Delivry-Complete* uses the AND Boolean operator, it fires when *all* the completion events of the individual delivery activities have fired.

Before the Invoice activity is run, the output from each of the delivery activities is accumulated into a *Delivery-Item* array:

```
Delivery-Response.
    .
    Move zeros to Delivery-Count
    Move Switch-Off to No-More-Events
    .
    Perform until No-More-Events
    EXEC CICS RETRIEVE SUBEVENT(Work-Event) EVENT('Delivry-Complete')
              RESP(data-area) RESP2(data-area) END-EXEC

    If RC NOT = DFHRESP(NORMAL)
       .
       If RC = DFHRESP(END)
          Set No-More-Events to TRUE
          EXEC CICS DELETE EVENT('Delivry-Complete')
       Else
           .
       End-If
    Else
       Move Event-Count to Work-Count
       Add 1 to Delivery-Count
       .
       EXEC CICS CHECK ACTIVITY(Work-Activity) COMPSTATUS(status)
                RESP(RC) RESP2(data-area) END-EXEC
       .
       If RC NOT = DFHRESP(NORMAL)
           .
       End-If.
       .
       If status NOT = DFHVALUE(NORMAL)
           .
       End-If.
       .
       EXEC CICS GET CONTAINER(Delivery-Container)
                    ACTIVITY(Work-Activity)
                    INTO(Delivery-Item(Work-Count))
```

```
                    RESP(data-area) RESP2(data-area) END-EXEC
    .
    End-If
    End-Perform
```

The contents of the *Delivery-Item* array are placed in the input data-container of the Invoice activity.

Note that:

- When SAL002 is reattached due to the firing of the *Delivry-Complete* composite event, it uses a succession of **EXEC CICS RETRIEVE SUBEVENT** commands to retrieve, in turn, each subevent on the subevent queue of the composite event - that is, each subevent whose firing was instrumental in the firing of the composite event. These subevents are the completion events for each of the delivery activities. The number of each subevent (contained in the *Event-Count* field of *Work-Event*) is used to identify the particular delivery activity for which the subevent is the completion event.
- When all the subevents have been retrieved, SAL002 deletes the composite event *Delivry-Complete*. This deletion is not strictly necessary, because user-defined events, other than activity completion events, are automatically deleted by CICS when a **RETURN ENDACTIVITY** command is issued.

  Deleting a composite event does *not* delete any associated subevents. In this example, the subevents are the completion events for child activities. The completion event for a child activity is deleted automatically when, as here, an **EXEC CICS CHECK ACTIVITY** command is issued by the parent after the child has completed. The **CHECK ACTIVITY** command is described in Chapter 5, “Dealing with BTS errors and response codes,” on page 49.

  It is an error for an activity to issue an **EXEC CICS RETURN ENDACTIVITY** command while there are still activity completion events in its event pool.

Chapter 7. Interacting with BTS processes and activities

You can use BTS processes and activities to interact with the world outside the BTS environment. A program running outside a process can use BTS to acquire access to an activity in the process.

In the examples we have looked at so far, after the initial order details are collected from a user terminal the Sale business transaction proceeds without further interaction with the outside world. Each activity is started automatically by CICS business transaction services, following the completion of its predecessor.

In practice, many business transactions require some external interaction. For example, most business transactions include activities that require human involvement. These activities are known as *user-related activities*. User-related activities cannot be started automatically by BTS, because they rely on the user being ready to process the work. Other examples of external interactions are dependencies on input from the World Wide Web or from WebSphere® MQ queues.

For example, the CICS transactions can:

Use BTS processes as servers
A client transaction outside a process can "acquire" the root activity of the process. This enables it to pass business data to the process in the process or root activity's containers. The transaction does not become part of the process - rather, it is able to activate the process and use it as a server.

Acquire BTS activities
A transaction outside a process can acquire a descendant activity within the process. Acquiring the activity gives the transaction access to the activity's containers, and allows it to activate the activity.

Both these examples use input events to signify that a process or activity requires some external interaction to take place before it can complete.

It contains:

- "Using client/server processing" on page 62
- "Activity processing" on page 69
- "Transferring data to asynchronous activations" on page 83.

Acquiring processes and activities

Before a program that runs outside a process can activate an activity in the process, it uses the **ACQUIRE** command to acquire access to the activity.

About this task

Acquiring an activity enables the program to:

- Read and write to the containers for the activity.
- Issue various commands, including RUN and LINK, against the activity. If the acquired activity is a root activity, the program can issue the commands against the process.

© Copyright IBM Corp. 1999, 2011

To gain access to an activity from outside the process that contains it, you use the ACQUIRE command. An activity that a program accesses with an ACQUIRE command is known as an *acquired activity*.

There are two forms of the ACQUIRE command:

ACQUIRE ACTIVITYID
Acquires the specified descendant (non-root) activity.

ACQUIRE PROCESS
Acquires the root activity of the specified process.

Note: When a program defines a process, it is automatically given access to root activity for the process. This enables the defining program to access the process containers and root activity containers before running the process. When a program gains access to a root activity with *either* a DEFINE PROCESS or an ACQUIRE PROCESS command, the process is known as the *acquired process*.

For definitive information about the ACQUIRE command, see The ACQUIRE command, in the *CICS Application Programming Reference*.

Process and Activity rules

These rules apply when acquiring or accessing an activity or a process.

1. A program can acquire only one activity within the same unit of work. The activity remains acquired until the next sync point. This means, for example, that a program:
   - Cannot issue both a DEFINE PROCESS and an ACQUIRE PROCESS command within the same unit of work.
   - Cannot issue both an ACQUIRE PROCESS and an ACQUIRE ACTIVITYID command within the same unit of work. That is, it can acquire *either* a descendant activity or a root activity, not one of each.
2. If a program is executing as an activation of an activity, it cannot:
   - Acquire an activity in the same process as itself. It cannot, for example, issue ACQUIRE PROCESS for the current process.
   - Use a LINK command to activate the activity that it has acquired.
3. A process of an acquired activity is accessible in the same way as the activity itself can access it. Thus, if the acquired activity is a descendant activity:
   - The containers of the process might be read but not updated.
   - The process might not be the subject of any command - such as RUN, LINK, SUSPEND, RESUME, or RESET - that directly manipulates the process or its root activity.

   Conversely, if the acquired activity is a root activity:
   - The containers of the process might be both read and updated.
   - The process might be the subject of commands such as RUN, LINK, SUSPEND, RESUME, or RESET. The ACQPROCESS keyword on the command identifies the subject process as the one the program that issues the command has acquired in the current unit of work.

Using client/server processing

CICS business transaction services support client/server processing. These examples show you how BTS client/server processing works.

A server process is one that is typically waiting for work. When work arrives, BTS restarts the process, which retrieves any state data that it has previously saved. Typically, the client invokes the server with a named input event, and sends it some input data in a data-container. From these inputs, the server determines what actions it needs to take. It returns any output for the client in a data-container.

When the client has dealt with any output returned by the server, it releases the server process. Releasing the server means that its in-memory instance is freed. The server process is maintained only by BTS.

Client/server examples

The client/server examples in this section show:

1. A client program initiating a server process and calling it with some work to do.
2. The server defining some input events for which it might be invoked again; then performing some work and returning output to the client.
3. After dealing with the output returned by the server, the client releasing the in-memory instance of the server.
4. The client reacquiring the server process and requesting it to run again.
5. The server process determining the input event that caused it to be invoked again, and retrieving some state data that it saved when it last ran; then performing some work and returning output to the client.
6. Eventually, the client telling the server to shut down, and the server responding to this event by indicating that it must not be invoked again.

The client program

Step through the client program PRG001, in COBOL pseudocode, as it creates a server process and requests that it is run.

```
Identification Division.
Program-id. PRG001.
Environment Division.
Data Division.
Working-Storage Section.
01  RC                          pic s9(8) comp.
01  Unique-Reference            pic x(36) value low-values.
    .
01  Process-Type                pic x(8)  value 'Servers'.
    .
01  Event-Name                  pic x(16) value low-values.
    .
01  Work-Buffer.
    .
01  Work-request                Pic x.
    88  Work-New                value 'N'.
    88  Work-Continue           value 'C'.
    88  Work-End                value 'E'.

Linkage Section.
01  DFHEIBLK.
    .
01  DFHCOMMAREA.
    .
    .
```

Figure 26. Example client program, PRG001 (Part 1)

```
Procedure Division using DFHEIBLK DFHCOMMAREA.
In-The-Beginning.
    .
    EXEC CICS SEND ...
         RESP(data-area) END-EXEC
    .
    EXEC CICS RECEIVE ...
         RESP(data-area) END-EXEC
    .
    Move ..unique.. TO Unique-Reference
    Move ..request.. TO Work-Request
    .
    Evaluate True
      When Work-New
        Perform New-Process
      When Work-Continue
        Move 'SRV-WORK' TO Event-Name
        Perform Existing-Process
      When Work-End
        Move 'SRV-SHUTDOWN' TO Event-Name
        Perform Existing-Process
      When Other
         .
    End Evaluate.

    .
    EXEC CICS GET CONTAINER('Server-Out')
                 ACQPROCESS INTO(Work-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS SEND ...
         RESP(data-area) END-EXEC
    .
    EXEC CICS RETURN END-EXEC
    .
New-Process.
    .
    EXEC CICS DEFINE PROCESS(Unique-Reference) PROCESSTYPE(Process-Type)
                 TRANSID('SERV')
                 PROGRAM('SRV001')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER('Server-In')
                 ACQPROCESS FROM(Work-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACQPROCESS
                 SYNCHRONOUS
             RESP(RC) RESP2(data-area) END-EXEC
    .
```

Figure 27. Example client program, PRG001 (Part 2)

```
Existing-Process.
    .
    EXEC CICS ACQUIRE PROCESS(Unique-Reference) PROCESSTYPE(Process-Type)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER('Server-In')
                 ACQPROCESS FROM(Work-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACQPROCESS
                 SYNCHRONOUS
                 INPUTEVENT(Event-Name)
             RESP(RC) RESP2(data-area) END-EXEC
    .
End Program.
```

Figure 28. Example client program, PRG001 (Part 3)

First, PRG001 determines if this is the first time the server is to be called. If it is, it establishes a unique name for this instance of the server process. Then it creates the server process by issuing a DEFINE PROCESS command with that unique name. PRG001 provides some input data for the server in a data-container named *Server-In*:

```
    EXEC CICS PUT CONTAINER('Server-In')
                 ACQPROCESS FROM(Work-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
```

The ACQPROCESS option associates the *Server-In* container with the process that PRG001 has "acquired". A program "acquires" access to a process in one of two ways: either, as here, by defining it; or, if the process has already been defined, by issuing an ACQUIRE PROCESS command.

Having created the server process, PRG001 issues a request to run it synchronously. The RUN ACQPROCESS command causes the currently acquired process to be activated. Because RUN ACQPROCESS rather than LINK ACQPROCESS is used, the server process is run in a separate unit of work from that of the client. PRG001 waits for the server to run, and then retrieves any data returned from a data-container named *Server-Out*.

PRG001 has now temporarily finished using the server process; the implicit sync point at RETURN causes it to be released.

To use this instance of the server again, PRG001 must first acquire access to the correct process. It does this by issuing an ACQUIRE PROCESS command which specifies the unique combination of the name and process-type of the process:

```
    EXEC CICS ACQUIRE PROCESS(Unique-Reference) PROCESSTYPE(Process-Type)
             RESP(data-area) RESP2(data-area) END-EXEC
```

Once again, PRG001 provides input data for the server in a data-container named *Server-In*, and requests the process to be run:

```
    EXEC CICS RUN ACQPROCESS
                 SYNCHRONOUS
                 INPUTEVENT(Event-Name)
             RESP(RC) RESP2(data-area) END-EXEC
```

PRG001 uses the INPUTEVENT option of the RUN command to tell the server why it has been invoked—in this case, it is for *SRV-WORK*. (The server must have defined an input event of that name.)

Again, PRG001 waits for the process to complete, retrieves any returned data, and releases the process.

Eventually, PRG001 tells the server to shut down by invoking it with an event of *SRV-SHUTDOWN*.

The server program

Step through the server program SRV001, in COBOL pseudocode, as it determines why it was called, carries out housekeeping, and performs its work.

```
Identification Division.
Program-id. SRV001.
Environment Division.
Data Division.
Working-Storage Section.
01  Event-Name                  pic x(16).
    88  DFH-Initial             value 'DFHINITIAL'.
    88  SRV-Request             value 'SRV-REQUEST'.
01  Sub-Event-Name              pic x(16).
    88  SRV-Work                value 'SRV-WORK'.
    88  SRV-Shutdown            value 'SRV-SHUTDOWN'.
01  Input-Buffer.
    .
01  Output-Buffer.
    .
01  State-Buffer.
    .
Linkage Section.
01  DFHEIBLK.
    .
Procedure Division.
Begin-Process.
      .
    EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
             RESP(data-area) RESP2(data-area) END-EXEC
      .
    Evaluate True
      When DFH-Initial
        Perform Initial-Request
        Perform Server-work
      When SRV-Request
        Perform Server-Event
      When Other
         .
    End Evaluate.
        .
    EXEC CICS RETURN END-EXEC
    .
```

Figure 29. Example server program, SRV001 (Part 1)

```
Server-Event.
     .
    EXEC CICS RETRIEVE SUBEVENT(Sub-Event-Name) EVENT(Event-Name)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    Evaluate True
      When SRV-Work
        Perform Server-Work
      When SRV-Shutdown
        Perform Server-Shutdown
      When Other
         .
    End Evaluate.
    .
Initial-Request.
    .
    EXEC CICS DEFINE INPUT EVENT('SRV-WORK')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE INPUT EVENT('SRV-SHUTDOWN')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE COMPOSITE EVENT('SRV-REQUEST') OR
                 SUBEVENT1('SRV-WORK')
                 SUBEVENT2('SRV-SHUTDOWN')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    Server-Work.
    .
    EXEC CICS GET CONTAINER('Server-In') INTO(Input-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    If DFH-Initial
       EXEC CICS DEFINE ACTIVITY('Work')
                    TRANSID('SWRK')
                    PROGRAM('PRG002')
                 RESP(data-area) RESP2(data-area) END-EXEC
       .
    Else
       EXEC CICS GET CONTAINER('Previous-State') INTO(State-Buffer)
                RESP(data-area) RESP2(data-area) END-EXEC
       .
    End-If.
    .
    EXEC CICS PUT CONTAINER('Work-Input')
                 ACTIVITY('Work') FROM(Input-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Work')
                 SYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
```

Figure 30. Example server program, SRV001 (Part 2)

```
    EXEC CICS CHECK ACTIVITY('Work') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    EXEC CICS GET CONTAINER('Work-Output')
                ACTIVITY('Work') INTO(Output-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER('Previous-State') FROM(State-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER('Server-Output') FROM(Output-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Server-Shutdown.
    EXEC CICS DELETE EVENT('SRV-WORK')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DELETE EVENT('SRV-SHUTDOWN')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DELETE EVENT('SRV-REQUEST')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RETURN ENDACTIVITY
             RESP(data-area) RESP2(data-area) END-EXEC
End Program.
```

Figure 31. Example server program, SRV001 (Part 3)

The server program, SRV001, first issues a RETRIEVE REATTACH EVENT command to determine the reason for its invocation. On its first invocation, the event returned is DFHINITIAL, which tells SRV001 to perform any initial housekeeping. The housekeeping of the SRV001 program includes defining two input events for which it could later be invoked again:

```
    EXEC CICS DEFINE INPUT EVENT('SRV-WORK')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE INPUT EVENT('SRV-SHUTDOWN')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE COMPOSITE EVENT('SRV-REQUEST') OR
                 SUBEVENT1('SRV-WORK')
                 SUBEVENT2('SRV-SHUTDOWN')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

The DEFINE COMPOSITE EVENT command defines a third, composite, event (*SRV-REQUEST*), and adds the two input events to it. Because the composite event uses the OR Boolean operator, it fires when *either* of the two input events fires; SRV001 is reattached.

SRV001 obtains its input data from a data-container named *Server-In*. It then performs the work activity *Work*.

When the work activity has completed, SRV001 saves some state data for the next time it is run, and returns the output data produced by the work activity to the client program in a data-container named *Server-Output*.

On subsequent invocations, SRV001 determines that it has been invoked to perform work. (The RETRIEVE REATTACH EVENT command returns the composite event *SRV-REQUEST*, and a RETRIEVE SUBEVENT command with an event-name of *SRV-REQUEST* returns the subevent *SRV-WORK*.)

Eventually, the RETRIEVE SUBEVENT command returns the subevent *SRV-SHUTDOWN*, and SRV001 responds by ending the server process. First it deletes the user events that it has defined, then issues an EXEC CICS RETURN ENDACTIVITY command to indicate that it has completed all its processing.

Activity processing

Activity processing is organized in two activations. The first activation sets up the environment. The second activation starts when a defined external interaction occurs.

About this task

To set up the environment to enable the second activation to take place, the first activation must:

1. Define an input event that depicts the external interaction. The activity cannot now complete until this input event has been dealt with.
2. Obtain an **activity identifier** that uniquely identifies this activity-instance. To do this, it issues an ASSIGN command.

   The transaction that starts the second activation must use this identifier to gain access to the activity.
3. Save details of the activity identifier and input event to a suitable medium. For example, a VSAM file or WebSphere MQ queue, to which the transaction that starts the second activation has access.
4. Return without completing. (That is, issue an EXEC CICS RETURN command on which the ENDACTIVITY option is omitted. Because of the user event in its event pool—the input event that it has defined—the activity does not complete but becomes dormant.)

When the external interaction occurs—for example, a clerk enters some data at a terminal—the transaction that starts the second activation of the activity is invoked. This transaction must:

1. Retrieve the activity identifier and input event
2. Gain access to the activity by issuing an ACQUIRE ACTIVITYID command that specifies the activity identifier.
3. Reactivate the activity, and specify why it is being activated by issuing a RUN ACQACTIVITY command that specifies the input event.

Figure 32 on page 70 shows an activity that interacts with the outside world. The first activation sets up the environment, saves details of the activity identifier and input event to a VSAM file, and returns without completing. Some time later, a user starts the SPAR transaction from a terminal. The SPAR transaction retrieves the activity identifier and input event, issues an ACQUIRE ACTIVITYID command to gain access to the activity, supplies the activity with some input data, and

reactivates it.

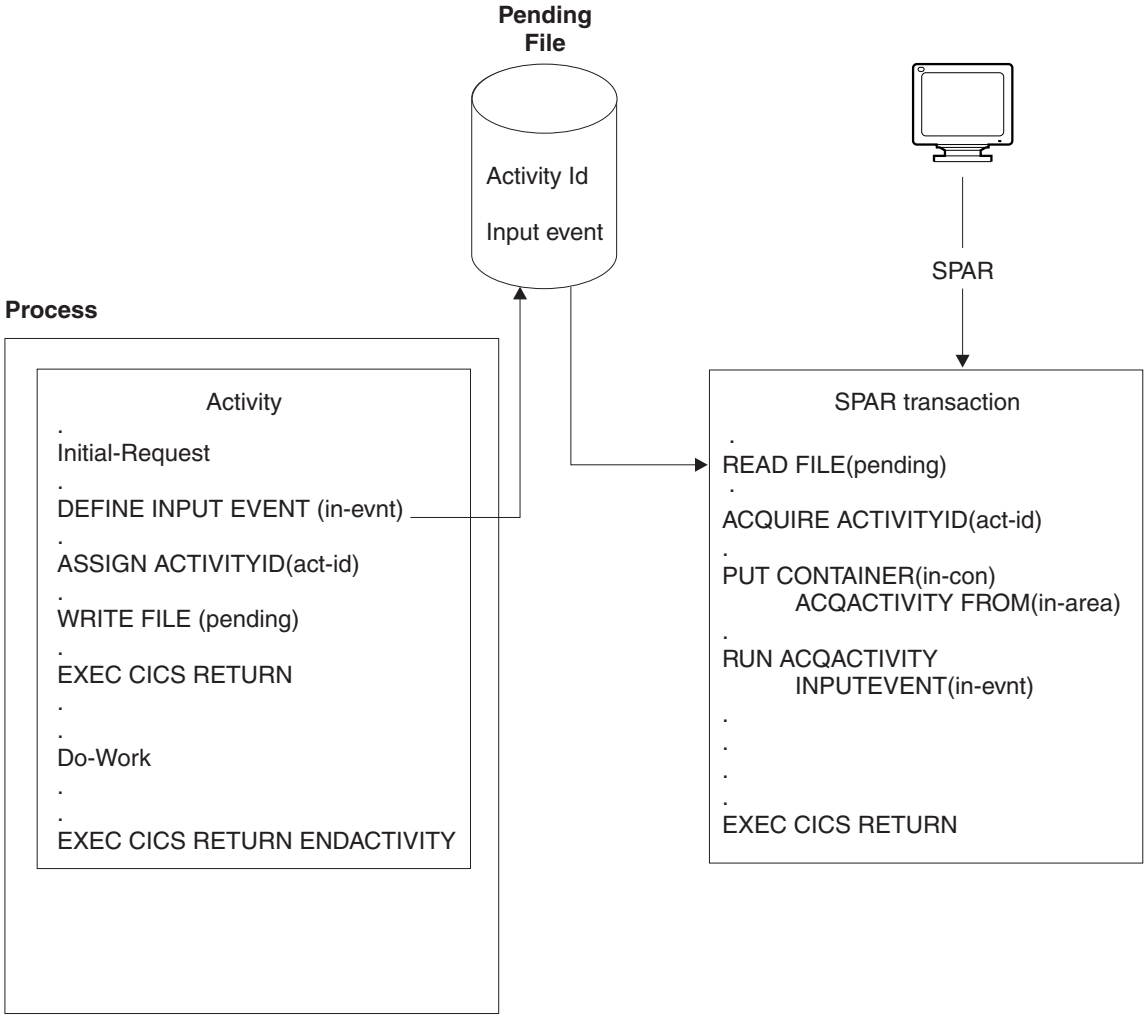


Figure 32. Acquiring an activity. On its initial activation, the activity sets up the environment and returns without completing. Some time later, the SPAR transaction is started from a terminal; it retrieves the activity identifier and input event, issues an ACQUIRE ACTIVITYID command to gain access to the activity, and reactivates it.

A user-related example

You can use the user-related example to demonstrate user-related activities by changing the logic and process flow of the Sale business transaction.

The Sale example application described in Chapter 4, "The Sale example application," on page 33 assumes that none of its later activities require human involvement. The only child activity to require human involvement is the first (Order), and this is included as part of the initial terminal request to start the new business transaction.

To demonstrate user-related activities, this section changes the logic and process flow of the Sale business transaction. Now, instead of the Invoice activity being started automatically after the Delivery activity has completed, it is not started until a user notifies the Sale transaction that the delivery has taken place. In addition, the Payment activity requires the user to enter data.

For more information about the Sale example, see Chapter 4, "The Sale example application," on page 33.

Data flow

Follow the data flows in the Sale example application when user-related activities are included.

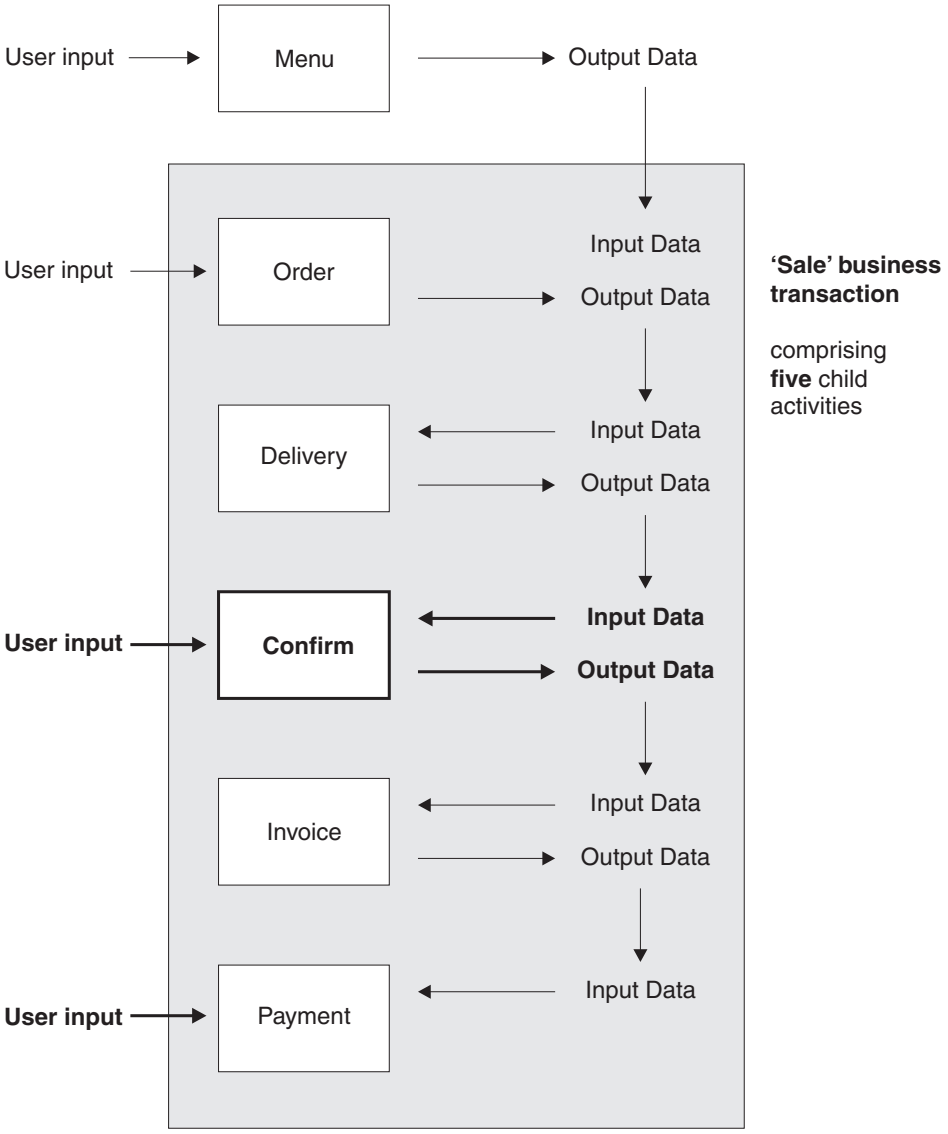


Figure 33. Data flow in the Sale example application, showing user-related activities. (The root activity is not shown.) Changes from the basic Sale example described in Chapter 4, "The Sale example application," on page 33 are shown in bold.

1. User data collected after the user selects the `Sale` menu option is used as input to the Order activity.
2. The user data collected by the Order activity is used as input to the Delivery activity.
3. The output data produced by the Delivery activity is used as input to the Confirm activity.

4. The output produced by the Confirm activity (which requires user input) is used as input to the Invoice activity.
5. The output produced by the Invoice activity is used as input to the Payment activity.

The root activity

Step through the Sale root activity, in COBOL pseudocode, with modifications for user-related activities. The main change is to introduce a new Confirm activity.

The modifications for user-related activities are highlighted in the examples here using bold text.

```
Identification Division.
Program-id. SAL002.
Environment Division.
Data Division.
Working-Storage Section.
01  RC                          pic s9(8) comp.
01  Process-Name                pic x(36).
01  Event-Name                  pic x(16).
    88  DFH-Initial             value 'DFHINITIAL'
    88  Delivery-Complete       value 'Delivry-Complete'.
    88  Delivery-Confirmed      value 'Delivry-Confirmd'.
    88  Invoice-Complete        value 'Invoice-Complete'.
    88  Payment-Complete        value 'Payment-Complete'.
01  Sale-Container               pic x(16) value 'Sale'.
01  Order-Container              pic x(16) value 'Order'.
01  Order-Buffer                pic x(..).
01  Delivery-Container           pic x(16) value 'Delivery'.
01  Delivery-Buffer             pic x(..).
01  Confirm-Container           pic x(16) value 'Confirm'.
01  Confirm-Buffer              pic x(..).
01  Invoice-Container           pic x(16) value 'Invoice'.
01  Invoice-Buffer              pic x(..).
Linkage Section.
01  DFHEIBLK.
    .
Procedure Division.
Begin-Process.
      .
    EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
              RESP(RC) END-EXEC
      .
    If RC NOT = DFHRESP(NORMAL)
      .
    End-If.
      .
    Evaluate True
      When DFH-Initial
        Perform Initial-Activity
        Perform Order-Activity
        Perform Order-Response
        Perform Delivery-Activity
      When Delivery-Complete
        Perform Delivery-Response
        Perform Delivery-Confirmation
      When Delivery-Confirmed
        Perform Confirm-Response
        Perform Invoice-Activity
      When Invoice-Complete
        Perform Invoice-Response
        Perform Payment-Activity
```

Figure 34. The SAL002 root activity program, with user-related modifications highlighted (Part 1)

```
      When Payment-Complete
        Perform Payment-Response
        Perform End-Process
      When Other
        .
    End Evaluate.
        .
    EXEC CICS RETURN END-EXEC
    .
Initial-Activity.
    .
    EXEC CICS ASSIGN PROCESS(Process-Name)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Order-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Order')
                 TRANSID('SORD')
                 PROGRAM('ORD001')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Sale-Container)
                 ACTIVITY('Order') FROM(Process-Name)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS LINK ACTIVITY('Order')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Order-Response.
    .
    EXEC CICS CHECK ACTIVITY('Order') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    .
Delivery-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Delivery')
                 TRANSID('SDEL')
                 EVENT('Delivry-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Order-Container)
                 ACTIVITY('Order') INTO(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Order-Container)
                 ACTIVITY('Delivery') FROM(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 35. The SAL002 root activity program, with user-related modifications highlighted (Part 2)

```
    EXEC CICS RUN ACTIVITY('Delivery')
                ASYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
    .
Delivery-Response.
    .
    EXEC CICS CHECK ACTIVITY('Delivery') COMPSTATUS(status)
              RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    .
Delivery-Confirmation.
    .
    EXEC CICS DEFINE ACTIVITY('Confirm')
                TRANSID('SCON')
                PROGRAM('CON001')
                EVENT('Delivry-Confirmd')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Delivery-Container)
                ACTIVITY('Delivery') INTO(Delivery-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Delivery-Container)
                ACTIVITY('Confirm') FROM(Delivery-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Confirm')
                ASYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
    .
Confirm-Response.
    .
    EXEC CICS CHECK ACTIVITY('Confirm') COMPSTATUS(status)
              RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    .
Invoice-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Invoice')
                TRANSID('SINV')
                EVENT('Invoice-Complete')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 36. The SAL002 root activity program, with user-related modifications highlighted (Part 3)

```
    EXEC CICS GET CONTAINER(Confirm-Container)
                ACTIVITY('Confirm') INTO(Confirm-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Confirm-Container)
                ACTIVITY('Invoice') FROM(Confirm-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Invoice')
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Invoice-Response.
    .
    EXEC CICS CHECK ACTIVITY('Invoice') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    .
Payment-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Payment')
                TRANSID('SPAY')
                EVENT('Payment-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Invoice-Container)
                ACTIVITY('Invoice') INTO(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Invoice-Container)
                ACTIVITY('Payment') FROM(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Payment')
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Payment-Response.
    .
    EXEC CICS CHECK ACTIVITY('Payment') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    .
```

Figure 37. The SAL002 root activity program, with user-related modifications highlighted (Part 4)

```
End-Process.
    .
    EXEC CICS RETURN ENDACTIVITY
              RESP(data-area) RESP2(data-area) END-EXEC
End Program.
```

Figure 38. The SAL002 root activity program, with user-related modifications highlighted (Part 5)

The main change to SAL002 is to introduce a new Confirm activity. The purpose of the Confirm activity is to confirm that delivery has taken place, before the Invoice activity is started. Confirmation requires the user to enter some data. The following pseudocode creates the Confirm activity:

```
Delivery-Confirmation.
    .
    EXEC CICS DEFINE ACTIVITY('Confirm')
                 TRANSID('SCON')
                 EVENT('Delivry-Confirmd')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Because the Confirm activity is executed asynchronously with the root activity, the EVENT option of DEFINE ACTIVITY is used to name the completion event of the activity as *Delivry-Confirmd*. CICS reattaches SAL002 when the Confirm activity event fires - that is, when the Confirm activity completes.

SAL002 places the data entered by the user for the Confirm activity into a data-container named *Delivery*, and issues the RUN command:

```
    EXEC CICS GET CONTAINER(Delivery-Container)
                 ACTIVITY('Delivery') INTO(Delivery-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Delivery-Container)
                 ACTIVITY('Confirm') FROM(Delivery-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Confirm')
                 ASYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
```

Now SAL002 terminates, returning control to CICS. BTS reattaches the root activity only when the Confirm activity has completed.

Implementation of a user-related activity

You can use the Confirm activity to notify the Sale business transaction that actual delivery has taken place. In the COBOL pseudocode for program CON001, you can see how it implements the Confirm user-related activity.

```
Identification Division.
Program-id. CON001
Environment Division.
Data Division.
Working-Storage Section.
01  RC                          pic s9(8) comp.
01  Event-Name                  pic x(16).
    88  DFH-Initial             value 'DFHINITIAL'
    88  User-Ready              value 'User-Ready'.
01  Data-Record.
    03  User-Reference          pic x(60).
    03  Act-Id                  pic x(52).
    03  Usr-Event               pic x(16).
01  Data-Record-Len             pic s9(8) comp.
    .
01  Delivery-Container          pic x(16) value 'Delivery'.
01  User-Container              pic x(16) value 'User'.
01  Confirm-Container           pic x(16) value 'Confirm'.
01  Delivery-Details.
    03  Deliv-Details ..
    03  User-Details ..
    .
Linkage Section.
01  DFHEIBLK.
    .
Procedure Division.
In-The-Beginning.
      .
    EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
              RESP(RC) END-EXEC
      .
    If RC NOT = DFHRESP(NORMAL)
      .
    End-If.
      .
    Evaluate True
      When DFH-Initial
        Perform Initialization
      When User-Ready
        Perform Do-Work
      When Other
        .
    End Evaluate.
        .
    EXEC CICS RETURN END-EXEC
    .
Initialization.
    .
    EXEC CICS DEFINE INPUT EVENT(User-Ready)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS ASSIGN ACTIVITYID(Act-Id)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    MOVE User-Ready TO Usr-Event
    MOVE LENGTH OF Data-Record TO Data-Record-Len
    .
```

Figure 39. Pseudocode for the CON001 program, that implements the Confirm activity (Part 1)

```
    EXEC CICS WRITE FILE('PENDING')
                FROM(Data-Record) LENGTH(Data-Record-Len)
                RIDFLD(User-Reference)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
Do-Work.
    .
    Merge contents of two input data-containers into Delivery-Details
    .
    EXEC CICS GET CONTAINER(Delivery-Container)
             INTO(Deliv-Details)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(User-Container)
             INTO(User-Details)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    Set up the output data-container
    .
    EXEC CICS PUT CONTAINER(Confirm-Container)
             FROM(Delivery-Details)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    Clean up
    .
    EXEC CICS DELETE FILE('PENDING') RIDFLD(User-Reference)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DELETE EVENT(User-Ready)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    End the activity
    .
    EXEC CICS RETURN ENDACTIVITY
         RESP(data-area) END-EXEC
    .
End Program.
```

Figure 40. Pseudocode for the CON001 program, that implements the Confirm activity (Part 2)

The initial activation of the Confirm activity:

The Confirm activity is activated for the first time after SAL002 issues the RUN ACTIVITY command.

On this initial activation, CON001:

1. Defines an input event for which the activity might then be activated.
2. Obtains the activity identifier which uniquely identifies this activity-instance.
3. Saves the name of the input event and the activity identifier in a pending file. The record in the pending file is given a key—which could, for instance, be the customer reference number which has been used throughout to identify this instance of the Sale business transaction.
4. Returns without completing.

```
Initialization.
    .
    EXEC CICS DEFINE INPUT EVENT(User-Ready)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS ASSIGN ACTIVITYID(Act-Id)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    MOVE User-Ready TO Usr-Event
```

```
MOVE LENGTH OF Data-Record TO Data-Record-Len
.
EXEC CICS WRITE FILE('PENDING')
            FROM(Data-Record) LENGTH(Data-Record-Len)
            RIDFLD(User-Reference)
         RESP(data-area) RESP2(data-area) END-EXEC
.
```

The USRX user transaction:

When ready to confirm delivery, the user starts the USRX user-written transaction, which starts the USRCON program. You can step through the USRCON pseudocode to see how conformation takes place.

USRCON executes outside the BTS environment - it is not part of the SAL001 process that contains the Confirm activity. Figure 41 on page 81 shows, in COBOL pseudocode, the USRCON program.

```
Identification Division.
Program-id. USRCON.
Environment Division.
Data Division.
Working-Storage Section.
01  Pending-Record.
    03  User-Reference                pic x(60).
    03  Act-Id                        pic x(52).
    03  Usr-Event                     pic x(16).
    .
01  User-Container                    pic x(16) value 'User'.
01  Confirmation-Details.
    03  ..
    .
Linkage Section.
01  DFHEIBLK.
    .
01  DFHCOMMAREA.
    .
Procedure Division using DFHEIBLK DFHCOMMAREA.
In-The-Beginning.
    .
    EXEC CICS SEND MAP('......') MAPSET('......') ...
    .
    EXEC CICS RECEIVE MAP('......') MAPSET('......) ..
    .
    Move ..unique.. to User-Reference.
    .
    EXEC CICS READ FILE('PENDING')
              INTO(Pending-Record) RIDFLD(User-Reference)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    . Acquire access to the Confirm activity of the SAL001 process
    .
    EXEC CICS ACQUIRE ACTIVITYID(Act-Id)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(User-Container)
             ACQACTIVITY
             FROM(Confirmation-Details)
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACQACTIVITY
              INPUTEVENT(Usr-Event)
              SYNCHRONOUS
         RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS CHECK ACQACTIVITY COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    EXEC CICS RETURN
         RESP(data-area) END-EXEC
    .
End Program.
```

Figure 41. Pseudocode for the USRCON program, that implements the USRX transaction

First, USRCON sends a map to the screen of the user and requests a unique reference. This reference must be the same as the key used by the CON001 program. This reference might be the customer reference or account number that has been used throughout to identify this instance of the Sale business transaction. However, it might need to be more specific than this. This would be the case if, for example:

- The Sale business transaction has more than one user-related activity.
- The user-related activity has defined more than one input event.

Using the unique reference, USRCON selects the appropriate record from the pending file. It then uses the value of *Act-Id* to acquire access to the Confirm activity for the SAL001 instance of the Sale business transaction:

```
EXEC CICS ACQUIRE ACTIVITYID(Act-Id)
     RESP(data-area) RESP2(data-area) END-EXEC
```

If the ACQUIRE command is successful, USRCON has access to the containers of the Confirm activity. USRCON creates a data-container for the Confirm activity (*User*), and puts some confirmation details into it:

```
EXEC CICS PUT CONTAINER(User-Container)
         ACQACTIVITY
         FROM(Confirmation-Details)
     RESP(data-area) RESP2(data-area) END-EXEC
```

The ACQACTIVITY option associates the new *User* container with the activity that USERCON has acquired.

Finally, USERCON activates the Confirm activity again, checks whether it completes successfully, and ends:

```
.
EXEC CICS RUN ACQACTIVITY
          INPUTEVENT(Usr-Event)
          SYNCHRONOUS
     RESP(data-area) RESP2(data-area) END-EXEC
.
EXEC CICS CHECK ACQACTIVITY COMPSTATUS(status)
         RESP(RC) RESP2(data-area) END-EXEC
.
EXEC CICS RETURN
     RESP(data-area) END-EXEC
.
```

The value of the INPUTEVENT option of the RUN command is the name of the input event previously defined by the Confirm activity. Although USRCON can check whether the activity it has acquired completes successfully, the execution of the CHECK ACQACTIVITY command does *not* cause CICS to delete the completion event of the Confirm activity. CICS deletes a completion event of the completed activity only after the execution of a CHECK ACTIVITY command issued by the parent of the activity.

The second activation of the Confirm activity:

The Confirm activity is activated for a second, and final, time by the RUN ACQACTIVITY command issued by USRCON.

On its second activation, CON001:

1. Establishes why it has been invoked.

2. Merges the contents of the two input data-containers, *Delivery* and *User*, supplied by SAL002 and USRCON.
3. Stores the updated delivery details into the output data-container of the Confirm activity (*Confirm*).

See Figure 39 on page 78. Finally, CON001 does some clean-up work. It:

1. Deletes the entry from the pending file.
2. Deletes the input event defined on its previous invocation. This is not strictly necessary, because the event would be deleted automatically by CICS on the execution of the RETURN ENDACTIVITY command that follows.
3. Issues an EXEC CICS RETURN ENDACTIVITY command to indicate that its processing is complete; the completion event of the Confirm activity (*Delivry-Confirmed*) is fired.

CICS notes completion of the Confirm activity and reattaches the root activity, because of the firing of the *Delivry-Confirmd* completion event defined by SAL002. After the execution of the CHECK ACTIVITY command issued by SAL002, CICS deletes the completion event of the Confirm activity.

Transferring data to asynchronous activations

You can transfer data to asynchronous activations by using a data container. When a server process or an acquired activity runs asynchronously, note these design considerations.

About this task

There are a number of ways in which your applications can handle the transfer of data to and from activities that are run asynchronously with the requestor. In the simplest case, a single data-container can be used for both input and output data. If the activity is activated only once, this activation presents no problems. Separate containers are used, one for input and one for output data. Again, if the activity is activated only once, this presents no problems. However, if the activity might be activated, asynchronously, multiple times, you must take care that the contents of containers are not overwritten inadvertently. You must take particular care when designing client/server applications, and applications which involve activities being acquired and run multiple times by transactions external to their parent process.

If an application chooses to run a server process or an acquired activity asynchronously, it needs to be aware of the state of the activity being activated. In most cases, the activity is dormant - awaiting the activation and ready to perform its function. The activation occurs almost immediately, the activity program runs, and places any results in a container. In a client/server application, the activity might then be left dormant, ready for the next request. If the activation is triggered by an external interaction, it is likely that the activity will complete; the firing of its completion event causes its parent to be activated again.

However, you must consider that, when the RUN ASYNCHRONOUS command is executed, the target activity might not be dormant, waiting for work - it might be in any of the other possible processing modes, or it could be suspended. If, for example, the target activity has been suspended, the asynchronous activation does not happen immediately. Thus, in a client/server application, it is possible for the client program to issue a request to the server before a previous request has been serviced. You should be aware of these possibilities when designing your

applications. If, for example, the protocol between a client program and its server activity relies on a single container for passing data, the client needs to check that the container is not occupied by a previous request before issuing subsequent requests. Another solution is for the client to use multiple containers to form a queue of requests for the server activity; the containers could be named sequentially.

Chapter 8. Compensation in BTS

You can use compensation in business transaction services (BTS) to reverse or modify actions taken by preceding activities or to stop business transactions.

If a single CICS transaction fails, any uncommitted changes that it has made to recoverable resources are automatically backed out by the CICS recovery manager. However, it is typically not practicable for a business transaction to be implemented as a single CICS transaction, due to the high rate of transaction abends and performance degradation that would result from holding locks for long periods.

Instead, using CICS business transaction services, each part of a business transaction is implemented as a separate BTS activity, consisting of one, or more CICS transactions. If an activity fails, the actions taken by preceding activities might need to be reversed, or possibly modified. Similarly, if application logic determines that the business transaction must be terminated, changes made by activities that have already completed might need to be reversed.

Modifying the actions of completed activities is called *compensation*.

Implementing compensation

Compensation is the act of modifying, or compensating for, the effects of a completed activity. The designer of the business transaction decides how compensation is implemented.

About this task

Often, compensating for an activity means undoing the actions that it took - for example, compensation for accepting an order might be to cancel the order.

Compensation of an activity is always controlled and instigated by the parent of the activity. It is therefore convenient to talk of compensation as an act that a parent performs on a child - as in “compensating an activity”. Strictly speaking, however, it is the parent that is compensated (it “receives compensation” for some previous action taken by the child. The previous action of the child is compensated for—it is reversed or modified.

Here are two ways in which you could implement compensation of a completed child activity.

1. **Run the activity again**.

   To run the activity again, you must first issue a RESET ACTIVITY command, to reset the activity to its initial state. You must then tell the activity that it is being invoked to perform compensation; you could do this by placing a flag in an input data-container. You cannot use the INPUTEVENT option of the RUN command to tell the activity why it is being invoked; specifying INPUTEVENT is invalid when an activity is in its initial state.

   In this method, the program used for compensation, the **compensation program**, is the same program used for normal (forward) execution of the activity.
2. **Define and run a new, compensation, activity**.

© Copyright IBM Corp. 1999, 2011

This method is more straightforward. You could use a PUT CONTAINER command to provide the compensation activity with the same input data that was passed to the activity for which it compensates.

In this method, the program used for compensation is likely to be different from the program used for the execution of the activity that is compensated.

The compensation example in this section uses this method.

A compensation example

You can use this example to change the logic of your Sale business transaction to issue reminders and to initiate the compensation process when certain criteria are met.

In this chapter, the logic of the Sale business transaction is changed so that :

- When payment has not been received within one week of the invoice being dispatched, a reminder is sent.
- If payment has still not been received two weeks after the reminder was sent, compensation is instigated.

  Compensation means that:

  1. The outstanding payment request is canceled.
  2. A request is sent for the goods to be returned.
  3. Confirmation of the goods being returned is required.
  4. The original order is canceled.

Process flow

Follow the Sale example application process flow with compensation actions included.

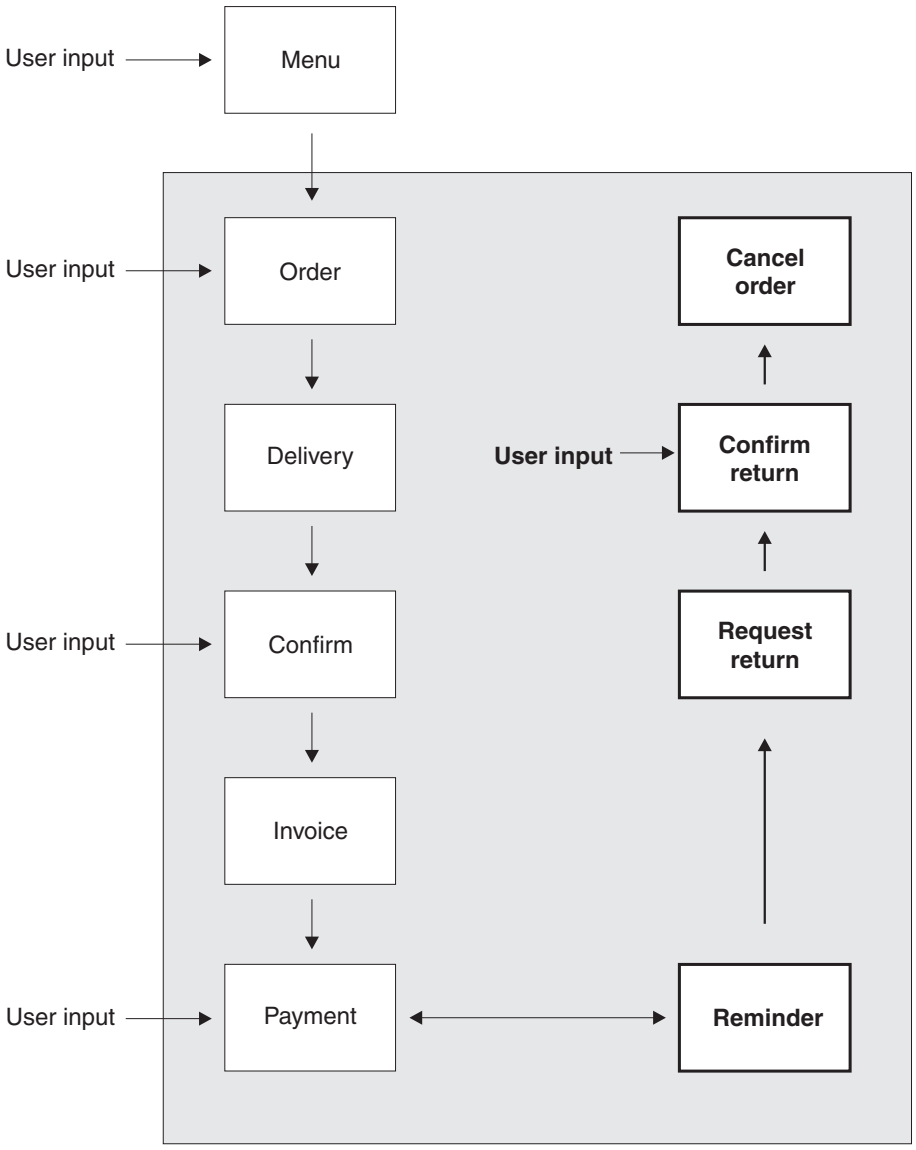


Figure 42. Process flow with compensation actions included. (The root activity is not shown.) Changes from the user-related example described in Chapter 7, "Interacting with BTS processes and activities," on page 61 are shown in bold.

- The terminal user enters the customer's order, which is accepted.
- The goods are delivered to the customer.
- The terminal user confirms that the goods have been delivered.
- An invoice is sent to the customer.
- A reminder is sent if payment has not been received within one week of the invoice being sent.
- If payment has still not been received two weeks after the reminder was sent, compensation is triggered. Compensation causes the following:
  - The outstanding payment request is canceled.
  - A letter is sent, requesting the goods to be returned.
  - Confirmation that the goods have been returned is requested.
  - The order is canceled.

The root activity

Step through the Sale root activity, in COBOL pseudocode, to see the modifications to include compensation actions.

The changes are shown here using bold text.

```
Identification Division.
Program-id. SAL002.
Environment Division.
Data Division.
Working-Storage Section.
01  RC                          pic s9(8) comp.
01  Process-Name                pic x(36).
01  Event-Name                  pic x(16).
    88  DFH-Initial             value 'DFHINITIAL'
    88  Delivery-Complete       value 'Delivry-Complete'.
    88  Delivery-Confirmed      value 'Delivry-Confirmd'.
    88  Invoice-Complete        value 'Invoice-Complete'.
    88  Payment-Due             value 'Payment-Due'.
    88  Payment-Complete        value 'Payment-Complete'.
    88  Reminder-Expired        value 'Remindr-Expired'.
    88  Reminder-Complete       value 'Remindr-Complete'.
01  Sale-Container               pic x(16) value 'Sale'.
01  Order-Container              pic x(16) value 'Order'.
01  Order-Buffer                pic x(..).
01  Delivery-Container           pic x(16) value 'Delivery'.
01  Delivery-Buffer              pic x(..).
01  Confirm-Container            pic x(16) value 'Confirm'.
01  Confirm-Buffer               pic x(..).
01  Invoice-Container            pic x(16) value 'Invoice'.
01  Invoice-Buffer               pic x(..).
01  Reminder-Container           pic x(16) value 'Reminder'.
01  Status                       pic x(16).
Linkage Section.
01  DFHEIBLK.
    .
```

Figure 43. The SAL002 root activity program, including compensation actions (Part 1)

```
Procedure Division.
Begin-Process.
     .
    EXEC CICS RETRIEVE REATTACH EVENT(Event-Name)
              RESP(RC) END-EXEC
     .
    If RC NOT = DFHRESP(NORMAL)
     .
    End-If.
     .
    Evaluate True
      When DFH-Initial
        Perform Initial-Activity
        Perform Order-Activity
        Perform Order-Response
        Perform Delivery-Activity
      When Delivery-Complete
        Perform Delivery-Response
        Perform Delivery-Confirmation
      When Delivery-Confirmed
        Perform Confirm-Response
        Perform Invoice-Activity
      When Invoice-Complete
        Perform Invoice-Response
        Perform Payment-Activity
      When Payment-Due
        Perform Payment-Due-Response
      When Payment-Complete
        Perform Payment-Response
      When Reminder-Expired
        Perform Reminder-Expired-Response
      When Reminder-Complete
        Perform Reminder-Response
      When Other
        .
    End Evaluate.
        .
    EXEC CICS RETURN END-EXEC
        .
Initial-Activity.
    .
    EXEC CICS ASSIGN PROCESS(Process-Name)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
Order-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Order')
                 TRANSID('SORD')
                 PROGRAM('ORD001')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Sale-Container)
                 ACTIVITY('Order') FROM(Process-Name)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS LINK ACTIVITY('Order')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 44. The SAL002 root activity program, including compensation actions (Part 2)

```
Order-Response.
    .
    EXEC CICS CHECK ACTIVITY('Order') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
    .
Delivery-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Delivery')
                 TRANSID('SDEL')
                 PROGRAM('DEL001')
                 EVENT('Delivry-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Order-Container)
                 ACTIVITY(Order-Container) INTO(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Order-Container)
                  ACTIVITY('Delivery') FROM(Order-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Delivery')
                 ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
Delivery-Response.
    .
    EXEC CICS CHECK ACTIVITY('Delivery') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
Delivery-Confirmation.
    .
    EXEC CICS DEFINE ACTIVITY('Confirm')
                 TRANSID('FCON')
                 PROGRAM('CON001')
                 EVENT('Delivry-Confirmd')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 45. The SAL002 root activity program, including compensation actions (Part 3)

```
    EXEC CICS GET CONTAINER(Deliver-Container)
                  ACTIVITY('Delivery') INTO(Delivery-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Deliver-Container)
                  ACTIVITY('Confirm') FROM(Delivery-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Confirm')
                 ASYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
    .
Confirm-Response.
    .
    EXEC CICS CHECK ACTIVITY('Confirm') COMPSTATUS(status)
              RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
Invoice-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Invoice')
                 TRANSID('SINV')
                 EVENT('Invoice-Complete')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Confirm-Container)
                 ACTIVITY('Confirm') INTO(Confirm-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Confirm-Container)
                 ACTIVITY('Invoice') FROM(Confirm-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Invoice')
                 ASYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 46. The SAL002 root activity program, including compensation actions (Part 4)

```
Invoice-Response.
    .
    EXEC CICS CHECK ACTIVITY('Invoice') COMPSTATUS(status)
             RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC NOT = DFHRESP(NORMAL)
        .
    End-If.
    .
    If status NOT = DFHVALUE(NORMAL)
        .
    End-If.
    .
Payment-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Payment')
                TRANSID('SPAY')
                EVENT('Payment-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE TIMER('Payment-Due')
                AFTER DAYS(7)
            RESP(data-area) RESP2(data-area) END-EXEC
    .
      EXEC CICS GET CONTAINER(Invoice-Container)
                ACTIVITY('Invoice') INTO(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Invoice-Container)
                ACTIVITY('Payment') FROM(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Payment')
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 47. The SAL002 root activity program, including compensation actions (Part 5)

```
Payment-Due-Response.
    .
    EXEC CICS DELETE TIMER('Payment-Due')
             RESP(RC) RESP2(data-area) END-EXEC
    .
    Perform Reminder-Activity
Payment-Response.
    .
    EXEC CICS CHECK ACTIVITY('Payment') COMPSTATUS(status)
              RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC = DFHRESP(NORMAL)
       If status = DFHVALUE(NORMAL)
          EXEC CICS DELETE TIMER('Payment-Due')
                    RESP(RC) RESP2(data-area) END-EXEC
          .
          Perform End-process
       Else
       .
       End-If
    Else
       .
    End-If
    .
Reminder-Activity.
    .
    EXEC CICS DEFINE ACTIVITY('Reminder')
                TRANSID('PAYR')
                EVENT('Remindr-Complete')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE TIMER('Remindr-Expired')
                AFTER DAYS(14)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS GET CONTAINER(Invoice-Container)
                ACTIVITY('Invoice') INTO(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Invoice-Container)
                ACTIVITY('Reminder') FROM(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Reminder')
                ASYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 48. The SAL002 root activity program, including compensation actions (Part 6)

```
Reminder-Expired-Response.
    .
    EXEC CICS DELETE TIMER('Remindr-Expired')
              RESP(RC) RESP2(data-area) END-EXEC
    .
    Perform Compensation
Reminder-Response.
    .
    EXEC CICS CHECK ACTIVITY('Reminder') COMPSTATUS(status)
              RESP(RC) RESP2(data-area) END-EXEC
    .
    If RC = DFHRESP(NORMAL)
       If status = DFHVALUE(NORMAL)
          EXEC CICS DELETE TIMER('Remindr-Expired')
                    RESP(RC) RESP2(data-area) END-EXEC
          .
          Perform End-process
       Else
          .
       End-If
    Else
       .
    End-If
    .
Compensation.
    .
    EXEC CICS DEFINE ACTIVITY('Payment-Compen')
                TRANSID('PAYC')
                PROGRAM('PEX001')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Invoice-Container)
                ACTIVITY('Payment-Compen') FROM(Invoice-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Payment-Compen')
                SYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE ACTIVITY('Confirm-Compen')
                TRANSID('CONC')
                PROGRAM('REQ001')
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Deliver-Container)
                ACTIVITY('Confirm-Compen') FROM(Delivery-Buffer)
             RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Confirm-Compen')
                SYNCHRONOUS
             RESP(data-area) RESP2(data-area) END-EXEC
    .
```

Figure 49. The SAL002 root activity program, including compensation actions (Part 7)

```
    EXEC CICS DEFINE ACTIVITY('Delivery-Compen')
                TRANSID('DELC')
                PROGRAM('RTN001')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Order-Container)
                ACTIVITY('Delivery-Compen') FROM(Order-Buffer)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Delivery-Compen')
                SYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS DEFINE ACTIVITY('Order-Compen')
                TRANSID('ORDC')
                PROGRAM('CAN001')
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS PUT CONTAINER(Sale-Container)
                ACTIVITY('Order-Compen') FROM(Process-Name)
              RESP(data-area) RESP2(data-area) END-EXEC
    .
    EXEC CICS RUN ACTIVITY('Order-Compen')
                SYNCHRONOUS
              RESP(data-area) RESP2(data-area) END-EXEC
    .
End-Process.
    .
    EXEC CICS RETURN ENDACTIVITY
              RESP(data-area) RESP2(data-area) END-EXEC
End Program.
```

Figure 50. The SAL002 root activity program, including compensation actions (Part 8)

Note the following:

- A reminder is set for the Payment activity:

  ```
  .
  EXEC CICS DEFINE TIMER('Payment-Due')
              AFTER DAYS(7)
            RESP(data-area) RESP2(data-area) END-EXEC
  .
  ```

  The DEFINE TIMER command defines a timer which will expire in one week. Because the EVENT option is not specified, the event associated with the timer—the timer event—is given the same name as the timer itself (*Payment-Due*). Now, SAL002 will be reattached when either of the following happens:

  1. The Payment activity completes. Because Payment is a user-related activity, it will complete only if a terminal user confirms that payment has been received.
  2. The timer expires.

- If SAL002 is invoked because the timer expires, it requests the Reminder activity to run. The Reminder activity too is user-related—the request to run it drives the first part of the activity, which sends a reminder letter to the customer, records the activity's details on a pending file, and waits to be reactivated by user input.

  As for the Payment activity, a timer is set for the Reminder activity. Now, SAL002 will be reattached when either of the following happens:

  1. The Reminder activity completes. Because Reminder is a user-related activity, it will complete only if a terminal user confirms that payment has been received.
  2. The timer expires.

- If SAL002 is next invoked because the timer expires, it compensates its completed child activities. For each child activity to be compensated, SAL002 defines a new (compensation) activity, provides the compensation activity with some input data, and runs it:

```
EXEC CICS DEFINE ACTIVITY('Payment-Compen')
             TRANSID('PAYC')
             PROGRAM('PEX001')
          RESP(data-area) RESP2(data-area) END-EXEC
.
EXEC CICS PUT CONTAINER(Invoice-Container)
             ACTIVITY('Payment-Compen') FROM(Invoice-Buffer)
          RESP(data-area) RESP2(data-area) END-EXEC
.
EXEC CICS RUN ACTIVITY('Payment-Compen')
             SYNCHRONOUS
          RESP(data-area) RESP2(data-area) END-EXEC
```

  Notice that the program used to execute the Payment-Compen compensation activity is different from that used for the Payment activity that is compensated. The PUT CONTAINER command provides the Payment-Compen activity with the same input data that was passed to the Payment activity.

  Table 5 shows which activities are compensated, and the actions taken by the compensation activity in each case.

Table 5. Compensation activities

Completed child activity	Compensation activity	Actions taken by compensation activity
Payment	Payment-Compen	Cancels the outstanding payment request
Confirm	Confirm-Compen	Sends a letter requesting return of goods
Delivery	Delivery-Compen	Requests confirmation that the goods have been returned
Order	Order-Compen	Cancels the original order request

- The user-defined timers (*Payment-Due* and *Remindr-Expired*) are deleted as soon as they are no longer required. This has the side effect of automatically deleting the timer events associated with them.
- The CHECK ACTIVITY command is used to check the response from each child activity. This has the side effect of automatically deleting the activity completion event, if the child has completed. (An activity must delete the completion events for all its child activities before it completes.)

  Note: In a real application, it would be necessary to issue CHECK ACTIVITY commands for the compensation activities. For the sake of brevity, these have been omitted from the example.

Dealing with application locking

When an activity completes, any updates that it has made to data are committed and the database manager releases its locks on the data. The updated data is then available to other activities, including activities that are part of other business applications.

About this task

These other activities may make decisions based on the state of the data. If you later compensate the completed activity and return the data to its previous state, some activities might have executed based on data which is no longer valid. If

these activities are part of the same process as the compensated activity, you can code your application to compensate them too. However, to cope with the possibility that activities in other applications might take decisions based on data that is later changed by compensation, your application must be coded differently.

If your applications include compensation activities which reverse previously committed data updates, they might need to include logic to provide logical record locking. The "application lock" does not need to be a hard lock preventing access to the data, but a flag which indicates that the data is part of an incomplete business process which might be reversed. All activities working with the "in-process" data could be coded to check this flag and then follow appropriate logic. To support this logic, when you design your database you need to include a "locked" field in your data records.

For example, you might have a "Welcome letter" application which scans the customer database for new customers who have placed their first order, and sends each a welcoming letter thanking them for their order and asking them to complete a customer satisfaction questionnaire. Perhaps your company considers it inappropriate to send such a letter if the order is not yet complete and payment received, because the welcome letter might be received along with less friendly letters demanding payment! Therefore, the Order activity of the Sale business application could set an *order-in-progress* flag on the order record, which would exclude the order from consideration by a "Welcome letter" process. Later, the Payment activity of the Sale application could unset the *order-in-progress* flag.

Chapter 9. Reusing existing 3270 applications in BTS

You can use BTS support for the 3270 bridge to integrate existing transactions, including more complex ones, into BTS applications. Sample programs show you how the integration works.

The 3270 bridge is described in Bridging to 3270 transactions in the External Interfaces Guide.

Running a 3270 transaction from BTS

BTS supports the 3270 bridge function. Therefore, BTS applications can be integrated with, and make use of, existing 3270-based applications. Follow this basic mechanism for running a 3270 transaction from a BTS application.

About this task

Even though BTS activities are not terminal-related (they are never started directly from a terminal), a BTS activity can be implemented by a 3270-based transaction. The bridge exit program is used to put a "BTS wrapper" around the original 3270 transaction.

Figure 51 shows the basic mechanism for running a 3270 transaction from a BTS application.

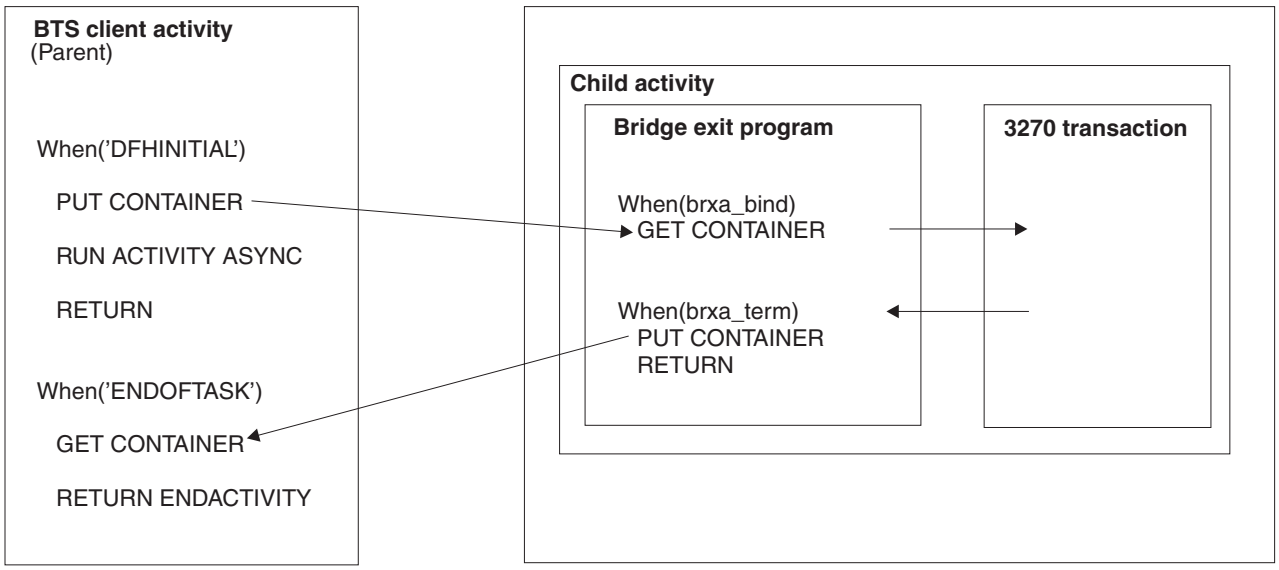


Figure 51. Running a 3270-based transaction as a BTS activity

1. A BTS activity, known in bridge terminology as the **client**, creates data to run a 3270 transaction. It puts the data in a container associated with a child activity.
2. The client runs the child activity—which is implemented by the 3270 transaction—asynchronously.
3. The BTS XM client identifies that the transaction should use the 3270 bridge and calls the bridge XM client.

© Copyright IBM Corp. 1999, 2011

4. On the 'bind' call to the bridge exit, the bridge exit program issues a GET CONTAINER command to retrieve the data to run the 3270 transaction.

   Note: In a bridge environment, the bridge exit program becomes part of the 3270 transaction. Thus, the exit program does not need to acquire the child activity before issuing the GET CONTAINER command—it is itself *part* of the child activity.
5. The 3270 transaction is run using the retrieved data. Any output data it produces is saved in an output message.
6. When the bridge exit program is invoked for termination of the 3270 transaction, it issues:
   a. A PUT CONTAINER command, to put the output message into a named data-container
   b. A RETURN command, which causes the child activity to complete.
7. The firing of the completion event of the child activity causes the parent (client) activity to be reactivated.
8. The client issues a GET CONTAINER command to retrieve the output from the 3270 transaction.

The following table contains example pseudocode for running a 3270-based transaction as a BTS activity.

Table 6. Pseudocode for running a 3270-based transaction as a BTS activity

Client activity	Bridge exit program

When DFH-Initial
  encode msg-in-buffer
  EXEC CICS DEFINE ACTIVITY ('3270-act')
  TRANSID('T327') EVENT('3270-Complete')
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS PUT CONTAINER('Message')
  ACTIVITY('3270-act') FROM(msg-in-buffer)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS RUN ACTIVITY('3270-act')
  ASYNCHRONOUS
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS RETURN END-EXEC
  .
  .
When 3270-Complete
  EXEC CICS GET CONTAINER('Message')
  ACTIVITY('3270-act') INTO(msg-out-buffer)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  decode msg-out-buffer
  EXEC CICS RETURN ENDACTIVITY
	Init.
	  pass userdata from the brdata to BRXA
	  .
	  .
	Bind.
	  EXEC CICS GET CONTAINER('Message')
	  INTO(3270-msg-in-buffer)
	  RESP(data-area) RESP2(data-area) END-EXEC
	  .
	  .
	Term.
	  EXEC CICS PUT CONTAINER('Message')
	  FROM(3270-msg-out-buffer)
	  RESP(data-area) RESP2(data-area) END-EXEC
	  .
	  EXEC CICS RETURN END-EXEC

The child activity is implemented by the 3270 transaction and the bridge exit program. All the required BTS commands are issued by the exit program.

Resource definition

To enable BTS 3270 bridge support, you must specify the name of a bridge exit program on the BREXIT option of the TRANSACTION definition for the 3270 transaction that you want to run.

If two or more bridge transport mechanisms require the BREXIT parameter to be specified on the transaction definition, you can use an alias transaction definition. For information about how other bridge transport mechanisms support specification of the BREXIT parameter, see the *CICS External Interfaces Guide*.

Running more complex transactions

You can run more complex 3270 transactions that produce intermediate messages, are conversational in design, or are pseudoconversational.

The basic mechanism described in "Running a 3270 transaction from BTS" on page 99 assumes a straightforward, "one shot" transaction, where the 3270 transaction does an **EXEC CICS RECEIVE MAP**, followed by one or more **EXEC CICS SEND MAP** requests, and ends with an **EXEC CICS RETURN**. In practice, things are not always so simple.

The topics in this section describe how to run various complex transactions:

Intermediate output messages

For a non-conversational 3270 transaction, the bridge exit program might be called to write an intermediate message when either the 3270 transaction has specified WAIT on the EXEC CICS SEND command or the output message buffer is full.

Under some bridge transport mechanisms, it makes sense for the bridge exit program to write an intermediate message containing the data so far. However, under BTS there is no point in trying to send an intermediate message back to the user.

If the exit program is called because of the WAIT option, it can do nothing and return.

If the exit program is called because the message buffer is full, it should:

1. Obtain a new, larger output buffer (by issuing a GETMAIN command).
2. Copy the contents of the original buffer into the new buffer.
3. Release the original buffer (by issuing a FREEMAIN command).

Using this approach, all output from the 3270 transaction is sent to the client at transaction end.

The sample bridge exit program, DFH0CBAE (see "Sample programs" on page 106) obtains all the storage it requires - including the storage for its output buffer - at the same time. It saves the address of the output buffer in field BRXA-OUTPUT-MESSAGE-PTR of the bridge exit area (BRXA) user area. We recommend that your exit programs do the same.

Note: When the exit program is called because the output buffer is full, field BRXA-FMT-RESPONSE of the BRXA is set to BRXA-FMT-OUTPUT-BUFFER-FULL. The current size of the storage is in field BRXA-OUTPUT-MESSAGE-LEN.

Conversational transactions

Follow this pseudocode to see how to run a conversational 3270 transaction. Potential problems with solutions are illustrated.

A potential problem is that, at one or more stages, the 3270 transaction requires further data to continue. The bridge exit program cannot obtain this data from the client. That is, it cannot end its current activation, to be reactivated with the required data—because the 3270 transaction has not completed, issuing an EXEC CICS RETURN command would merely return control to the latter. Nor can the exit program get information back to the client by issuing an EXEC CICS SYNCPOINT command, because this would modify the 3270 transaction.

One solution is for the bridge exit program itself to obtain, or compute, the required data. Perhaps a better solution is for the exit program to create a subtask to obtain the data. It could, for example, create a separate child activity (a grandchild of the client) to deal with each request for data - each intermediate map - sent by the 3270 transaction. For convenience, we shall refer to such child activities as "conversational activities". Figure 52 illustrates this approach.

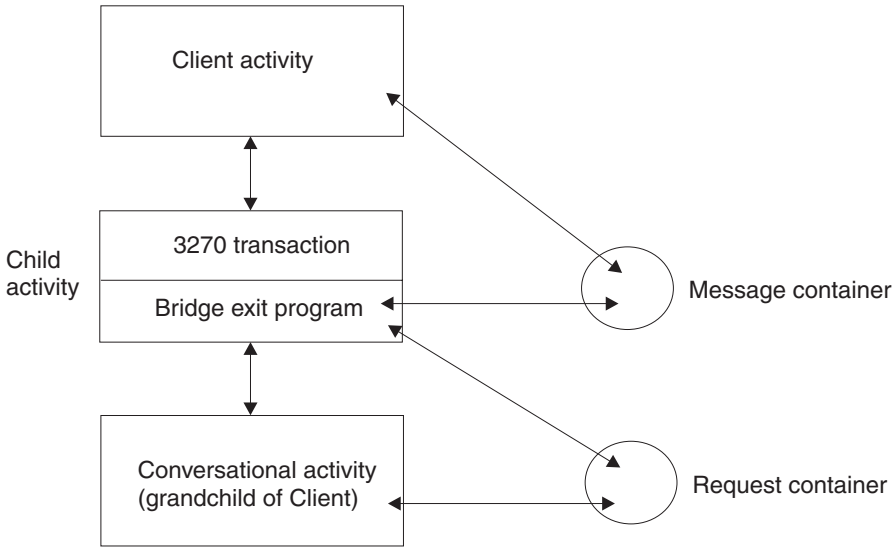


Figure 52. Running a 3270 conversational transaction as a BTS activity. The bridge exit program creates a child activity to deal with each map sent by the 3270 transaction.

One possible problem of creating a separate activity to deal with each intermediate map is that the output message sent to the client by the exit program at transaction end contains only the final 3270 map. If it is important that intermediate messages are preserved, the conversational activities could put them in other containers associated with the client.

The following table contains example pseudocode for running a 3270 conversational transaction.

Table 7. Pseudocode for running a 3270 conversational transaction as a BTS activity. The bridge exit program creates a child activity to deal with each map sent by the 3270 transaction.

Bridge exit program	"Conversational" activity

Read_Message.
  encode conv-in-buffer from 3270-msg-out-buffer
  EXEC CICS DEFINE ACTIVITY (next-conv-act-name)
  TRANSID(conv-transaction-id)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS PUT CONTAINER('Request')
  ACTIVITY(next-conv-act-name)
  FROM(conv-in-buffer)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS LINK ACTIVITY(next-conv-act-name)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS CHECK ACTIVITY(next-conv-act-name)
  COMPSTATUS(status) ABCODE(a)
  RESP(data-area) RESP2(data-area) END-EXEC
  If status NOT = DFHVALUE(NORMAL)
  EXEC CICS ABEND ABCODE(a)
  NODUMP
  RESP(data-area) RESP2(data-area) END-EXEC
  End-If.
  .
  EXEC CICS GET CONTAINER('Request')
  ACTIVITY(next-conv-act-name)
  INTO(3270-msg-in-buffer)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  .
Write_Message.
  Intermediate writes cannot be sent
  to the client.
  EXEC CICS NOOP
  RESP(data-area) RESP2(data-area) END-EXEC
  .
	WHEN DFH-Initial
	  EXEC CICS GET CONTAINER('Request')
	  INTO(msg-in-buffer)
	  RESP(data-area) RESP2(data-area) END-EXEC
	  .
	  decode msg-in-buffer
	  encode msg-out-buffer
	  .
	  EXEC CICS PUT CONTAINER('Request')
	  FROM(msg-out-buffer)
	  RESP(data-area) RESP2(data-area) END-EXEC
	  .
	  EXEC CICS RETURN END-EXEC

The exit program issues a LINK ACTIVITY, rather than a RUN ACTIVITY SYNCHRONOUS, command to activate the "conversational" child activity. This is necessary to ensure that the child executes in the same unit of work as the exit program.

Pseudoconversational transactions

Follow this pseudocode to see how to run a pseudoconversational 3270 transaction.

A pseudoconversation is indicated by the fact that the output data returned to the client by the exit program contains a bridge facility token (and possibly a next-transaction ID). It is the responsibility of the client to check the appropriate field in the output message and to start the next transaction.

The example shown here contains pseudocode for running a 3270 pseudoconversational transaction.

Table 8. Pseudocode for running a 3270 pseudoconversational transaction as a BTS activity

Client activity	Bridge exit program

When DFH-Initial
  encode msg-in-buffer
  EXEC CICS DEFINE ACTIVITY (3270-act-name)
  TRANSID(transaction-id) EVENT(3270-Complete)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS PUT CONTAINER('Message')
  ACTIVITY(3270-act-name) FROM(msg-in-buffer)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS RUN ACTIVITY(3270-act-name)
  ASYNCHRONOUS
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS RETURN END-EXEC
  .
  .
When 3270-Complete
  EXEC CICS CHECK ACTIVITY(3270-act-name)
  COMPSTATUS(status) ABCODE(a)
  RESP(data-area) RESP2(data-area) END-EXEC
  If status NOT = DFHVALUE(NORMAL)
  EXEC CICS ABEND ABCODE(a)
  NODUMP
  RESP(data-area) RESP2(data-area) END-EXEC
  End-If.
  .
  EXEC CICS GET CONTAINER('Message')
  ACTIVITY(3270-act-name) INTO(msg-out-buffer)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  decode msg-out-buffer
  If mqcih-facility = blank
  EXEC CICS RETURN ENDACTIVITY END-EXEC
  Else
  encode msg-in-buffer
  EXEC CICS DEFINE ACTIVITY (3270-act-name)
  TRANSID(next-transaction-id)
  EVENT(3270-Complete)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS PUT CONTAINER('Message')
  ACTIVITY(3270-act-name)
  FROM(msg-in-buffer)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS RUN ACTIVITY(3270-act-name)
  ASYNCHRONOUS
  FACILITYTOKN(8-byte token)
  RESP(data-area) RESP2(data-area) END-EXEC
  .
  EXEC CICS RETURN END-EXEC
  End-If.
	Init.
	  pass userdata from the brdata to BRXA
	  .
	  .
	Bind.
	  EXEC CICS GET CONTAINER('Message')
	  INTO(3270-msg-in-buffer)
	  RESP(data-area) RESP2(data-area) END-EXEC
	  .
	  .
	Term.
	  EXEC CICS PUT CONTAINER('Message')
	  FROM(3270-msg-out-buffer)
	  RESP(data-area) RESP2(data-area) END-EXEC
	  .
	  EXEC CICS RETURN END-EXEC

Note that:

- The client starts each transaction in the pseudoconversation by defining and running a new child activity, rather than by reactivating the same child activity with a different input event. This is necessary, in case the next-transaction IDs returned by the 3270 application are different—that is, in case each step of the

pseudoconversation is implemented by a differently named transaction. (The variable *next-transaction-id* is used to name the transaction that implements each new child activity.)

- In this example, the variable *3270-act-name* is used to name each child activity differently. An alternative approach might be to delete the completed child activity before redefining it with a different TRANSID.
- In this example, the variable *3270-Complete* is used to name each activity completion event differently. This is not strictly necessary, because if the previous child activity completed normally its completion event is deleted from the event pool of the client following the CHECK ACTIVITY command.
- The output message returned by the bridge exit program should contain an 8-byte token representing the bridge facility. So that the bridge facility is reused for the next transaction in the pseudoconversation, the client uses the FACILITYTOKN option of the RUN ACTIVITY command to pass the token to the next child activity.

Transaction routing of pseudoconversations

The 3270 bridge does not support transaction routing of pseudoconversations. If a 3270 transaction is pseudoconversational, and is started from BTS, all of its constituent transactions must run in the same CICS region. Understand the options available in this situation.

If one of the transactions is routed to a different region, an ABRH abend occurs. One way to ensure that all the transactions run in the same region is for the client to run the child activities synchronously. Activities that are run synchronously always run in the local region - they are never routed.

However, although all the transactions in a pseudoconversation have to run in the same region, they do not have to run in the same region as the client; nor do they have to run in a specific region. If you use CICSPlex SM for routing purposes, you can define all the 3270 transactions in a pseudoconversation as part of the same transaction group.

Defining all the 3270 transactions in a pseudoconversation as part of the same transaction group gives you two options:

1. You can define the transaction group to run on a specific named region.
2. You can define the transaction group to run on whichever region the first transaction within a BTS process runs on. This option is preferred.

Using timers

You can use timers to avoid indefinite waits for a 3270 transaction to reply.

To avoid indefinite waits for a 3270 transaction to reply, the client could set a timer. If the timer expires, the client is reactivated and assumes that an error has occurred. The client can cancel the 3270 transaction, by issuing a CANCEL ACTIVITY command if the activity has not started, or by using a SET TASK PURGE command if it has.

Abend processing

You can use this pseudocode example to process your 3270 transaction abends.

If the 3270 transaction ends abnormally, an abend call is made to the bridge exit. This call occurs at the end of the transaction - it cannot be used to implement an abend handler.

If it is necessary for the exit program to reply to the client, it cannot do so by issuing a PUT CONTAINER command. Because BTS activities are always recoverable, the command would be backed out. One solution is for the exit program to write a message to an unrecoverable transient data or temporary storage queue. It could, for example, delegate this task to a child activity.

The following example contains pseudocode for dealing with an abend of the 3270 transaction. The *Requestor* activity is a child of the bridge exit; it handles the abend.

Table 9. Pseudocode for dealing with an abend of the 3270 transaction

Bridge exit program	Requestor activity
Abend. encode abend-in-buffer from 3270-msg-out-buffer EXEC CICS DEFINE ACTIVITY ('Requestor') TRANSID('ABE1') RESP(data-area) RESP2(data-area) END-EXEC . EXEC CICS PUT CONTAINER('Abend') ACTIVITY('Requestor') FROM(abend-in-buffer) RESP(data-area) RESP2(data-area) END-EXEC . EXEC CICS LINK ACTIVITY('Requestor') RESP(data-area) RESP2(data-area) END-EXEC . EXEC CICS CHECK ACTIVITY('Requestor') COMPSTATUS(status) ABCODE(a) RESP(data-area) RESP2(data-area) END-EXEC If status NOT = DFHVALUE(NORMAL) EXEC CICS ABEND ABCODE(a) NODUMP RESP(data-area) RESP2(data-area) END-EXEC . End-If. . EXEC CICS RETURN END-EXEC	WHEN DFH-Initial EXEC CICS GET CONTAINER('Abend') INTO(msg-in-buffer) RESP(data-area) RESP2(data-area) END-EXEC . decode msg-in-buffer output a message to a non-recoverable TD or TS queue . EXEC CICS RETURN END-EXEC

The exit program issues a LINK ACTIVITY, rather than a RUN ACTIVITY SYNCHRONOUS, command to activate the *Requestor* activity. This is necessary because the child must execute in the same unit of work as the exit program.

Transaction restart

The 3270 bridge does not support transaction restart. If a client activity is restarted and tries to reuse a bridge facility token, an ABRH abend occurs.

Sample programs

CICS supplies sample programs that demonstrate how to integrate 3270-based transactions into BTS applications.

The samples are:

- DFH0CBAC, a client activity program

- DFH0CBAE, a bridge exit program
- DFH0CBAI, creates an input message
- DFH0CBAO, processes an output message.

The samples are supplied, in COBOL source code, in the SDFHSAMP library. They contain explanatory comments. Like the pseudocode examples in this chapter, the samples use containers named `Message`, `Request`, and `Abend`.

Note: To use the samples, you will also need to compile the 3270 bridge formatter program, DFH0CBRF.

Sample resource definitions are in RDO group DFH$BABR.

The sample programs are compatible with the 3270 bridge support pack, CA1E. The BTS passthrough transaction is BRCB.

Part 3. Administering CICS business transaction services

You can define and control the CICS business transaction services environment using system definitions, resource definitions, security in BTS, operator commands, and sysplex-enabled BTS functions.

Table 10. Administration road map

If you want to...	Refer to...
• Define BTS data sets to CICS • Specify the program used to route BTS activities dynamically around a BTS-set	Chapter 10, "System definition for BTS," on page 111
Define BTS process-types to CICS	Chapter 11, "Resource definition for BTS," on page 117
Make your BTS processes and activities secure	Chapter 12, "Security in BTS," on page 123
Use CEMT commands to control BTS resources	Chapter 13, "BTS operator commands," on page 133
Learn how to operate BTS in a sysplex	Chapter 14, "Using BTS in a sysplex," on page 149

© Copyright IBM Corp. 1999, 2011

Chapter 10. System definition for BTS

You can use system definitions to define your BTS data sets and to name your routing program.

This chapter contains:

- "Defining BTS data sets"
- "Naming the routing program" on page 115.

Defining BTS data sets

You can use the IDCAMS program to define your BTS data sets to MVS™.

About this task

You must define two types of BTS data set:

- Repository data sets.
- A local request queue data set.

Repository data sets

When a process is not running under the control of the CICS business transaction services domain, its state and the states of its constituent activities are preserved by being written to a VSAM data set known as a *repository*.

The states of all processes of a particular process-type (and of their activity instances) are stored on the same repository data set. Records for multiple process-types can be written to the same repository. You specify the repository on which processes of a particular process-type are stored when you define the process-type - see "CEDA DEFINE PROCESSTYPE" on page 119.

You must define at least one BTS repository data set to MVS. You might decide to define more than one, assigning a different set of process-types to each. One reason for defining more than one BTS repository might be storage efficiency - perhaps some of your process-types tend to produce longer records than others. To enable you to distinguish between process-types during a browse, you do not need to assign each process-type to a separate repository.

If you operate BTS in a sysplex, several CICS regions might share access to one or more repository data sets. This enables requests for the processes and activities stored on the data sets to be routed across the participating regions - see Chapter 14, "Using BTS in a sysplex," on page 149.

You must define the repository file as recoverable. Specify the following parameters to IDCAMS:

INDEXED
BTS repository data sets must be in KSDS format.

KEYS(50 0)
The file key. The file key is 50 bytes in length and is located at offset X'0' in the record.

© Copyright IBM Corp. 1999, 2011

LOG(UNDO|ALL)
The recovery options for the data set:

UNDO
The data set is recoverable.

ALL Forward recovery is required. If you specify LOG(ALL), you must also specify a log stream on the **LOGSTREAMID** parameter.

LOGSTREAMID(*log_stream_ID***)**
The identifier of the log stream to which forward recovery records are to be written. This parameter is required only if you specify LOG(ALL).

RECORDSIZE(*average maximum***)**
The average and maximum size of records on the data set, in bytes.

Specify *maximum* as 16384 bytes. CICS automatically splits any records that are larger than 16 KB.

It is difficult to predict the average size of repository records. (A notional record of 20000 bytes, for example, is split into one record of 16384 bytes and one of 3616 bytes.) Initially, specify *average* as 8 KB. If you find in practice that the average size of records is markedly different from this average, you can specify a different value.

SPANNED
A single record can span control intervals. Specify this parameter if the record size is larger than the CI.

Figure 53 shows example JCL for defining a BTS repository data set. This JCL is for illustration only.

```
//SMITHGOT JOB (WINVMC,SMITH),CLASS=E,USER=username
//IJMRBTS EXEC PGM=IDCAMS,REGION=6144K
//SYSPRINT DD SYSOUT=A
//AMSDUMP  DD SYSOUT=A
//SYSIN    DD *
  DELETE ('CICSTS42.CICS.BTS') PURGE CLUSTER
  DEFINE CLUSTER (                        -
                NAME( CICSTS42.CICS.BTS )  -
                LOG(UNDO)                  -
                CYL(2,1)                   -
                CISZ(4096)                 -
                SPANNED                    -
                VOLUMES (P2DA62)           -
                KEYS( 50 0 )               -
                INDEXED                    -
                RECORDSIZE(8192  16384 )   -
                FREESPACE( 5 5 )           -
                SHAREOPTIONS( 2 3 )        -
                )                          -
  INDEX         (                          -
                NAME( CICSTS42.CICS.BTS.INDEX )   -
                )                          -
  DATA          (                          -
                NAME( CICSTS42.CICS.BTS.DATA )    -
                )
/*
//
```

Figure 53. Example JCL for defining a BTS repository data set

To ensure that your repositories are continuously available, you are recommended to define them to use the backup while open (BWO) facility provided by

DFSMSdss and DFSMShsm. For details of BWO, and how to define VSAM data sets to use it, see the *Backup while open (BWO) of VSAM files* section in the *CICS System Definition Guide*.

Local request queue data set

The local request queue data set stores pending BTS requests; for example, timer requests or requests to run activities. It is recoverable and is used to ensure that, if CICS fails, no pending requests are lost.

Requests that CICS can run immediately, for example, requests to run activities, are stored on the data set only briefly. Requests that CICS cannot run immediately, for example, timer or unserviceable requests, might be stored for longer periods. When CICS has processed a request, the request is deleted from the data set.

The local request queue data set differs from repository data sets:

- It is a mandatory CICS data set. You must define one, even if you do not use BTS.

  Note: Procedure DFHDEFDS in library SDFHINST contains a definition of the LRQ. For information about how to use DFHDEFDS, see the *Creating data sets unique to each CICS region, DFHDEFDS job* chapter in the *CICS Transaction Server for z/OS Installation Guide*.
- You must define one, and only one, to each CICS region.
- It is never shared. The local request queue data set relates solely to requests generated on the local region.

Specify the following parameters to IDCAMS:

INDEXED
BTS local request queue data sets must be in KSDS format.

KEYS(40 0)
The file key. The file key is 40 bytes in length and is located at offset X'0' in the record.

LOG(UNDO|ALL)
The recovery options for the data set:

UNDO
The data set is recoverable.

ALL Forward recovery is required. If you specify LOG(ALL), you must also specify a log stream on the LOGSTREAMID parameter.

LOGSTREAMID(*log_stream_ID***)**
The identifier of the log stream to which forward recovery records are to be written. This parameter is required only if you specify LOG(ALL).

RECORDSIZE(*average maximum***)**
The average and maximum size of records on the data set, in bytes.

Specify *average* as 2024 bytes and *maximum* as 2024 bytes.

Figure 54 on page 114 shows example JCL for defining a BTS local request queue data set.

```
//SMITHGOT JOB (WINVMC,SMITH),CLASS=E,USER=username
//IJMRLRQ  EXEC PGM=IDCAMS,REGION=6144K
//SYSPRINT DD SYSOUT=A
//AMSDUMP  DD SYSOUT=A
//SYSIN    DD *
  DEFINE CLUSTER (                        -
                NAME( CICSTS42.CICS.LRQ )  -
                LOG(UNDO)                  -
                CYL(2,1)                   -
                VOLUME (SYSDAV)            -
                KEYS( 40 0 )               -
                INDEXED                    -
                RECORDSIZE( 2024 2024 )    -
                FREESPACE( 0 10 )          -
                SHAREOPTIONS( 2 3 )        -
                )                          -
  DATA          (                          -
                NAME( CICSTS42.CICS.LRQ.DATA )    -
                CISZ(2048) -
                ) -
  INDEX         ( -
                NAME( CICSTS42.CICS.LRQ.INDEX )   -
                )                          -
/*
//
```

Figure 54. Example JCL for defining a BTS local request queue data set. The numbers are for illustration only.

The LRQ data set is critical to the operation of BTS. Because its loss could severely impact the progression of BTS activities, you should consider defining it to use backup while open (BWO) and forward recovery.

Note: For information about how to define the LRQ file to the CICS CSD, see "Defining the LRQ file to the CSD" on page 117.

Local request queue data set space allocation

The space that you allocate for the LRQ data set depends on how often you plan to trim the file and the amount of BTS work likely to be undertaken in the periods between maintenance.

Each **RUN ASYNCHRONOUS** or **DEFINE TIMER** request causes a record to be written to the LRQ file, each record being 2 KB in length.

You must allow plenty of space for contingencies. Remember that records representing timers that are defined to expire far in the future will remain on the data set until their timers expire. Even after the records have expired, VSAM still maintains control interval key ranges for their key values, even though the records for these keys have been deleted by CICS. These key ranges are never reused because they represent time in the past, and their control interval space can only be recovered by periodic maintenance of the data set.

For more information about maintaining your local request queue data set, see "Local request queue data set maintenance."

Local request queue data set maintenance

You must regularly perform maintenance on the local request queue (LRQ) data set to prevent the data set from becoming too large.

The key of the LRQ data set includes a time stamp. This means that BTS generally adds requests to the end of the file. Although BTS deletes requests when they have

completed, VSAM does not physically delete the records; so the space is not reused. Therefore, you must perform regular maintenance on the local request queue (LRQ) data set to prevent the data set from becoming too large.

You can use an AMS command such as REPRO to shrink the data set. For example, to remove the records that have been logically deleted by VSAM. Before using the REPRO command, you must quiesce CICS. Disabling the LRQ file definition is not sufficient - if you only disable the LRQ file definition, activations might fail with ASP7 abends.

Before undertaking maintenance, quiesce the CICS region. If you use CICSPlex SM for routing BTS activities, you could route work away from the region by altering a CICSPlex SM workload definition (WLMDEF). If you use a CICS distributed routing program, you could alter your routing program. Although the region has been temporarily quiesced, BTS operations in the sysplex are uninterrupted.

When reorganizing the data set, consider the percentage of control areas specified on the FREESPACE parameter on the cluster definition. If there are many records representing timers far in the future, and new timers are to be added for events that will expire earlier than these existing records, a large FREESPACE percentage will decrease the number of likely control area splits that might occur. If most new timer events are added for times later than existing events, the FREESPACE percentage for control areas can be set to 0.

Naming the routing program

If you are using BTS in a sysplex, you must name the routing program that will be used to dynamically route BTS activities around the BTS-set. Use the DSRTPGM system initialization parameter to name the routing program that you will use.

Table 11. The DSRTPGM system initialization parameter

DFHSIT	[TYPE={**CSECT** | DSECT}]
	.
	.
	.
	[,DSRTPGM={**NONE** | DFHDSRP | program-name | EYU9XLOP}]
	.
	.
	.
END	DFHSITBA

DSRTPGM={NONE|DFHDSRP|program-name|EYU9XLOP}
specifies the name of the distributed routing program to be used for dynamically routing:

- Eligible CICS business transaction services (BTS) processes and activities.

  For information about which BTS processes and activities are eligible for dynamic routing, see "Which BTS activities can be dynamically routed?" on page 150.
- Eligible non-terminal-related EXEC CICS START requests.

  For information about which non-terminal-related START requests are eligible for dynamic routing, see Non-terminal-related START commands the *Non-terminal related START commands* chapter in the *CICS Intercommunication Guide*.

DFHDSRP
The CICS sample distributed routing program.

EYU9XLOP
The CICSPlex SM routing program.

NONE
For eligible BTS processes and activities, no routing program is invoked. BTS processes and activities cannot be dynamically routed.

For eligible non-terminal-related START requests, the CICS sample distributed routing program, DFHDSRP, is invoked.

program-name
The name of a user-written program.

Note: See also the DTRPGM parameter, used to name the dynamic routing program.

Related concepts

Chapter 14, “Using BTS in a sysplex,” on page 149
You can operate BTS in a single CICS region. However, CICS business transaction services are sysplex-enabled. For example, you can use workload separation to ensure that processes of the same process-type are handled by a particular set of regions, and you can use workload routing to route activity requests across a set of regions.

Chapter 11. Resource definition for BTS

You can use BTS API commands to define most BTS resources such as processes, activities, events, and containers at run time. However, you must define some BTS resources on the CICS system definition file (CSD).

The only BTS resources that must be defined on the CICS system definition file (CSD) are:

Process-types
See "CEDA DEFINE PROCESSTYPE" on page 119.

Note: As an alternative to using RDO CEDA DEFINE PROCESSTYPE commands to define your process-types, you can use the CICSPlex SM Business Application Services (BAS) PROCDEF object. You might want to use the PROCDEF object if you are using BTS in a sysplex, with routing of processes and activities controlled by CICSPlex SM. For information about BAS, see the *CICSPlex System Manager Managing Business Applications*.

The BTS data set files
The CICS files that relate to the physical VSAM data sets used by BTS must be defined to CICS file control in the standard way, as described in FILE attributes the *File definition attributes* chapter in the *CICS Resource Definition Guide*.

BTS uses two kinds of data set:

`Local request queue data set`
is described in "Local request queue data set" on page 113.

For information about how to define the LRQ to the CSD, see "Defining the LRQ file to the CSD."

`Repository data sets`
on which process and activity records are stored, are described in "Repository data sets" on page 111.

For information about how to define repository files to the CSD, see "Defining repository files to the CSD" on page 118.

Audit logs
The journals used for auditing purposes must be defined to the CICS log manager in the standard way, as described in JOURNALMODEL attributes the *JOURNALMODEL definition attributes* in the*CICS Resource Definition Guide*.

Defining the LRQ file to the CSD

You can use the default FILE definition, DFHLRQ, in the CICS-supplied RDO group DFHCBTS, to define your local request queue (LRQ) data set. For performance reasons, you might want to change the default definition.

About this task

DFHCBTS is included in the default CICS startup group list, DFHLIST. Figure 55 on page 118 shows the default definition.

© Copyright IBM Corp. 1999, 2011

```
DEFINE   FILE(DFHLRQ)   GROUP(DFHCBTS)
DESCRIPTION(Scheduler Services - Local Request Queue)
         RLSACCESS(NO)              LSRPOOLNUM(1)
         READINTEG(UNCOMMITTED)     DSNSHARING(ACCREQS)
         STRINGS(10)                STATUS(ENABLED)
         OPENTIME(FIRSTREF)         DISPOSITION(OLD)
         DATABUFFERS(11)            INDEXBUFFERS(10)
         TABLE(NO)                  RECORDFORMAT(V)
         ADD(YES)                   BROWSE(YES)
         DELETE(YES)                READ(YES)
         UPDATE(YES)                JOURNAL(NO)
         JNLREAD(NONE)              JNLSYNCREAD(NO)
         JNLUPDATE(NONE)            JNLADD(NONE)
         JNLSYNCWRITE(NONE)         RECOVERY(BACKOUTONLY)
         FWDRECOVLOG(NO)            BACKUPTYPE(STATIC)
```

Figure 55. The supplied, default, FILE definition for the LRQ

For example, because the LRQ is frequently accessed it might be sensible to put the file in its own LSR pool, or to define it to use VSAM nonshared resources. By doing this, BTS operations are not affected by other file accesses.

If it is possible that, in your BTS system, many activities might be initiated at the same time (consider a banking application that produces a statement for each customer at the same time each month), it might be a good idea to set the number of strings and buffers to the value of the MXT system initialization parameter. Doing so avoids string and buffer waits. (You could also use TRANCLASS definitions to further throttle the number of concurrent transaction instances and thus LRQ file accesses.)

To alter the definition of DFHLRQ, you can do either of the following actions:

- Edit the supplied default FILE definition in the DFHCBTS group, and cold- or initial-start your CICS region. (The DFHCBTS group is supplied by IBM but you are not prevented from modifying it.)

  If you use this method, it is good practice to take a backup copy of the definition after you have altered it, to prevent your changes being lost if maintenance is applied to the CSD.
- Create your own FILE definition for DFHLRQ in a CSD group other than DFHCBTS, and cold- or initial-start your CICS region. Your CSD group must be installed *before* the DFHCBTS group. (On startup, when CICS encounters the definition in the DFHCBTS group after your own definition has been installed, it issues message DFHAM4803 as a warning.)

Note: To change the operational attributes of the DFHLRQ file with SPI commands while CICS is running, you must first close the file, which might affect BTS operation. Not all attributes can be changed by SPI commands.

Defining repository files to the CSD

You can define repository files to the CSD by specifying permitted operations, STRINGS values, VSAM record-level sharing (RLS) access, and any remote systems that share repository data.

About this task

On the FILE definition that defines the repository file to CICS:

- Specify that ADD, BROWSE, DELETE, READ, and UPDATE operations are all permitted.
- Specify the value of the STRINGS option to reflect the likely number of concurrent activations of processes that use the repository. The default is STRINGS(1), which is unlikely to be high enough.
- If you are using VSAM record-level sharing (RLS) to share the repository data between the regions of a BTS-set, specify RLSACCESS(YES).

  If you are using function-shipping to a file-owning region (FOR) to share the repository data between the regions of a BTS-set, specify REMOTESYSTEM(name_of_FOR).

  For information about BTS-sets, see Chapter 14, "Using BTS in a sysplex," on page 149.

CEDA DEFINE PROCESSTYPE

You can use the CICS business transaction services (BTS) CEDA DEFINE command to define and execute complex business applications called *processes*.

A process is represented in memory as a block of storage containing information relevant to its execution. It also has associated with it at least one additional block of information called an activity instance. When not executing under the control of the CICS business transaction services domain, a process and its activity instances are written to a data set known as a *repository*.

You can categorize your BTS processes by assigning them to different process-types. This categorization is useful, for example, for browsing purposes. The activities that constitute a process are of the same process-type as the process itself. A PROCESSTYPE definition defines a BTS process-type. The definition names the CICS file which relates to the physical VSAM data set (repository) on which details of all processes of this type (and their activity instances) are to be stored.

You might want to record the progress of BTS processes and activities for audit purposes, and to help diagnose errors in BTS applications. If so, you can name the CICS journal to which audit records are to be written, and the level of auditing that is required, for processes of the specified type.

Figure 56 on page 120 shows the relationship between PROCESSTYPE definitions, FILE definitions, and BTS data sets. Notice that multiple PROCESSTYPE definitions can reference the same FILE definition; and that multiple FILE definitions can reference the same BTS data set.

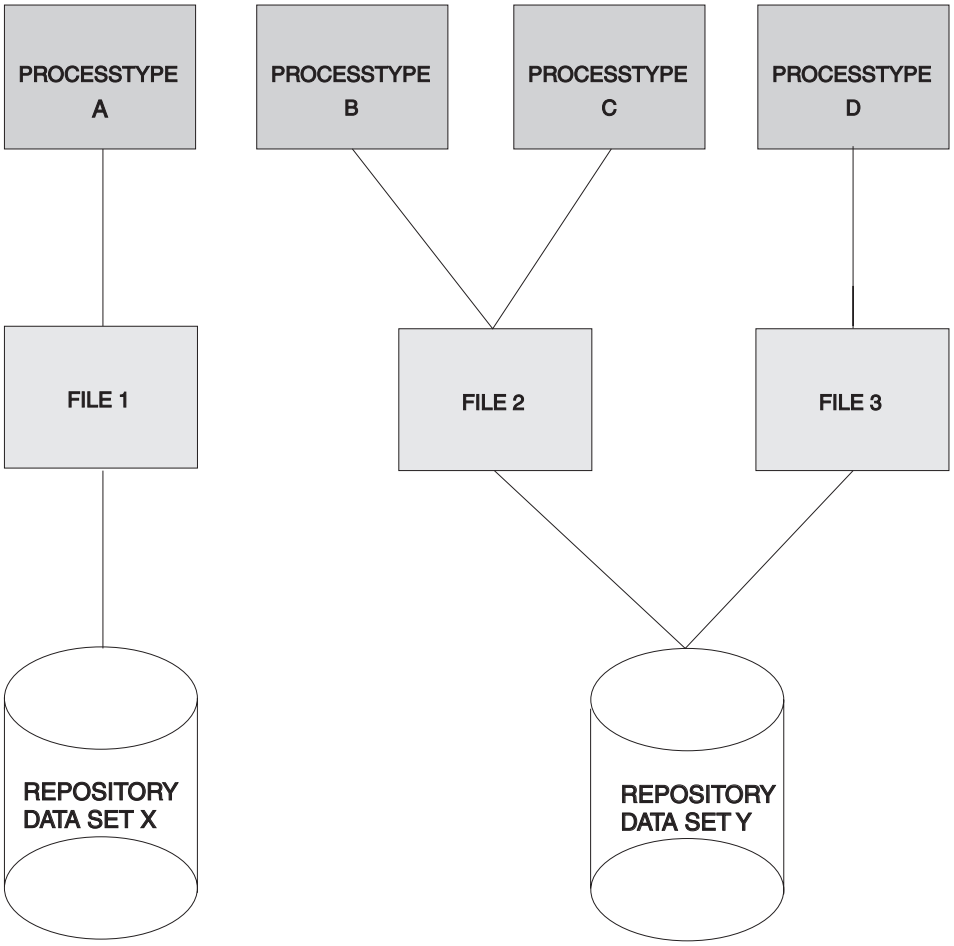


Figure 56. PROCESSTYPE definitions, FILE definitions, and repository data sets

DEFINE panel

```
 Processtype  ==>
  Group        ==>
  DEScription  ==>

INITIAL STATUS
 STAtus       ==> Enabled          Enabled | Disabled
DATA SET PARAMETERS
 File         ==>
AUDIT TRAIL
 Auditlog     ==>
 Auditlevel   ==> Off               Off | Process ? Activity ? Full
```

Figure 57. The DEFINE panel for PROCESSTYPE

Options

AUDITLEVEL({OFF|*PROCESS***?***ACTIVITY***?***FULL***})**
specifies the initial level of audit logging for processes of this type. If you specify any value other than OFF, you must also specify the AUDITLOG option.

ACTIVITY
Activity-level auditing. Audit records are written from:

1. The process audit points

2. The activity primary audit points.

FULL Full auditing. Audit records are written from:

1. The process audit points
2. The activity primary *and* secondary audit points.

OFF No audit trail records are written.

PROCESS
Process-level auditing. Audit records are written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see "Specifying the level of audit logging" on page 170.

AUDITLOG(*name***)**
specifies the name of a CICS journal to which audit trail records are written, for processes of this type and their constituent activities. The name can be up to eight characters long. If you do not specify an audit log, no audit records are kept for processes of this type.

DESCRIPTION(*text***)**
You can provide a description of the resource you are defining in this field. The DESCRIPTION text can be up to 58 characters in length. There are no restrictions on the characters that you can use. However, if you use parentheses, ensure that for each left parenthesis there is a matching right one. For each single apostrophe in the text, code 2 apostrophes.

FILE(*name***)**
specifies the name of the CICS file definition that is used to write the process and activity records of this process-type to its associated repository data set. The name can be up to eight characters long. The acceptable characters are A-Z 0-9 $ @ and #. Lowercase characters are converted to uppercase.

You must specify the FILE option.

PROCESSTYPE(*name***)**
specifies the name of this PROCESSTYPE definition. The name can be up to eight characters in length. The acceptable characters are A-Z a-z 0-9 $ @ # . / -_ % & ? ! : | " = ¬ , ; < >. Leading and embedded blank characters are not permitted. If the name supplied is less than eight characters, it is padded with trailing blanks up to eight characters.

STATUS({ENABLED|*DISABLED***})**
specifies the initial status of the process-type following a CICS initialization with START=COLD or START=INITIAL. After initialization, you can use the CEMT SET PROCESSTYPE command to change the status of the process-type. The status of the process-type following a restart is recovered to its status at the previous shutdown.

DISABLED
Processes of this type cannot be created. An EXEC CICS DEFINE PROCESS request that tries to create a process of this type results in the INVREQ condition being returned to the application program.

ENABLED
Processes of this type can be created.

Chapter 12. Security in BTS

Security considerations for CICS business transaction services are the authority to access BTS resources, the user IDs under which a process and its constituent activities run, the authority to attach the process and its constituent activities and the authority to use BTS system programming commands.

CICS security is described in detail in RACF security overview in the RACF Security Guide. Users of external security managers (ESMs) other than the Resource Access Control Facility (RACF®) must read this information with the documentation for their own ESM.

Resource security in BTS

You can protect BTS resources such as processes, activities, and containers in the same way as resources accessed by CICS file control commands. That is, resource-level security for a process, its activities, and their containers is based on the CICS file definition that specifies the repository data set to which records for processes of this type are written.

Users who run programs that define or acquire processes or activities of a particular process-type need UPDATE access to the corresponding CICS file.

Note: When a task issues an ACQUIRE command, CICS allows the appropriate record to be read from the BTS repository, even if the userid associated with the request has only READ access. However, when the task issues a sync point the record is written back to the data set and, if the userid does not have UPDATE access, the task abends.

Users who inquire on or browse processes or activities of a particular process-type need at least READ access to the corresponding CICS file.

Process and activity user IDs

You can use either the RUN or LINK command to activate a process or activity. The command that you use affects the user ID under which the process or activity runs.

User IDs for activities activated by RUN commands

When a process or activity is activated by a RUN command, it might run using a different user ID than the transaction that issues the RUN.

The application programmer can specify under whose authority a process or activity is to run, when it is activated by a RUN command, by coding the user ID option of the DEFINE PROCESS or DEFINE ACTIVITY command. If the user ID option is omitted, its value defaults to the userid of the transaction that issues the DEFINE command.

The user ID obtained from the DEFINE command is referred to as the **defined process userid** or the **defined activity userid**. In the remainder of this chapter, we use the term "defined user ID" to mean either a defined process user ID or a defined activity user ID.

If the user ID option of DEFINE PROCESS or ACTIVITY is specified, CICS performs (at define time) a surrogate security check to verify that the userid of

© Copyright IBM Corp. 1999, 2011

the transaction that issued the DEFINE command is authorized to use the defined user ID. The RACF profile used for surrogate checking of a BTS process or activity is `userid.DFHSTART` in the SURROGAT class.

The following example RACF commands authorize a user as a surrogate user of a defined process userid and of a defined activity userid:

```
RDEFINE SURROGAT defined_process_userid.DFHSTART UACC(NONE)
          OWNER(defined_process_userid)

PERMIT defined_process_userid.DFHSTART CLASS(SURROGAT)
          ID(define_process_command_userid) ACCESS(READ)

RDEFINE SURROGAT defined_activity_userid.DFHSTART UACC(NONE)
          OWNER(defined_activity_userid)

PERMIT defined_activity_userid.DFHSTART CLASS(SURROGAT)
          ID(define_activity_command_userid) ACCESS(READ)
```

Userids for activities activated by LINK commands
When a process or activity is activated by a LINK command, it runs under the userid of the transaction that issues the LINK.

Resource-level security checking *in* a process or activity is based on the user ID under whose authority the process or activity runs - that is, the defined userid or the userid of the transaction that issues the LINK command. This user ID must have UPDATE access to the CICS file that corresponds to the process-type.

Attach-time security for processes and activities

You can use Attach-time security to check if a transaction has authority to attach (activate) a process or activity. Attach-time security applies only when a process or activity is activated by a RUN command, not when it is activated by a LINK.

If attach-time security is required for a process, the defined userid - that is, the userid obtained from the DEFINE PROCESS command, must be given UPDATE access to the CICS file that corresponds to the BTS data set on which details of the process and its constituent activities are stored.

Command security in BTS

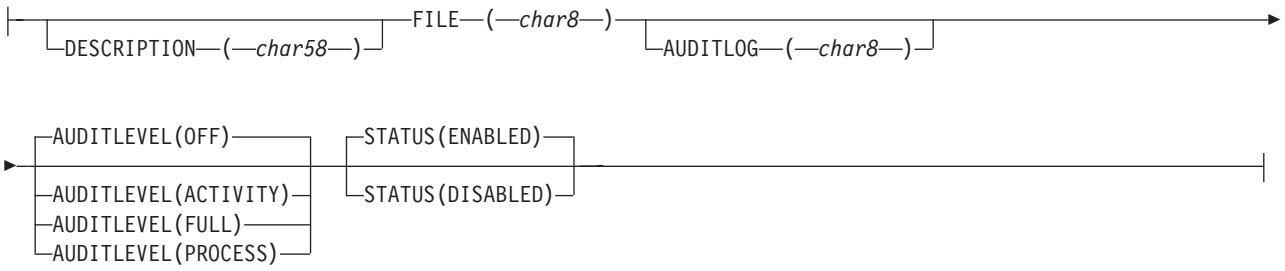
You can use CICS command-level security to protect your BTS system programming commands: **EXEC CICS CREATE PROCESSTYPE**, **DISCARD PROCESSTYPE**, **INQUIRE PROCESSTYPE**, and **SET PROCESSTYPE**.

CREATE PROCESSTYPE

Use the **CREATE PROCESSTYPE** command to add the definition of a BTS process-type to the local CICS region.

CREATE PROCESSTYPE

ATTRIBUTES:



Conditions: ILLOGIC, INVREQ, LENGERR, NOTAUTH

Description

CREATE PROCESSTYPE adds the definition of a BTS process-type to the local CICS region. The definition is built without reference to data stored on the CSD file. If there is already a process-type by the name you specify in the local CICS region, the new definition replaces the old one; if not, the new definition is added.

A syncpoint is implicit in CREATE PROCESSTYPE processing, except when an exception condition is detected early in processing the command. Uncommitted changes to recoverable resources made up to that point in the task are committed if the CREATE executes successfully and rolled back if not. For other general rules about CREATE commands, see *Creating resource definitions*, in the *CICS System Programming Reference*.

Options

ATTRIBUTES(data-value)
specifies the attributes of the PROCESSTYPE being added. The list of attributes must be coded as a single character string using the syntax shown in **PROCESSTYPE attributes**. For general rules for specifying attributes, see *The ATTRIBUTES option*, in the *CICS System Programming Reference*. For details of specific attributes, see "CEDA DEFINE PROCESSTYPE" on page 119.

ATTRLEN(data-value)
specifies the length in bytes of the character string supplied in the ATTRIBUTES option, as a halfword binary value. The length can be from 0 to 32767.

PROCESSTYPE(data-value)
specifies the name (1-8 characters) of the PROCESSTYPE definition to be added to the CICS region. The acceptable characters are A-Z a-z 0-9 $ @ # . / -_ % & ? ! : | " = ¬ , ; < >. Leading and embedded blank characters are not permitted. If the name supplied is less than eight characters, it is padded with trailing blanks up to eight characters.

Conditions

ILLOGIC
RESP2 values:

2 The command cannot be executed because an earlier CONNECTION or TERMINAL pool definition has not yet been completed.

INVREQ
RESP2 values:

n There is a syntax error in the ATTRIBUTES string, or an error occurred during either the discard or resource definition phase of the processing.

200 The command was executed in a program defined with an EXECUTIONSET value of DPLSUBSET or a program invoked from a remote system by a distributed program link without the SYNCONRETURN option.

LENGERR
RESP2 values:

1 The length you have specified in ATTRLEN is negative.

NOTAUTH
RESP2 values:

100 The user associated with the issuing task is not authorized to use this command.

101 The user associated with the issuing task is not authorized to create a PROCESSTYPE definition with this name.

102 The caller does not have surrogate authority to install the resource with the particular userid.

DISCARD PROCESSTYPE

Use the **DISCARD PROCESSTYPE** command to remove the definition of a specified process-type from the local CICS region.

DISCARD PROCESSTYPE

Conditions: INVREQ, NOTAUTH, PROCESSERR

Description

DISCARD PROCESSTYPE removes the definition of a specified process-type from the local CICS region.

Note:

1. Only disabled process-types can be discarded.
2. If you are using BTS in a single CICS region, you can use the DISCARD PROCESSTYPE command to remove process-types. However, if you are using BTS in a sysplex, it is strongly recommended that you use CPSM to remove them. If you don't use CPSM, problems could arise if Scheduler Services routes to this region work that requires a discarded definition.

Options

PROCESSTYPE(data-value)
specifies the name (1-8 characters) of the process-type to be removed.

Conditions

INVREQ
RESP2 values:

2 The process-type named in the PROCESSTYPE option is not disabled.

NOTAUTH
RESP2 values:

100 The user associated with the issuing task is not authorized to use this command.

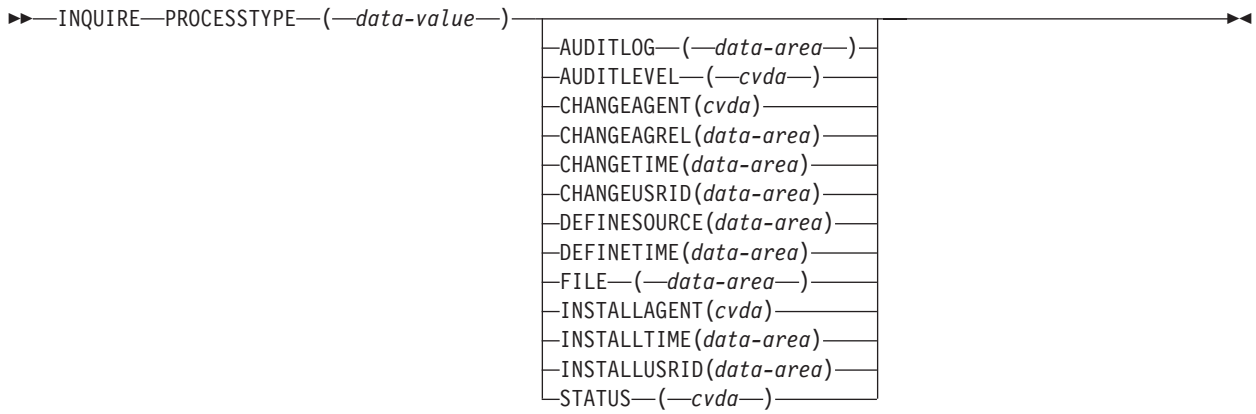
PROCESSERR
RESP2 values:

1 The process-type named in the PROCESSTYPE option is not defined in the process-type table (PTT).

INQUIRE PROCESSTYPE

Use the **INQUIRE PROCESSTYPE** command to retrieve the attributes of a process-type.

INQUIRE PROCESSTYPE



Conditions: NOTAUTH, PROCESSERR

Description

INQUIRE PROCESSTYPE returns the attributes of a specified process-type.

The resource signature

You can use this command to retrieve the resource signature fields. You can use these fields to manage resources by capturing details of when the resource was defined, installed, and last changed. For more information, see Auditing resources

in the Resource Definition Guide. The resource signature fields are CHANGEAGENT, CHANGEAGREL, CHANGETIME, CHANGEUSRID, DEFINESOURCE, DEFINETIME, INSTALLAGENT, INSTALLTIME, and INSTALLUSRID. See Summary of the resource signature field values in the Resource Definition Guide for detailed information about the content of the resource signature fields.

Options

AUDITLEVEL(*cvda***)**
Indicates the level of audit currently active for processes of the specified type. CVDA values are as follows:

ACTIVITY
Activity-level auditing. Audit records are written from the following points:
- The process audit points
- The activity primary audit points.

FULL Full auditing. Audit records are written from the following points:
- The process audit points
- The activity primary *and* secondary audit points.

OFF No audit trail records are written.

PROCESS
Process-level auditing. Audit records are written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see "Specifying the level of audit logging" on page 170.

AUDITLOG(*data-area***)**
Returns the 8-character name of the CICS journal used as the audit log for processes of the specified type.

CHANGEAGENT(*cvda***)**
Returns a CVDA value that identifies the agent that made the last change to the resource definition. The possible values are as follows:

CREATESPI
The resource definition was last changed by an **EXEC CICS CREATE** command.

CSDAPI
The resource definition was last changed by a CEDA transaction or the programmable interface to DFHEDAP.

CSDBATCH
The resource definition was last changed by a DFHCSDUP job.

DREPAPI
The resource definition was last changed by a CICSPlex SM BAS API command.

CHANGEAGREL(*data-area***)**
Returns a 4-digit number of the CICS release that was running when the resource definition was last changed.

CHANGETIME(*data-area***)**
Returns an ABSTIME value that represents the time stamp when the resource

definition was last changed. For more information about the format of the ABSTIME value, see FORMATTIME in CICS Application Programming.

CHANGEUSRID(*data-area***)**
Returns the 8-character user ID that ran the change agent.

DEFINESOURCE(*data-area***)**
Returns the 8-character source of the resource definition. The DEFINESOURCE value depends on the CHANGEAGENT value. For more information, see Summary of the resource signature field values in the Resource Definition Guide.

DEFINETIME(*data-area***)**
Returns an ABSTIME value that represents the time stamp when the resource definition was created.

FILE(*data-area***)**
Returns the 8-character name of the CICS file associated with the process-type.

INSTALLAGENT(*cvda***)**
Returns a CVDA value that identifies the agent that installed the resource. The possible values are as follows:

CREATESPI
The resource was installed by an **EXEC CICS CREATE** command.

CSDAPI
The resource was installed by a CEDA transaction or the programmable interface to DFHEDAP.

GRPLIST
The resource was installed by **GRPLIST INSTALL**.

INSTALLTIME(*data-area***)**
Returns an ABSTIME value that represents the time stamp when the resource was installed.

INSTALLUSRID(*data-area***)**
Returns the 8-character user ID that installed the resource.

PROCESSTYPE(*data-value***)**
Specifies the name (1 - 8 characters) of the process-type being inquired on.

STATUS(*cvda***)**
Indicates whether new processes of the specified type can currently be defined. CVDA values are as follows:

DISABLED
The installed definition of the process-type is disabled. New processes of this type cannot be defined.

ENABLED
The installed definition of the process-type is enabled. New processes of this type can be defined.

Conditions

NOTAUTH
RESP2 values:

100 The user associated with the issuing task is not authorized to use this command.

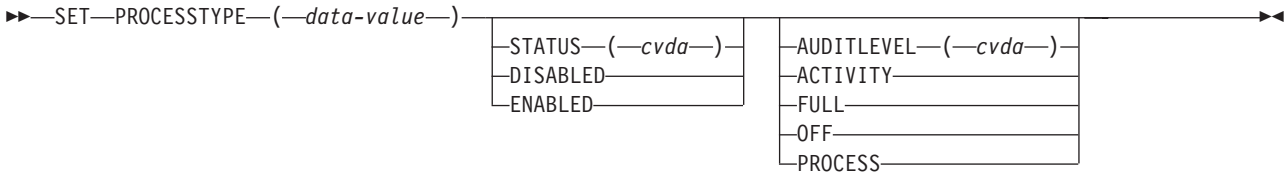
PROCESSERR
RESP2 values:

1 The process-type specified on the PROCESSTYPE option could not be found.

SET PROCESSTYPE

Use the **SET PROCESSTYPE** command to change the attributes of a process-type.

SET PROCESSTYPE



Conditions: INVREQ, NOTAUTH, PROCESSERR

Description

SET PROCESSTYPE allows you to change the current state of audit logging and the enablement status of PROCESSTYPE definitions installed on this CICS region.

Note: Process-types are defined in the process-type table (PTT). CICS uses the entries in this table to maintain its records of processes (and their constituent activities) on external data sets. If you are using BTS in a single CICS region, you can use the SET PROCESSTYPE command to modify your process-types. However, if you are using BTS in a sysplex, it is strongly recommended that you use CPSM to make such changes. This is because it is essential to keep resource definitions in step with each other, across the sysplex.

Options

AUDITLEVEL(cvda)
specifies the level of audit logging to be applied to processes of this type.

Note: If the AUDITLOG attribute of the installed PROCESSTYPE definition is not set to the name of a CICS journal, an error is returned if you try to specify any value other than OFF.

The CVDA values are:

ACTIVITY
Activity-level auditing. Audit records will be written from:
1. The process audit points
2. The activity primary audit points.

FULL Full auditing. Audit records will be written from:
1. The process audit points
2. The activity primary *and* secondary audit points.

OFF No audit trail records will be written.

PROCESS
Process-level auditing. Audit records will be written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see "Specifying the level of audit logging" on page 170.

PROCESSTYPE(value)
specifies the 8-character name of a process-type defined in the process-type table (PTT), whose attributes are to be changed.

STATUS(cvda)
specifies whether new processes of this type can be created. The CVDA values are:

DISABLED
The installed definition of the process-type is disabled. New processes of this type cannot be defined.

ENABLED
The installed definition of the process-type is enabled. New processes of this type can be defined.

Conditions

INVREQ
RESP2 values:

2 The process-type is not disabled, and therefore cannot be enabled.

3 You have specified an invalid CVDA value on the AUDITLEVEL option.

5 You have specified an invalid CVDA value on the STATUS option.

6 You have specified a value of FULL, PROCESS, or ACTIVITY on the AUDITLEVEL option, but the AUDITLOG attribute of the PROCESSTYPE definition does not specify an audit log.

NOTAUTH
RESP2 values:

100 The user associated with the issuing task is not authorized to use this command.

PROCESSERR
RESP2 values:

1 The process-type named in the PROCESSTYPE option is not defined in the process-type table (PTT).

Chapter 13. BTS operator commands

Use these operator commands to inquire about and control CICS business transaction services resources.

It contains:

- "CBAM BTS browser"
- "CEMT INQUIRE PROCESSTYPE" on page 140
- "CEMT INQUIRE TASK" on page 144
- "CEMT SET PROCESSTYPE" on page 145.

For introductory and guidance information about the CICS master terminal transaction, CEMT, see the *CEMT master terminal* chapter in the *CICS Supplied Transactions*.

CBAM BTS browser

You can use the CBAM transaction to browse the CICS business transaction services objects (process-types, processes, activities, containers, events, and timers) known to your region.

CBAM is a menu-driven transaction. The menus are hierarchically organized. By navigating downwards through the menus, you can display:

1. All the process-types that have been defined to this region with installed PROCESSTYPE definitions.
2. All the processes of a selected process-type. These processes are the processes of the selected type that currently exist on the repository data set pointed to by the installed PROCESSTYPE definition.

   Note: If you are operating BTS in a sysplex and the repository is shared with one or more other regions, some of the processes might have been defined on other regions.
3. The constituent activities of a selected process.
4. The details (program, transaction ID, user ID) of a selected activity.
5. One of the following resources:
   - The containers associated with a selected activity or process, *or*
   - The events in an event pool of the selected activity, *or*
   - The timers defined to a selected activity.

Note: This overview of the CBAM menu hierarchy is slightly simplified. Selectable fields allow you to bypass some screens.

CBAM is a "read-only" transaction - you cannot update any of the displayed attributes by over typing them.

Running the CBAM browser transaction

Start the CBAM browser transaction by typing CBAM on the command line and pressing the ENTER key. This gives you a list of all the process-types that have been defined to this region.

© Copyright IBM Corp. 1999, 2011

Process-types screen

The *Process-types* screen lists the process-types defined to a region and shows the CICS repository file name, process-type status, and the level of audit logging.

```
CBAM

  Processtype File     Status   Auditlevel

  CBTSAUDA    DFHBARF  Enabled  Activity
  CBTSSHR     DFHBSHR  Enabled  Off
  CBTSSHRF    DFHBSHR  Disabled Activity
  CBTSSHR2    DFHBSHR2 Disabled Off
  MORTLOANS   DFHMORT  Enabled  Process




 Use cursor and Enter for Processes

PF3=Return   7=Back   8=Forward
```

Figure 58. CBAM transaction: initial screen, showing the process-types defined to this region

The displayed fields mean:

Auditlevel
The level of audit logging currently active for processes of this type:

Activity
Activity-level auditing. Audit records are written from:

1. The process audit points
2. The activity primary audit points.

Full Full auditing. Audit records are written from:

1. The process audit points
2. The activity primary *and* secondary audit points.

Off No audit records are written.

Process
Process-level auditing. Audit records are written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see "Specifying the level of audit logging" on page 170.

File
The CICS repository file on which records for processes of this type are stored.

Status
Whether the PROCESSTYPE is enabled or disabled—that is, whether new processes of this process-type can be defined.

Processtype
The name of a process-type.

If you place the cursor on the name of a process-type, or anywhere on the same line, and press ENTER, you get a list of all the processes of that type that currently exist on the repository data set pointed to by the installed PROCESSTYPE definition—see Figure 59 on page 135.

Processes screen

The *Processes* screen lists the process names, the mode of a process, the completion status, whether a process is suspended, and lists process containers for the process named in the process field.

```
CBAM                                        Processtype MORTLOANS

  Process                           Mode       Comp       Susp    Conts

  MORT000000014                     Dormant    Incomplete No       _
  MORT000000015                     Complete   Forced     No       _
  MORT000000016                     Dormant    Incomplete Sus      _
  MORT000000017                     Active     Incomplete No       _
  PERS000000114                     Initial    Incomplete No       _




 Use cursor and Enter for Activities or Containers (tab to Conts)

PF3=Return   7=Back   8=Forward
```

Figure 59. CBAM transaction: processes screen

The displayed fields mean:

Comp
The completion status of the process:

Abend
The program that implements the root activity abended.

Forced
The process was forced to complete—for example, it was canceled with a CANCEL ACQPROCESS command.

Incomplete
The process is incomplete.

Normal
The process completed normally.

Conts
A selectable field. If you place the cursor on this field and press ENTER, you get a list of the process-containers for the process named in the **Process** field. For an example of the CBAM *Containers* screen, see Figure 62 on page 138.

Mode
The mode of the process. One of:

- Active
- Cancelling
- Complete
- Dormant
- Initial.

For an explanation of each of these modes, see “Processing modes” on page 18.

Process
The name of a process.

Susp
Whether the process is currently suspended:

No The process is not currently suspended.

Sus The process is currently suspended.

If you place the cursor on the name of a process and press ENTER, you get a list of the process's constituent activities—see Figure 60.

Activities screen

The *Activities* screen lists the activity names, the activity mode, the completion status, and whether a process is suspended.

```
CBAM         Process MORT000000017                Processtype MORTLOANS

  Activity                                       Mode       Comp       Susp

  DFHROOT                                        Dormant    Incomplete No
    NEWMORT                                      Complete   Normal     No
    PAYMENT-RECEIVED                             Complete   Normal     No
    PAYMENT-OVERDUE                              Complete   Normal     No
    INTEREST-CHANGE                              Complete   Normal     No
    CAPITAL-REPAYMNT                             Dormant    Incomplete No
      Credit-Account                             Complete   Normal     No
      Adjust-Interest                            Active     Incomplete No




 Use cursor and Enter for details

PF3=Return   7=Back   8=Forward
```

Figure 60. CBAM transaction: activities screen

The displayed fields mean:

Activity
The name of an activity.

The list of constituent activities is indented. The amount by which an activity is indented represents its level in the process's activity tree.

Comp
The completion status of the activity:

Abend
The activity abended.

Forced
The activity was forced to complete—for example, it was canceled with a CANCEL ACTIVITY command.

Incomplete
The activity is incomplete.

Normal
The activity completed normally.

Mode
The mode of the activity. One of:

- Active
- Cancelling
- Complete

- Dormant
- Initial.

For an explanation of each of these modes, see "Processing modes" on page 18.

Susp
Whether the activity is currently suspended:

No The activity is not currently suspended.

Sus The activity is currently suspended.

If you place the cursor on the name of an activity and press ENTER, you get details of the activity—see Figure 61.

Activity details screen

The *Activity details* screen lists the name of the program that implements the selected activity, the transaction identifier, and the user ID under which the activity runs.

```
CBAM         Process MORT000000017                 Processtype MORTLOANS

  Activity DFHROOT

  Program    MORTGAGE
  Transid    MORT
  Userid     CBTSMOR


  Containers
  Events
  Timers




 Use cursor and Enter for Containers, Events or Timers

PF3=Return   7=Back   8=Forward
```

Figure 61. CBAM transaction: activity details screen

There are also three selectable fields:

Containers
Pressing ENTER on this field displays a list of the containers associated with the selected activity. For more information, see Figure 63 on page 138.

Events
Pressing ENTER on this field displays a list of the events in the event pool of the selected activity. For more information, see Figure 64 on page 138.

Timers
Pressing ENTER on this field displays a list of the timers defined to the selected activity. For more information, see Figure 65 on page 139.

Containers screen

The *Containers* screen lists each container associated with a specified process or activity and shows the length, in bytes, of the data contained in it.

```
CBAM         Process MORT000000017                    Processtype MORTLOANS

  Container        Datalength

  ACCOUNT-NO              36
  BORROWER-INFO         1000




  PF3=Return   7=Back   8=Forward
```

Figure 62. CBAM transaction: containers screen for a process

```
CBAM         Process MORT000000017                    Processtype MORTLOANS

  Activity DFHROOT

  Container        Datalength

  STATUS                 500



 PF3=Return   7=Back   8=Forward
```

Figure 63. CBAM transaction: containers screen for an activity

Events screen

The *Events* screen lists the events in the specified event pool of the activity. The events listed are the events that are currently in the event pool. Events that have been deleted do not appear in the list.

```
CBAM         Process MORT000000017                    Processtype MORTLOANS

  Activity DFHROOT

  Event            Type      Fired  Composite       Timer

  ALL-TIMERS       Composite Yes    OR
  ANNUAL-STATMNT   Timer     Yes    ALL-TIMERS      ANNUAL-STATEMENT
  ANNUAL-ST-DONE   Activity  No
  CAPITAL-REPAYMNT Input     No
  CAP-REPT-DONE    Activity  No
  DFHINITIAL       System    Yes
  INTEREST-CHANGE  Input     No
  PAYMENT-OVRDUE   Timer     No     ALL-TIMERS      PAYMENT-OVERDUE
  PAYMENT-RECEIVED Input     No









PF3=Return   7=Back   8=Forward
```

Figure 64. CBAM transaction: events screen

The displayed fields mean:

Composite
If the event is a composite, the Boolean operator (AND or OR) applied to its predicate.

If the event is a subevent, the name of the composite event of which it forms part.

Event
The name of an event.

Fired
The fire status of the event.

This field shows the *current* fire status of the event, *not* whether the event has ever fired in the past. For example, the fire status of an atomic event that has fired and been retrieved (but not deleted) is shown as 'No', because the act of retrieving the event has reset its fire status to NOTFIRED.

No Not fired

Yes Fired

Timer
If the event is a timer event, the name of its associated timer.

Type
The type of the event:

Activity
Activity completion

Composite
Composite

Input Input

System
System

Timer Event associated with a timer

Timers screen

The *Timers* screen lists the timers that are currently defined to a specified activity and shows their status.

```
CBAM         Process MORT000000017                      Processtype MORTLOANS

  Activity DFHROOT

  Timer             Status    Event            Date       Time

  ANNUAL-STATMNT    Expired   ANNUAL-STATEMENT 12151998   235959
  PAYMENT-OVRDUE    Unexpired PAYMENT-OVERDUE  06301999   235959




PF3=Return   7=Back   8=Forward
```

Figure 65. CBAM transaction: timers screen

The displayed fields mean:

Date
The expiry date of the timer, in the form **mmddyyyy**.

Event
The name of the event associated with the timer.

Status
The state of the timer:

Expired
The timer expired normally.

Forced
Expiry of the timer was forced with a FORCE TIMER command.

Unexpired
The timer has not yet expired.

Time
The expiry time of the timer, in the form **hhmmss**.

Timer
The name of a timer.

CEMT INQUIRE PROCESSTYPE

Retrieve information about a CICS business transaction services process-type.

Description

The **INQUIRE PROCESSTYPE** command returns information about the BTS PROCESSTYPE definitions installed on this CICS region. In particular, it shows the current state of audit logging for each displayed process-type.

The resource signature

You can use this command to display the resource signature fields. You can use these fields to manage resources by capturing details of when the resource was defined, installed, and last changed. For more information, see Auditing resources in the Resource Definition Guide. The resource signature fields are CHANGEAGENT, CHANGEAGREL, CHANGETIME, CHANGEUSRID, DEFINESOURCE, DEFINETIME, INSTALLAGENT, INSTALLTIME, and INSTALLUSRID. See Summary of the resource signature field values in the Resource Definition Guide for detailed information about the content of the resource signature fields.

Input

Press the Clear key to clear the screen. You can start this transaction in two ways:

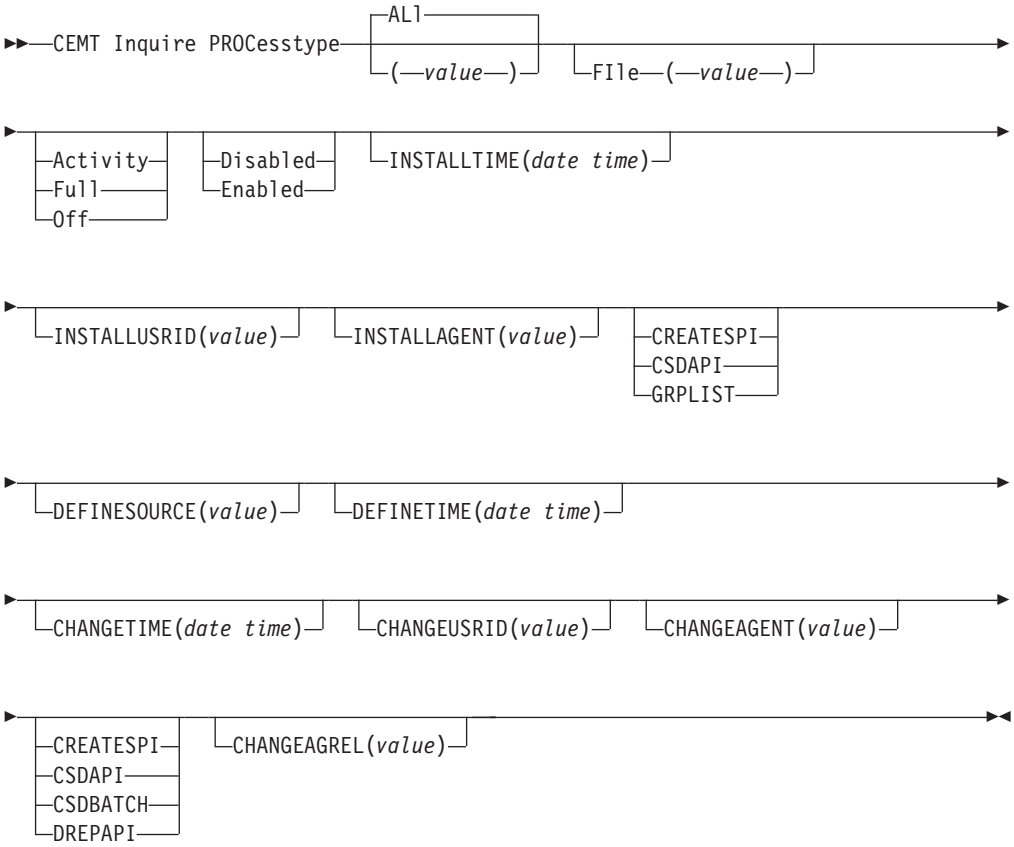
- Type `CEMT INQUIRE PROCESSTYPE` (or suitable abbreviations for the keywords). The resulting display lists the current status.
- Type `CEMT INQUIRE PROCESSTYPE` (or suitable abbreviations for the keywords), followed by the attributes that are necessary to limit the range of information that you require. For example, if you enter `cemt i proc en`, the resulting display shows the details of only those process-types that are enabled.

You can change various attributes in the following ways:

- Overtype your changes on the INQUIRE screen after tabbing to the appropriate field.

- Use the CEMT SET PROCESSTYPE command.

CEMT INQUIRE PROCESSTYPE



ALl
The default. Information about all process-types is displayed, unless you specify a selection of process-types to be queried.

(value)
The name (1 - 8 characters) of one or more PROCESSTYPE definitions installed in the process-type table (PTT).

Sample screen

```
I PROC
STATUS:   RESULTS - OVERTYPE TO MODIFY
 Pro(PROCESSTYPE12  ) Fil(FILE12  ) Aud(ADTLOG12) Pro Ena
 Pro(PROCESSTYPE13  ) Fil(FILE13  ) Aud(ADTLOG12) Off Ena
 Pro(PTYPE2B        ) Fil(FILE2B  ) Aud(DFHJ2B  ) Ful Ena
 Pro(PTYPE39        ) Fil(FILE39  ) Aud(DFHJ39  ) Off Ena
 Pro(SALESTYPE1     ) Fil(SALESF1 ) Aud(PLOG51  ) Off Dis
 Pro(SALESTYPE4     ) Fil(SALESF4 ) Aud(PLOG51  ) Act Ena
 Pro(SALESTYPE6     ) Fil(SALESF6 ) Aud(PLOG51  ) Off Ena
```

Figure 66. CEMT INQUIRE PROCESSTYPE screen

If you place the cursor against a specific entry in the list and press ENTER, CICS displays an expanded format, as shown in Figure 67 on page 142.

```
I PROCESSTYPE(*)
RESULT - OVERTYPE TO MODIFY
  Processtype(SALESTYPE4     )
  File(SALESF4 )
  Auditlog(PLOG51  )
  Auditlevel(Activity )
  Status( Enabled )
```

Figure 67. The expanded display of an individual entry

Displayed fields

Auditlevel
Displays the level of audit logging currently active for processes of this type. The values are as follows:

ACtivity
Activity-level auditing. Audit records are written from the following points:
1. The process audit points
2. The activity primary audit points.

FUll Full auditing. Audit records are written from the following points:
1. The process audit points
2. The activity primary *and* secondary audit points.

Off No audit trail records are written.

Process
Process-level auditing. Audit records are written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see "Specifying the level of audit logging" on page 170.

AUditlog*(value)*
Displays the 8-character name of the CICS journal used as the audit log for processes of this type.

CHANGEAGENT(*value***)**
Displays a value that identifies the agent that made the last change to the resource definition. You cannot use CEMT to filter on some of these values because they are duplicated. The possible values are as follows:

CREATESPI
The resource definition was last changed by an **EXEC CICS CREATE** command.

CSDAPI
The resource definition was last changed by a CEDA transaction or the programmable interface to DFHEDAP.

CSDBATCH
The resource definition was last changed by a DFHCSDUP job.

DREPAPI
The resource definition was last changed by a CICSPlex SM BAS API command.

CHANGEAGREL(*value***)**
Displays the 4-digit number of the CICS release that was running when the resource definition was last changed.

CHANGETIME(*date time***)**
Displays the date and time when the resource definition was last changed. The format of the date depends on the value that you selected for the DATFORM system initialization parameter for your CICS region. The format of the time is hh:mm:ss.

CHANGEUSRID(*value***)**
Displays the 8-character user ID that ran the change agent.

DEFINESOURCE(*value***)**
Displays the source of the resource definition. The DEFINESOURCE value depends on the CHANGEAGENT option. For details, see Summary of the resource signature field values in the Resource Definition Guide.

DEFINETIME(*date time***)**
Displays the date and time when the resource was created. The format of the date depends on the value that you selected for the DATFORM system initialization parameter for your CICS region. The format of the time is hh:mm:ss.

File*(value)*
Displays the 8-character name of the CICS repository file on which the process and activity records for processes of this type are stored.

INSTALLAGENT(*value***)**
Displays a value that identifies the agent that installed the resource. You cannot use CEMT to filter on some of these values because they are duplicated. The possible values are as follows:

CREATESPI
The resource was installed by an **EXEC CICS CREATE** command.

CSDAPI
The resource was installed by a CEDA transaction or the programmable interface to DFHEDAP.

GRPLIST
The resource was installed by **GRPLIST INSTALL**.

INSTALLTIME(*date time***)**
Displays the date and time when the resource was installed. The format of the date depends on the value that you selected for the DATFORM system initialization parameter for your CICS region. The format of the time is hh:mm:ss.

INSTALLUSRID(*value***)**
Displays the 8-character user ID that installed the resource.

PROCesstype*(value)*
Indicates that this panel relates to a PROCESSTYPE inquiry and displays the 8-character name of a process-type.

Status
Displays whether new processes of this type can be created. The values are as follows:

Disabled
The installed definition of the process-type is disabled. New processes of this type cannot be defined.

Enabled
The installed definition of the process-type is enabled. New processes of this type can be defined.

CEMT INQUIRE TASK

Retrieve information about a user task.

Description

The **INQUIRE TASK** command returns information about user tasks. Only information about user tasks can be displayed or changed; information about CICS-generated system tasks or subtasks cannot be displayed or changed. System tasks are those tasks started (and used internally) by CICS, and not as a result of a user transaction.

Input

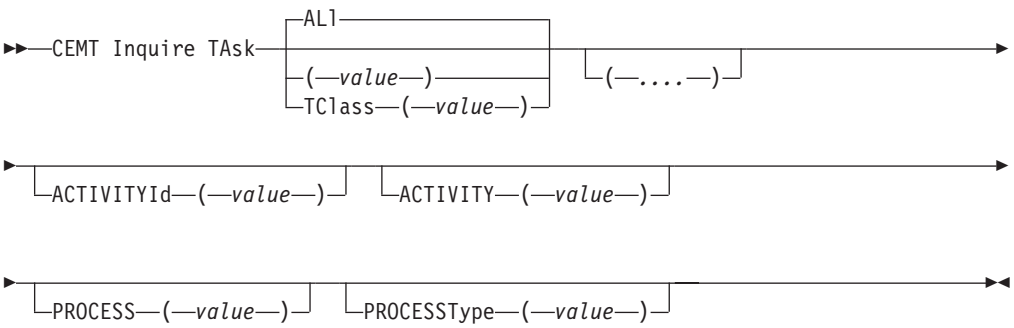
Press the Clear key to clear the screen. You can start this transaction in two ways:

- Type `CEMT INQUIRE TASK` (or suitable abbreviations for the keywords). The resulting display lists the current status.
- Type `CEMT INQUIRE TASK` (or suitable abbreviations for the keywords), followed by the attributes that are necessary to limit the range of information that you require. For example, if you enter `cemt i ta i`, the resulting display shows the details of only those tasks for which the data is not shared with other tasks (isolated).

You can change various attributes in the following ways:

- Overtype your changes on the INQUIRE screen after tabbing to the appropriate field.
- Use the CEMT SET TASK command.

CEMT INQUIRE TASK



(value)
The CICS-generated task number, in the range 1 - 99999.

ALl
The default. The maximum number of tasks that can be displayed is 32000.

TClass*(value)*
The 8-character transaction class name to which the transactions belong. The maximum number of tasks that can be displayed is 32000.

You cannot specify a list of identifiers, or use the asterisk (*) or plus (+) symbols to specify a family of tasks.

Sample screen

```
IN TASK
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0000033) Tra(CEMT) Fac(944D) Sus Ter Pri( 255 )
    Sta(TO) Use(BELL    ) Uow(AB9001D5F56CC800) Hty(ZCIOWAIT) Hva(DFHZARQ
 Tas(0000037) Tra(CEMT) Fac(S21D) Run Ter Pri( 255 )
    Sta(TO) Use(BELL    ) Uow(AB9002E745F93A00)
```

Figure 68. CEMT INQUIRE TASK screen

Note: Blank fields on the screen indicate that a value does not apply or is negative; that is, it begins with No. Underscores indicate input-only fields.

If you place the cursor against a specific entry in the list and press ENTER, CICS displays an expanded format, as shown in Figure 69.

```
IN TASK
RESULT - OVERTYPE TO MODIFY
  Task(0000033)
  Tranid(CEMT)
  Facility(944D)
  Runstatus(Suspended)
  Ftype(Term)
  Priority( 255 )
  Purgetype(              )
  Startcode(TO)
  Userid(BELL)
  Uow(AB9001D5F56CC800)
  Htype()
  Hvalue()
  Htime()
  Indoubt(Backout)
  Indoubtwait(Wait)
  Bridge()
  Activityid()
  Activity()
  Process()
  Processtype()
```

Figure 69. The expanded display of an individual entry

Displayed fields

Activityid*(value)*
Displays the 52-character, CICS-assigned, identifier of the CICS business transaction services activity that this task is executing on behalf of.

Activity*(value)*
Displays the 16-character, user-assigned, name of the CICS business transaction services activity that this task is executing on behalf of.

Process*(value)*
Displays the 36-character name of the CICS business transaction services process that this task is executing on behalf of.

Processtype*(value)*
Displays the 8-character process-type of the CICS business transaction services process that this task is executing on behalf of.

CEMT SET PROCESSTYPE

Change the attributes of a CICS business transaction services process-type.

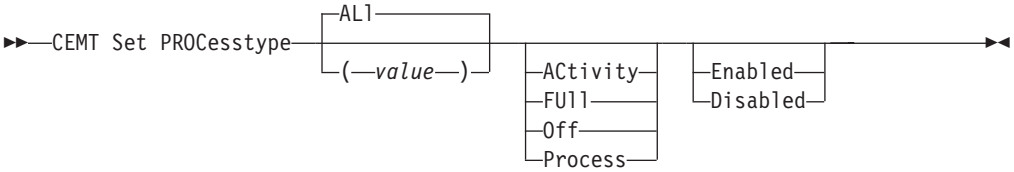
Description

SET PROCESSTYPE enables you to change the current state of audit logging and the enablement status of BTS PROCESSTYPE definitions installed on this CICS region.

Note: Process-types are defined in the process-type table (PTT). CICS uses the entries in this table to maintain its records of processes (and their constituent activities) on external data sets. If you are using BTS in a single CICS region, you can freely use the SET PROCESSTYPE command to modify your process-types. However, if you are using BTS in a sysplex, it is strongly recommended that you use CPSM to make such changes. This is because it is essential to keep resource definitions in step with each other, across the sysplex.

Syntax

CEMT SET PROCESSTYPE



Options

ACtivity|FUllvOffvProcess
specifies the level of audit logging to be applied to processes of this type.

Note: If the AUDITLOG attribute of the installed PROCESSTYPE definition is not set to the name of a CICS journal, an error is returned if you try to specify any value other than OFF.

The values are:

ACtivity
Activity-level auditing. Audit records will be written from:

1. The process audit points
2. The activity primary audit points.

FUll Full auditing. Audit records will be written from:

1. The process audit points
2. The activity primary *and* secondary audit points.

Off No audit trail records will be written.

Process
Process-level auditing. Audit records will be written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see "Specifying the level of audit logging" on page 170.

ALl
specifies that any changes you specify are made to all process-types that you are authorized to access.

Enabled|Disabled
specifies whether new processes of this type can be created. The values are:

Disabled
The installed definition of the process-type is disabled. New processes of this type cannot be defined.

Enabled
The installed definition of the process-type is enabled. New processes of this type can be defined.

PROCesstype(value)
specifies the 8-character name of the process-type whose attributes are to be changed.

Chapter 14. Using BTS in a sysplex

Terminology: This chapter assumes that you are familiar with the terminology and concepts of CICS dynamic routing. For introductory information about dynamic routing, see the *CICS Intercommunication Guide*.

You can operate BTS in a single CICS region. However, CICS business transaction services are sysplex-enabled. For example, you can use workload separation to ensure that processes of the same process-type are handled by a particular set of regions, and you can use workload routing to route activity requests across a set of regions.

When you use workload routing the activities that constitute the process might execute on several regions, and different activations of the same activity might execute on different regions

The scope of a BTS-set

You can use the scope of a *BTS-set* to define the set of CICS regions across which related BTS processes and activities can run.

All the regions in a BTS-set:

- Must be interconnected, to support routing of activities between the regions. This does not mean that the regions must be in the same CICSplex, but that each region in the BTS-set must be connected to every other region. For more information, see "Understanding distributed routing" on page 151.
- Must have access to the BTS repository data set (or data sets) on which details of the relevant processes are stored.

  There are two methods of sharing the repository data set:

  1. Using VSAM record-level sharing (RLS). To use this method, all the regions of the BTS-set must be within the same MVS Parallel Sysplex. (A "Parallel Sysplex" is a sysplex in which the MVS images are linked through a coupling facility). VSAM RLS requires a coupling facility. Each region in the BTS-set must open the shared repository file or files in RLS mode.
  2. Using function-shipping to a file-owning region (FOR).

Within an MVS sysplex, it is possible to have multiple BTS-sets. For more information, see Figure 72 on page 155. Imagine, for example, that within your sysplex you operate two CICSplexes. You could decide to divide your BTS processes by process-type, between the two CICSplexes. Alternatively, you could decide to set up two BTS-sets within the same CICSplex.

Using separate BTS-sets is a high-level form of workload separation. By definition, routing of activities *between* BTS-sets is not possible.

A note about audit logs

When you create a BTS-set, the activities that constitute a single process might run on several regions. Therefore, to collate the audit data for each process, your audit logs must use MVS shared log streams.

© Copyright IBM Corp. 1999, 2011

If all the CICS regions in your BTS-set are in the same MVS image, you can define the log streams to use either coupling facility structures or DASD-only logging. However, if the CICS regions are on *different* MVS images, the log streams must use coupling facility structures rather than DASD-only logging. This is because CICS regions on different MVS images cannot access the same DASD-only log stream at the same time.

If your BTS-set spans multiple MVS images and you use DASD-only log streams for your BTS logs, you are unable to use shared log streams. In this case, the audit records for a particular process could be split between several log streams; you have to collate the data yourself.

For further information about audit logs, see Chapter 16, "Creating a BTS audit trail," on page 169.

Dynamic routing of BTS activities

You can use a BTS-set to dynamically route your BTS processes and activities across the participating regions.

BTS routing is at the activation level; for example, in the same process, different activations of the same activity might execute on different regions. When an event is signaled, the relevant activity is activated in the most appropriate region in the BTS-set, based on one or more of these criteria:

- Any workload separation specified by the system programmer
- Any affinities its associated transaction has with a particular region
- The availability of regions
- The relative workload of regions.

Which BTS activities can be dynamically routed?

Not all activations of BTS processes and activities can be routed. Processes and activities that are activated asynchronously with the requestor, with a RUN ASYNCHRONOUS command, can be routed either dynamically or statically.

Processes and activities that are activated synchronously with the requestor, with a RUN SYNCHRONOUS or LINK command, are always run locally. They cannot be routed *dynamically or statically*. A RUN SYNCHRONOUS or LINK command issued against an activity whose associated transaction is defined as DYNAMIC(YES), or as remote, results in the activity being run locally. When an activity is activated by a LINK command, *all* the attributes of its associated transaction are ignored, because the activity runs under the TRANSID of the parent - there is no **context-switch**.

Thus, to be eligible for dynamic routing:

- A process or activity must be run asynchronously with the requestor, with a RUN ASYNCHRONOUS command.
- The TRANSACTION definition for the CICS transaction associated with the process or activity must specify DYNAMIC(YES).

"Daisy-chaining" is not supported. Once a BTS process or activity has been routed to a target region it cannot be rerouted from the target to a third region, even though its associated transaction is defined as DYNAMIC(YES) in the target.

Understanding distributed routing

CICS has two dynamic routing models, the *hub routing model* and the *distributed routing model*. For each model a user-replaceable sample routing program is supplied. The CICSPlex SM routing program is suitable for use with both models.

The two user-replaceable sample routing programs are the dynamic routing program, DFHDYP, which implements the hub routing model, and the distributed routing program, DFHDSRP, which implements the distributed routing model. Both models and their associated routing programs are described in detail in the *Two routing models* chapter in the *CICS Intercommunication Guide*.

The CICSPlex SM routing program, EYU9XLOP, can be used with either routing model - that is, it can function as either a dynamic routing program, a distributed routing program, or both.

BTS routing uses the distributed routing model. It is important to understand how the *distributed routing model* differs from the traditional *hub routing model*.

The hub model

The *hub* is the model that has traditionally been used with CICS dynamic transaction routing. The model has advantages and disadvantages.

A dynamic routing program running in a terminal-owning region (TOR) routes transactions between several application-owning regions (AORs). Usually, the AORs (unless they are AOR/TORs) do not perform dynamic routing. Figure 70 shows a hub routing model.

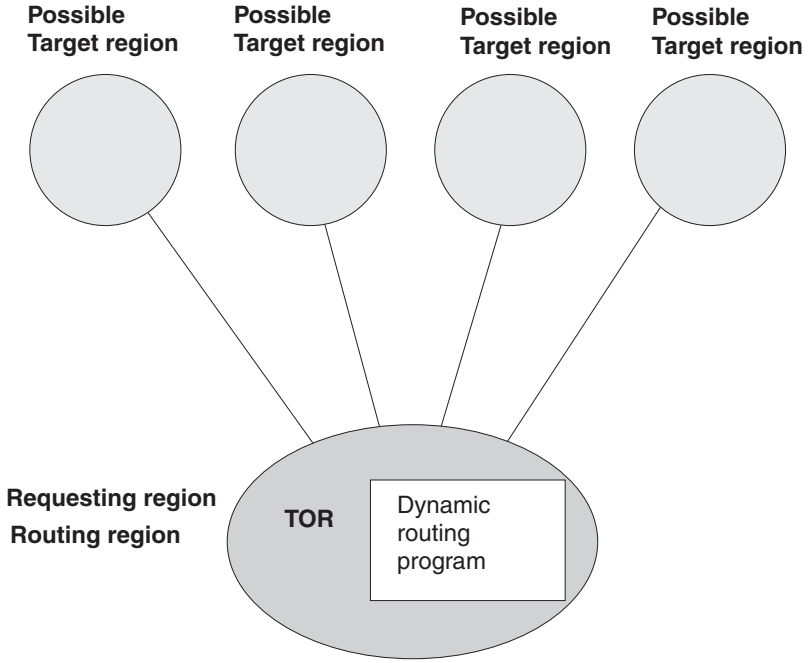


Figure 70. Dynamic routing using a hub routing model. One routing region (the TOR) selects between several target regions.

The hub model applies to the routing of:

- Transactions started from terminals
- Transactions started by terminal-related EXEC CICS START commands

- Program-link requests received from outside CICS. The receiving region acts as a hub or TOR because it routes the requests among a set of back-end server regions.

The hub model is a hierarchical system—routing is controlled by one region (the TOR); normally, a routing program runs only in the TOR.

Advantage of the hub model

It is a relatively simple model to implement. For example, compared to the distributed model, there are few inter-region connections to maintain.

Disadvantages of the hub model

- If you use only one hub to route transactions and program-link requests across your AORs, the hub TOR is a single point-of-failure.
- If you use more than one hub to route transactions and program-link requests across the same set of AORs, you may have problems with distributed data. For example, if the routing program keeps a count of routed transactions for load-balancing purposes, each hub TOR will need access to this data.

The distributed model

In the distributed model used for BTS routing, each participating CICS region might be both a routing region and a target region. A distributed routing program runs in each region. The distributed model has advantages and disadvantages.

Figure 71 on page 153 shows a distributed routing model.

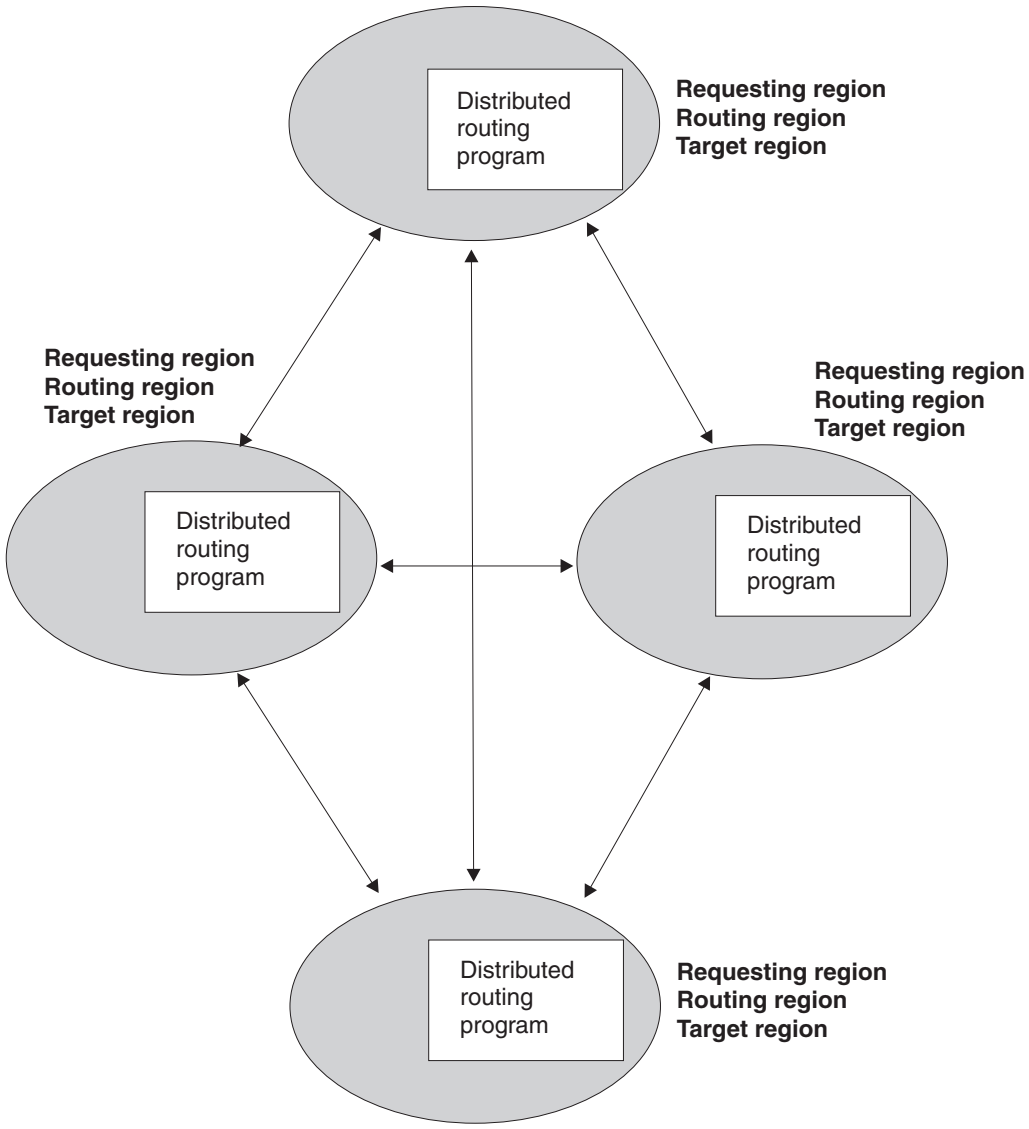


Figure 71. Dynamic routing using a distributed routing model. Each region may be both a routing region and a target region.

The distributed model applies to the routing of:

- BTS processes and activities
- Method requests for enterprise beans and CORBA stateless objects
- Non-terminal-related **EXEC CICS START** requests.

The distributed model is a peer-to-peer system—each participating CICS region may be both a routing region and a target region. A distributed routing program runs in each region.

Advantage of the distributed model

There is no single point-of-failure.

Disadvantages of the distributed model

- Compared to the hub model, there are a great many inter-region connections to maintain.

- You may have problems with distributed data. For example, any data used to make routing decisions must be available to all the regions. CICSPlex SM solves this problem by using dataspaces.

Controlling BTS dynamic routing

You can use the CICSPlex System Manager (CPSM) product or a distributed routing program to control the dynamic routing of your BTS activities.

About this task

You can control the dynamic routing of your BTS activities by either of the following means:

1. Writing your own CICS distributed routing program - see "Using a CICS distributed routing program" on page 156.
2. Using the CICSPlex System Manager (CPSM) product to:
   - Specify workload separation for your BTS processes
   - Manage affinities
   - Control workload routing of the transactions associated with BTS activities.

   See "Using CICSPlex SM with BTS" on page 159.

Creating a BTS-set

When using BTS in a sysplex, you must have several sets of BTS regions (BTS-sets). Each set deals with one or more process-types (types of business transaction). You can create these sets by cloning individual regions.

About this task

Figure 72 on page 155 shows a sysplex that contains two BTS-sets. BTS-set 1 handles all processes of type PAYROLL. All the regions in BTS-set 1 are interconnected and have access to the BTS repository that contains details of PAYROLL-type processes. BTS-set 2 handles all processes of types TRAVEL and MISC. All the regions in BTS-set 2 are interconnected and have access to the BTS repository that contains details of TRAVEL and MISC-type processes.

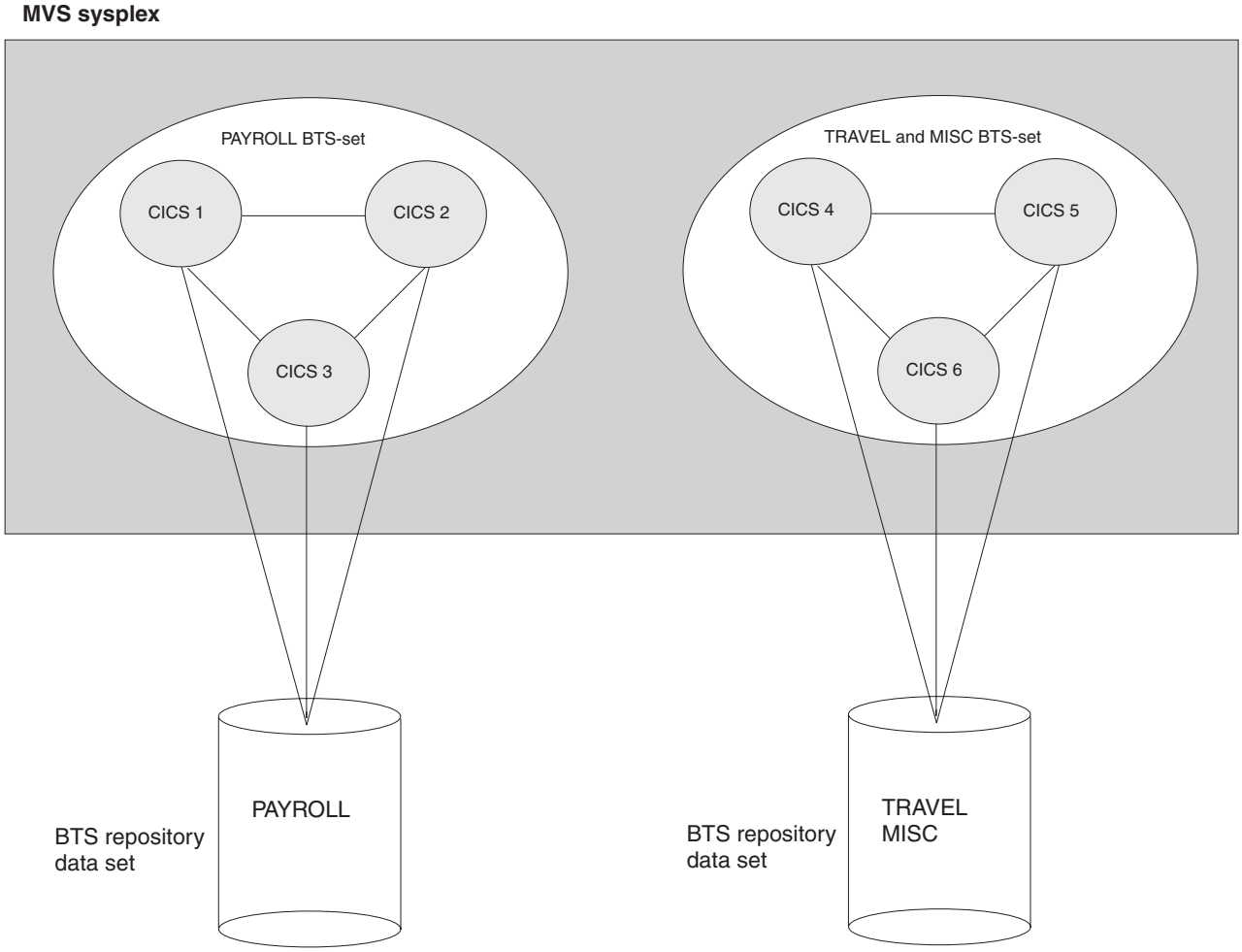


Figure 72. A sysplex containing two BTS-sets. BTS-set 1 handles all processes of type PAYROLL. BTS-set 2 handles all processes of types TRAVEL and MISC.

The number of regions in a BTS-set is related to:

- The number of process-types handled by the BTS-set
- The workload associated with each process-type.

To create each BTS-set, perform the following steps on each of the regions in the set.

Procedure

1. Define a connection to every other region in the BTS-set. For performance reasons, you are recommended to use MRO or MRO/XCF rather than APPC connections.
2. Give the region access to the BTS repository that contains details of the process-types it is servicing. The name of the repository file is specified on the PROCESSTYPE definition or definitions.

   If you are using VSAM RLS to share the repository file, on the FILE definition that defines the repository file to CICS, specify RLSACCESS(YES).

   If you are using function-shipping to share the repository file, on the FILE definition that defines the repository file to CICS, specify REMOTESYSTEM(*name_of_file-owning_region*).

3. On the TRANSACTION definition for each transaction associated with a BTS activity, specify DYNAMIC(YES). Do not specify the REMOTESYSTEM option.

   For general information about defining transactions for transaction routing, and specific information about defining transactions associated with BTS activities, see the *CICS Intercommunication Guide*.
4. Enable the distributed routing program. See the next section.

Naming the routing program

You can specify, discover, and change the distributed routing program.

To specify the distributed routing program, use the **DSRTPGM** system initialization parameter. The name you specify might be that of the CICSPlex SM routing program, EYU9XLOP, or of your own user-written program. For information about **DSRTPGM**, see "Naming the routing program" on page 115.

After CICS has initialized, you can discover which distributed routing program, if any, is in use by issuing an **INQUIRE SYSTEM** command. The DSRTPROGRAM option returns the program name.

After CICS has initialized, you can change the distributed routing program currently in use by issuing a **SET SYSTEM** command. The DSRTPROGRAM option specifies the program name.

Using a CICS distributed routing program

To write your own distributed routing program, you can use the CICS-supplied default distributed routing program, DFHDSRP, as a model.

How the distributed routing program relates to the dynamic routing program

The two CICS-supplied user-replaceable routing programs, DFHDYP and DFHDSRP complement each other. Use DFHDYP for dynamic routing and DFHDSRP for distributed routing. Consider the differences between them.

The dynamic routing program, DFHDYP
Can be used to route:

- Transactions started from terminals
- Transactions started by terminal-related START commands
- Program-link requests.

The distributed routing program, DFHDSRP
Can be used to route:

- BTS processes and activities
- Method requests for enterprise beans and CORBA stateless objects
- Non-terminal-related START requests.

The two routing programs:

1. Are specified on separate system initialization parameters.
2. Are passed the same communications area. Certain fields that are meaningful to one program are not meaningful to the other.
3. Are invoked at similar points - for example, for route selection, route selection error, and (optionally) at termination of the routed transaction or program-link request.

Together, these three factors give you a good deal of flexibility. You could, for example, do any of the following:

1. Use different user-written programs for dynamic and distributed routing.
2. Use the same user-written program for both dynamic and distributed routing.
3. Use a user-written program for dynamic routing and the CICSPlex SM routing program for distributed routing, or vice versa.

The distributed routing program differs from the dynamic routing program in several important ways:

1. The dynamic routing program is invoked only if the resource (the transaction or program) is defined as DYNAMIC(YES). The distributed routing program, however, is invoked (for eligible non-terminal-related START requests, BTS activities, and method requests for enterprise beans and CORBA stateless objects) even if the associated transaction is defined as DYNAMIC(NO) - though it cannot route the request. This means that the distributed routing program is better able to monitor the effect of statically routed requests on the relative workloads of the target regions.
2. Because the dynamic routing program uses the hierarchical "hub" routing model - one routing program controls access to resources on several target regions - *the routing program that is invoked at termination of a routed request is the same that was invoked for route selection.*

   The distributed routing program, however, uses the distributed model, which is a peer-to-peer system - the routing program itself is distributed. *The routing program that is invoked at initiation, termination, or abend of a routed transaction is not the same program that was invoked for route selection* - it is the routing program on the target region.
3. The distributed routing program is invoked at more points than the dynamic routing program. When and where the distributed routing program is called shows the points at which the distributed routing program is invoked, and the region on which each invocation occurs.

Writing a distributed routing program

You can use the CICS-supplied default distributed routing program, DFHDSRP, as a model when writing your own distributed routing program.

For general information about user-replaceable programs, and specific information about how to write a distributed routing program, see *Writing a distributed routing program*, in the *CICS Customization Guide*.

When your routing program is called

For processes and activities started by **RUN ASYNCHRONOUS** commands, your distributed routing program is called at various points on the requesting region and on the target region.

On the requesting region:

1. Either of these activities:
   - For routing the activity. This routing occurs when the transaction associated with the activity is defined as DYNAMIC(YES).
   - For notification of a statically routed activity. This notification occurs when the transaction associated with the activity is defined as DYNAMIC(NO). The routing program is not able to route the activity. It could, however, do other things.

2. If an error occurs in route selection. For example, if the target region returned by the routing program on the route selection call is unavailable. This gives the routing program the opportunity to specify an alternate target. This process iterates until the routing program selects a target that is available or sets a non-zero return code.
3. After CICS has tried (successfully or unsuccessfully) to route the activity to the target region.

   This invocation signals that (unless the requesting region and the target region are one and the same) the responsibility of the requesting region for the transaction has been discharged. The routing program might, for example, use this invocation to release any resources that it has acquired on behalf of the transaction.

On the target region:

These invocations occur only if the routing program on the requesting region has specified that it must be called again on the target region:

1. When the activation starts on the target region (that is, when the transaction that implements the activity starts).
2. If the routed activation (transaction) ends successfully.
3. If the routed activation (transaction) abends.

Figure 73 shows the points at which the distributed routing program is called, and the region on which each invocation occurs. The "target region" is not necessarily remote - it could be the local (requesting) region, if the routing program chooses to run the activity locally.

Figure 73. When and where the distributed routing program is called

When it is called on the target region for transaction initiation, termination, or abend, the routing program can update a count of BTS activities that are currently running on that region. When it is called on the requesting region for route selection, the routing program can use the counts maintained by all the regions in the routing set (including itself) as input to its routing decision. This routing decision requires that all the regions in the BTS-set have access to a common data store on which the counts are stored. For further details, see *Writing a distributed routing program*, in the *CICS Customization Guide*.

Restrictions on the routing program

The distributed routing program runs outside a unit of work environment. Therefore your program must not alter any recoverable resources or issue file control or temporary storage requests.

Using CICSPlex SM with BTS

You can use CICSPlex SM services to route BTS activities around your BTS-sets, avoiding the need to write your own distributed routing program.

Overview of CICSPlex SM workload management

You can use CICSPlex SM workload management (WLM) functions to route transactions defined as dynamic from a requesting region to a suitable target region selected at the time the transaction is started.

WLM functions

The CICSPlex SM dynamic routing program supports functions:

Workload separation
Routes specific transactions to a group of target regions based on BTS process-type, or a combination of process type and transaction name, or any combination of user ID, terminal ID, and transaction name. For example, using the CICSPlex SM workload separation function, you can specify that transactions beginning with the characters 'SAL' and initiated by members of your sales department must be routed to the group of target regions, SALESGRP, allocated to that department.

Workload routing
Routes transactions amongst a group of target regions according to the availability and activity levels of those regions. You can use workload routing in addition to, or in place of, workload separation. For example, CICSPlex SM can route the transaction workload among the SALESGRP regions by selecting, as each transaction is initiated, the target region that is likely to deliver the best performance.

Intertransaction affinity
Requires related transactions to be processed by the same target region. You can use IBM CICS Interdependency Analyzer for z/OS to identify affinities between transactions.

For further introductory information about CICSPlex SM and workload management, see the *CICSPlex System Manager Concepts and Planning*.

Using CICSPlex SM to route BTS activities

You can use CICSPlex SM to route BTS Activities around a BTS-set. The routing is based on the workload separation criteria you define. Take note of any affinities that might be introduced.

When routing BTS activities around a BTS-set, CICSPlex SM Workload Management selects a target region based on:

- Any workload separation criteria that you have defined
- The current workloads of the eligible regions
- Any active affinities

- The speed of the communication links to the eligible regions.

The CICSPlex SM component of CICS Transaction Server for OS/390, Version 1 Release 3 understands BTS processes and activities. This makes it possible to separate a BTS workload based on process-type. For example, you could specify that WLM is to route all processes of process-type 'TRAVEL' to one region in the BTS-set, and all processes of type 'PAYROLL' to another region.

CICSPlex SM WLM and the IBM CICS Interdependency Analyzer for z/OS understand affinities between BTS activities and processes. Although BTS itself does not introduce any affinities, and discourages programming techniques that do, it does support legacy code, which might introduce affinities. You must define such affinities to CICSPlex SM WLM, so that it is able to make sensible routing decisions. It is important to specify the lifetime of each affinity; failure to do so might restrict WLM routing options unnecessarily.

Note that:

- A single CICSPlex SM can control routing within multiple BTS-sets. It cannot route activities *across* BTS-sets.
- Workload separation can be performed at two levels:
  1. By creating multiple BTS-sets.
  2. By CICSPlex SM within a BTS-set.

Part 4. BTS problems and performance

Solve BTS-related problems and measure and tune BTS performance with this information.

© Copyright IBM Corp. 1999, 2011

Chapter 15. BTS problem determination

Try these solutions for some of the more common problems that you might encounter when running a BTS system.

Dealing with stuck processes

Consider the causes of a "stuck" process, when it cannot proceed because it is waiting for an event that cannot, or does not, occur. You can use timers or process timers to restart a stuck process.

There are several possible causes:

- A faulty application design - see "Application design errors."
- A request to start an activity on a remote system is "unserviceable" - see "Dealing with unserviceable requests" on page 167.
- A CICS region fails - see "Dealing with CICS failures" on page 168.

Application design errors

A stuck process might be caused by a program logic error.

For example, consider the following scenarios:

1. **Outstanding user events:**
   a. One of the activities of the process returns from what it believes to be its final activation. It issues an EXEC CICS RETURN command without the ENDACTIVITY option.
   b. There are no events on the reattachment queue of the activity, but there is a user event in its event pool.
   c. There is no means for the event to be fired. Perhaps it is an input event which has fired, caused reattachment, and been retrieved, but which the activity has neglected to delete.

   In this case, the activity becomes dormant, and there is no way for it to reactivated. The process is stuck.

   The recommended way to prevent this scenario is to add the ENDACTIVITY option to the EXEC CICS RETURN command that ends the final activation of the activity. Coding RETURN ENDACTIVITY deletes any outstanding events—other than activity completion events for child activities, which the activity must deal with properly—and allows the activity to complete normally.
2. **Waiting for an external interaction:**

   A user-related activity returns from its initial activation and becomes dormant, waiting for an external interaction to occur. (User-related activities are described in "Activity processing" on page 69.) However, the expected user input does not happen. Perhaps the clerk is sick, or the data required is not available. The process is stuck.

   You can recover from this scenario by setting a timer which, if the expected external interaction does not occur within a specified period, causes the activity (or its parent) to be reactivated anyway.
3. **Timer error:**

© Copyright IBM Corp. 1999, 2011

A programming error results in a timer being set to expire in five days rather than 5 minutes. The process is stuck. See “Restarting stuck processes.”

Note: To force a timer to expire before its specified time, use the FORCE TIMER command.

Restarting stuck processes

You can use activity timers or process timers to restart a stuck process. You can use status containers to identify stuck activities. You can use a utility program to help you determine the status of a process.

About this task

For advice on restarting processes that are stuck because of unserviceable requests, see “Dealing with unserviceable requests” on page 167.

For advice on restarting processes that are stuck because of a CICS failure, see “Dealing with CICS failures” on page 168.

Using activity timers

Use activity timers to restart processes that are stuck because of application errors or other reasons. For example, a parent might set a timer that causes it to be reactivated after a specified period, if a particular child activity does not complete.

About this task

The best way to restart processes that are stuck for other reasons - including application errors - is to use timers. The parent names the timer in a way that associates it with a particular child. If the child completes within the specified period, the parent deletes the timer.

One reason for making the application responsible for restarting itself is that it is difficult from *outside* a process to tell whether the process is stuck or merely dormant, particularly if the process is long-lived. Processes of different types can have varying “natural” lifespans; and these lifespans can vary according to system load, availability of remote regions, and so on. The application itself is best placed to know how long each of its activities should run before they can be assumed to be stuck.

You probably do not want to set timers for all your activities. For example, you might think it unnecessary to set a timer for a simple activity that completes its processing in one activation, has no children, and is to be run synchronously. However, you might want to set a timer for an activity to which one or more of the following apply:

- It is to be run asynchronously.
- It requires multiple activations to complete its processing.
- It is long-lived.
- It involves external interaction—for example, user input.

Using process timers

Use process timers to restart a stuck process. You can set timers for individual child activities, for the process, or for both. That is, the root activity can set a timer with an expiry time after the whole process might be expected to complete.

About this task

If the process is short-lived, you might decide not to set any activity timers, but to set a process timer instead.

If the process is long-lived, do not set a process timer without also setting timers for at least some individual activities. Setting timers prevents the possibility of a delay in restarting the process. For example, if a process that is expected to last six months becomes stuck after one day while processing its first activity, and you have set only a process timer, the process could lie dormant for, say, seven months before the root activity is reactivated to deal with the problem.

If the root activity is activated by the process timer, it could, for example:

1. Browse and inquire on each of its descendant activities, checking completion status and mode. For examples of the use of the BTS browsing and inquiry commands, see "Browsing examples" on page 219.
2. If it succeeds in identifying the stuck activity, issue a CANCEL command to cancel it. If the stuck activity is not a child but a lower-level descendant of the root activity, the root must first acquire the stuck activity.
3. The completion event of the stuck activity fires, causing the parent activity to be reactivated. The CHECK ACTIVITY command issued by the parent returns a completion status of FORCED. The parent should be coded to handle the abnormal completion of one of its children. The process is no longer stuck.

Using status containers

Use status containers to make it easier for a root activity to identify which of its descendant activities are stuck.

About this task

Status containers are data-containers that contain information about what an activity is currently doing. Whereas you can use an INQUIRE ACTIVITYID command to discover the mode and completion status of an activity, the information in a status container is likely to be at a more detailed level. For example, each activity in a process might have a data-container called, perhaps, STATUS, which it regularly updates—perhaps at the beginning and end of each activation, and each time it starts new work. A status container might, for instance, contain the date and time, and a string describing the work that the activity has started or ended, or the fact that it is dormant because it is waiting for the completion of a particular child activity.

You can think of an activity as a finite state machine—it is always in one of a limited number of processing states. The "processing states" we refer to here are application-dependent and distinct from the BTS-defined *modes* of an activity. Each activity could regularly update its status container with its current processing state.

Using a utility program

You can use a utility program to help you determine whether a process is stuck or just dormant. CICS supplies the audit trail utility, DFHATUP, and the repository utility, DFHBARUP, or you can write your own.

CICS-supplied utility programs

CICS supplies two utility programs for diagnostic purposes. The audit trail utility, DFHATUP, and the repository utility, DFHBARUP.

The two utility programs for diagnostic purposes are detailed here:

The audit trail utility, DFHATUP
You can use DFHATUP to print selected audit records from a log stream. If you use auditing to track the progress of your processes across the sysplex, to investigate a stuck process you could print its audit records.

DFHATUP is described in Chapter 16, “Creating a BTS audit trail,” on page 169.

The repository utility, DFHBARUP
You can use DFHBARUP to print selected records from a repository. To investigate a stuck process, you could print its repository records.

DFHBARUP is described in Chapter 17, “Examining BTS repository records,” on page 191.

User-written utility programs

You can write a utility program that can check for and restart stuck processes, the utility program is most effective when your activities use status containers.

Your utility program could, for example:

1. Browse all processes of a specified process-type.
2. Browse the descendant activities of each process returned in step 1.
3. Inquire on the status data-container of each activity, and retrieve its contents.
4. Identify a stuck activity from the contents of its status container.
5. Issue an ACQUIRE command to acquire the stuck activity.
6. Issue a CANCEL command to cancel the stuck activity. The completion event for the stuck activity fires, causing its parent to be reactivated. The CHECK ACTIVITY command issued by the parent returns a completion status of FORCED. Ensure that the parent is coded to handle the abnormal completion of one of its children. The process is no longer stuck.

Dealing with activity abends

You can deal with activity abends by coding your application to try the failed activity again or compensate the siblings of the failed activity.

About this task

Compensation is the act of modifying, or compensating for, the effects of a completed activity. For example, compensation can reverse or modify actions taken by preceding activities or terminate business transactions. If a program that implements an activity abends, the parent of the activity receives control. (If the failed activity was run asynchronously, the parent is reactivated.) The **CHECK ACTIVITY** command issued by the parent returns a COMPSTATUS of ABEND - see Chapter 5, “Dealing with BTS errors and response codes,” on page 49.

Your application should be coded to deal with an activity abend. The parent of the failed activity might, for example, choose to do either of the following:

- Try the failed activity again - see “Trying failed activities again” on page 52
- Compensate the siblings of the failed activity - see Chapter 8, “Compensation in BTS,” on page 85.

Dealing with unserviceable requests

An *unserviceable request* is a request that cannot currently be satisfied; for example, an activity that is not available, or the region on which the request is to run is not accessible.

Unserviceable routing requests

If you operate BTS in a sysplex, you can route processes and activities across a set of CICS regions called a *BTS-set*. When a process or activity is started by a RUN ASYNCHRONOUS command, it can be routed either statically or dynamically. Mostly, you might choose dynamic rather than static routing.

For detailed information about routing processes and activities, see Chapter 14, "Using BTS in a sysplex," on page 149.

Static routing

Using static routing, you name the target region to which the activity is to be routed on the REMOTESYSTEM option of the installed transaction definition, for the transaction associated with the activity. If the target region is unavailable at the time the activity is to be started, CICS treats the request as unserviceable.

Dynamic routing

Using dynamic routing, the target region is chosen by your routing program, the distributed routing program or the CICSPlex SM routing program. If the target region that it returns is unavailable, the routing program is invoked again and can select a different target. Alternatively, it can (by setting a non-zero return code) indicate that the request is to be treated as unserviceable.

For definitive information about writing a distributed routing program, see *Writing a distributed routing program*, in the *CICS Customization Guide*.

Why classify requests as unserviceable?

Why should your routing program classify requests as "unserviceable"? Why should it not reroute the request to an alternative region, assuming that alternatives are available?

Sometimes, due to a transaction affinity, it might be essential that an activation runs on a specific region, and no other. If so, rather than selecting an alternative target region, your routing program can return the same target (even though it is currently unavailable), and classify the request as unserviceable.

How CICS handles unserviceable requests

CICS identifies unserviceable requests by issuing a DFHSH message and tries the request again every minute.

When a request is "unserviceable", CICS:

1. Issues message DFHSH0105, which identifies the request and indicates that it cannot be serviced.
2. Tries the request again every minute. If the request is successfully serviced, CICS issues message DFHSH0108.

3. Each hour, if the request still cannot be serviced, issues message DFHSH0106. This message indicates the time remaining before CICS purges the request, if it has not been serviced in the meantime.
4. After 24 hours, if the request still cannot be serviced, stops trying to service it and issues message DFHSH0107. The request is discarded.

Resolving unserviceable requests

In many cases, CICS resolves unserviceable requests automatically. If, for example, an unavailable target region becomes available within 24 hours of the request being issued, CICS routes the request correctly.

You should watch for occurrences of DFHSH0105 and DFHSH0106 messages. You should investigate why the request is unserviceable, and take any necessary corrective action. It might be, for example, that a resource required to satisfy the request (an activity or process) is inaccessible; or that a remote region, or a link to it, is unavailable.

Dealing with CICS failures

If one of your CICS regions fails, all BTS processes on the failing region are halted. Processes on other regions might also become stuck, because expected events are not generated. If a CICS region fails, you must perform an emergency restart.

Only in rare circumstances - for example, if the CICS global catalog or system log is corrupted - might it be necessary to perform an initial or cold start after a failure. If it *is* necessary, perform a cold start in preference to an initial start.

Emergency starts

During an emergency restart, CICS automatically restores BTS processes to the state they were in before the failure. Any activities that were active at the time of the failure are rerun. That is, if an activation (transaction) was running, it is backed out and restarted. The activity is sent the same reattachment event that caused the failed activation. Its data-containers contain the same data they held at the start of the failed activation.

Initial and cold starts

During an initial or cold start:

- BTS repository data sets are unchanged.

  Note: Repository data sets are never reinitialized at CICS startup, because they might be shared.
- The local request queue data set is unchanged. All information about BTS timers, pending, and unserviceable requests is preserved. However, it is likely that some of this information is now irrelevant or invalid, because it refers to processes that no longer exist.

Chapter 16. Creating a BTS audit trail

You can create an audit trail for BTS processes and activities to track the progress of transactions. You can control the amount of audit logging. Follow the example to see a series of activations.

About this task

It contains:

- "Introduction to BTS audit trails"
- "Specifying the level of audit logging" on page 170
- "Audit trail constraints when using DASD-only log streams" on page 173
- "Audit trail examples" on page 173
- "Using the audit trail utility program, DFHATUP" on page 176.

Introduction to BTS audit trails

You can create an audit trail for the BTS processes and activities that run in your CICS systems. You can use the audit trail to track the progress of complex business transactions and to diagnose problems in programs that are being developed to form a new business application.

The CICS code contains BTS audit points in much the same way as it contains trace points. However, there are three main differences between audit records and trace entries:

1. Trace entries are written to an internal trace table within the CICS address space. In contrast, the audit trail of a process is written to a CICS journal, which resides on an MVS log stream.
2. Trace entries record the progress of tasks over a relatively short period, typically seconds, minutes, or hours. In contrast, the audit trail of a process can extend to days, weeks, or even months.
3. Trace entries relate to activity in a single CICS region. In contrast, in a sysplex the execution of different parts of a process might take place on different regions within the sysplex. Therefore, each audit record contains system, date, and time information. Typically, an audit record for a BTS activity also contains:
   - The identifier of the activity
   - The process to which the activity belongs
   - Information about the event which caused the activity to be invoked, canceled, suspended, or resumed; or that fired when it completed.

   Because log streams can be shared by more than one region, it is possible to write audit records from different regions to the same log.

There are four, incremental, auditing levels:

1. None
2. Process-level
3. Activity-level
4. Full.

© Copyright IBM Corp. 1999, 2011

How to specify the levels, and what they mean, is described in "Specifying the level of audit logging."

Audit log records are written to an MVS log stream by the CICS Log Manager. You can read the records offline using the CICS audit trail utility program, DFHATUP. DFHATUP allows you to:

- Filter records for specific process-types, processes, and activities
- Interpret records into a readable format.

You can use the CICS journal utility program, DFHJUP, to copy the audit log stream to a backup file and to delete the log stream. By editing the JCL used to run DFHATUP, you can make DFHATUP accept the backup file as input.

Audit records are buffered; they are written to the log stream only when the buffer is full or a sync point occurs. This means that, when multiple CICS regions share the same log stream, audit records might not be in exact date and time order.

Specifying the level of audit logging

You can control the amount of audit logging that CICS performs for each process, using the AUDITLOG and AUDITLEVEL attributes of the PROCESSTYPE definition.

For detailed information about defining process-types, see "CEDA DEFINE PROCESSTYPE" on page 119. However, note the following considerations:

- When a process is first defined, BTS obtains the audit level and audit log information for the process from the installed PROCESSTYPE definition, and copies it into the process record. During the lifetime of the process, this copy of the audit information is used to determine auditing. If the auditing information is changed by, for example, a CEMT SET PROCESSTYPE command, this does not affect existing processes.
- If an installed PROCESSTYPE definition does not specify a CICS journal name in its AUDITLOG field, CICS does not do any audit logging for processes and activities of that type until the definition is replaced with one that does contain the name of an audit log.
- The AUDITLOG field must not specify the SMF data set.
- Several process-types can share the same audit log.
- In a sysplex, different parts of a process might run on different CICS regions. If you want to write audit records for all the parts, you must ensure that all the regions have the same audit log information in their installed PROCESSTYPE definitions. However, see "Audit trail constraints when using DASD-only log streams" on page 173.

The AUDITLEVEL option of the PROCESSTYPE definition allows you to specify one of four logging levels for processes of the defined type:

ACTIVITY
Specifies activity-level auditing. Audit records are written from:

1. The process audit points
2. The activity primary audit points.

That is, an audit record is written:

1. Whenever a process of this type:
   - Is defined

- Is requested to run
- Is requested to link
- Is acquired
- Completes
- Is reset
- Is canceled
- Is suspended
- Is resumed

2. Each time data is placed in a process container belonging to a process of this type—that is, each time a PUT CONTAINER PROCESS or PUT CONTAINER ACQPROCESS command is issued against a process of this type
3. Each time a process container belonging to a process of this type is deleted
4. Each time a root activity (DFHROOT) of this type of process is activated.
5. Every time a non-root activity belonging to a process of this type:
   - Is requested to link
   - Is activated
   - Completes.

FULL Specifies full auditing. Audit records are written from:

1. The process audit points
2. The activity primary *and* secondary audit points.

That is, an audit record is written:

1. Whenever a process of this type:
   - Is defined
   - Is requested to run
   - Is requested to link
   - Is acquired
   - Completes
   - Is reset
   - Is canceled
   - Is suspended
   - Is resumed

   Each time data is placed in a process container belonging to a process of this type

   Each time a process container belonging to a process of this type is deleted

   Each time a root activity (DFHROOT) of this type of process is activated
2. Every time a non-root activity belonging to a process of this type:
   - Is defined
   - Is requested to run
   - Is requested to link
   - Is activated
   - Completes

- Is acquired
- Is reset
- Is canceled
- Is suspended
- Is resumed
- Is deleted.

OFF Specifies that no audit trail records are written. This is the default value.

PROCESS

Specifies process-level auditing. Audit records are written from the process audit points only. That is, an audit record is written whenever a process of this type:

- Is defined
- Is requested to run
- Is requested to link
- Is acquired
- Completes
- Is reset
- Is canceled
- Is suspended
- Is resumed

Each time data is placed in a process container belonging to a process of this type

Each time a process container belonging to a process of this type is deleted

Each time a root activity (DFHROOT) of this type of process is activated

Note: If you specify any value for AUDITLEVEL other than OFF, you must also specify the AUDITLOG option of the PROCESSTYPE definition.

You must choose a level of auditing that suits your needs. The more records that are written to the audit log, the longer your business transaction takes to run. The fewer records written, the less information there is for auditing or diagnostic purposes.

To reset the AUDITLEVEL attribute of an installed PROCESSTYPE definition, use the CEMT SET PROCESSTYPE command. Changes are preserved across a restart of CICS. Changes to an installed PROCESSTYPE definition do not affect *existing* processes.

If a request to write an audit record fails:

- CICS issues an error message.
- Auditing for processes of this process-type is suspended until the audit error is corrected and a CEMT SET JOURNALNAME(*journal*) ACTION(RESET) command is issued. If the reset completes successfully, auditing is resumed and a CICS message is issued to this effect. Some audit records are lost.

Audit trail constraints when using DASD-only log streams

If you are running BTS in a sysplex, the activities that make up a process might run on different CICS regions. If you want to use audit logging, you must ensure that audit records can be written to a single log stream from any region on which any of the activities run.

About this task

If the CICS regions are in the same MVS image, you can define the log stream to use either a coupling facility structure or DASD-only logging. However, if the CICS regions are on *different* MVS images, the log stream must use a coupling facility structure rather than DASD-only logging. This is because CICS regions on different MVS images cannot access the same DASD-only log stream at the same time.

If the regions are in different MVS images and you use DASD-only logging, you are unable to use shared log streams for your BTS logs. This means that audit records for a single process might be split across several log streams; you must collate them yourself.

Audit trail examples

Follow the sequence of activations of a BTS process, SALES1234567890. The activities that make up the process run on two CICS regions.

For clarity, the example does not show the activations of any other processes that might also be running in these regions.

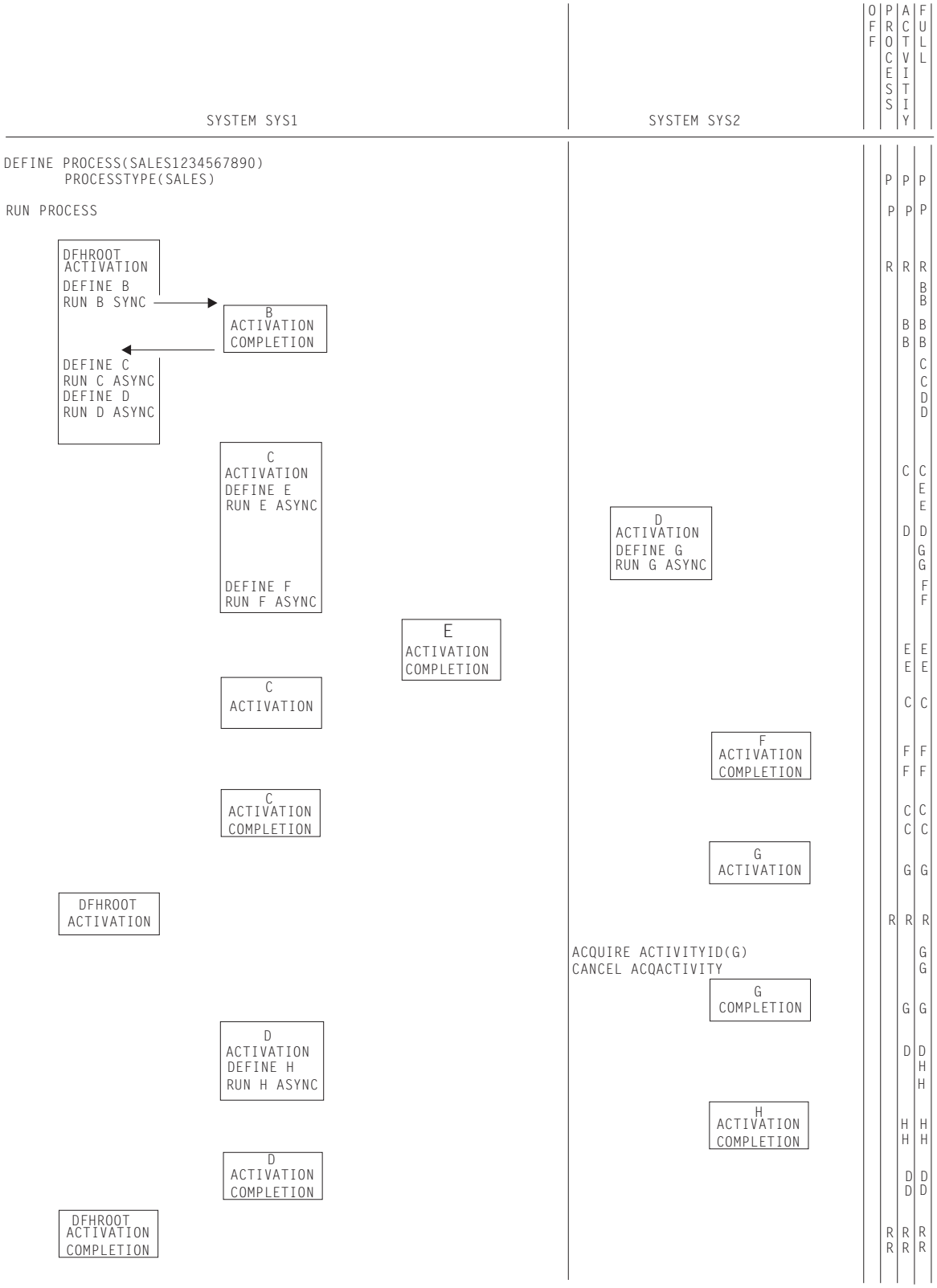


Figure 74. Example audit trails. The columns on the right show, for each audit level setting, the points at which audit records would be written. The letters in these columns are the names of the activities for which records are being written.

In this example, an application running on region SYS1 defines a new process, SALES1234567890, and requests it to run. The root activity of the new process begins running on SYS1. It defines and runs an activity B, which executes synchronously. When control returns to the root activity, it defines activities C and D and schedules them to run asynchronously. After the root activity has returned, activity C starts on SYS1 and activity D starts on SYS2.

Activity C schedules child activities E and F to run asynchronously and returns. E and F run on different systems. When each of its child activities completes, C is reactivated and checks the completion status of the child. Lastly, C completes normally, which causes the root activity to be reactivated.

Activity D defines a child activity G and schedules it to run asynchronously. Later, another transaction issues ACQUIRE ACTIVITYID and CANCEL ACQACTIVITY commands against activity G. G completes in a FORCED state. D is reactivated and discovers what has happened to G with a CHECK ACTIVITY command. In response to G's failure, D defines a new activity H and requests it to run asynchronously. D then returns and H runs on the other region. When H completes normally, D is reactivated and completes normally. This causes the root activity to be reactivated. The root activity issues a CHECK ACTIVITY command to see how D completed, and then completes normally, ending the process.

Note: For conciseness, some commands that could result in audit records being written - for example, PUT CONTAINER ACQPROCESS and SUSPEND - are omitted from the example.

Process-level auditing

A setting of PROCESS on the AUDITLEVEL attribute of a PROCESSTYPE definition specifies process-level auditing for processes of the defined type. Records are written from the audit points for processes.

If process-level auditing is set for the process in the example, only six records are written to the audit log (see Figure 74 on page 174):

1. When the process is defined
2. When the process is requested to run
3. When the root activity of the process is activated for the first time
4. When the root activity of the process is activated for the second time
5. When the root activity of the process is activated for the third time
6. When the process completes.

Activity-level auditing

A setting of ACTIVITY on the AUDITLEVEL attribute of a PROCESSTYPE definition specifies activity-level auditing for processes of the defined type.

Records are written from:

- The audit points for processes
- The primary audit points for activities.

If activity-level auditing is set for the process in the example, the following records are written to the audit log:

- The six records described in “Process-level auditing.”
- Each time one of DFHROOTs descendant activities is activated.

- When each descendant activity completes. This includes the completion of activity G, which has a completion status of FORCED.

  Note: Records are *not* written when an activation ends in an incomplete state. Thus, in the example, a record is not issued when the root activity ends after defining activity D.

Full auditing

Set FULL on the AUDITLEVEL attribute of a PROCESSTYPE definition to write full audit information to the audit log. Note the effect on performance.

Records are written from:

- The audit points for processes
- The primary audit points for activities
- The secondary audit points for activities.

If full auditing is set for the process in the example, the following records are written to the audit log:

- All those written for activity-level auditing
- When each activity is defined
- When each activity is scheduled to run
- When activity G is acquired
- When activity G is canceled.

Note: Full auditing has an adverse effect on performance. It is intended to provide the maximum amount of information to help track down problems when applications are being developed. It is not intended to be used on production systems.

Using the audit trail utility program, DFHATUP

You can use the audit trail utility program, DFHATUP, to read BTS audit records from a log stream and to print them. The DFHATUP utility formats the records to make them easier to interpret. You can also filter selected records.

Using DFHATUP to read audit logs

You can use the DFHATUP batch utility to capture, format, and display records from your audit logs.

About this task

You must run DFHATUP as a batch job against a log stream that is not in use by any CICS regions. If you run it against a log stream that is connected to CICS, DFHATUP is unable to find any records that CICS has in its buffers.

DFHATUP reads the records in the order that they were written to the MVS log stream. By including control statements in the SYSIN data set, you can select the records that DFHATUP writes to the output data set, SYSPRINT. DFHATUP formats the selected records before writing them to SYSPRINT.

DFHATUP ignores any records that it does not recognize as BTS audit records.

Sample job stream to run the DFHATUP program

Follow this example job stream to see how you can use the DFHATUP utility program to process your audit log data.

Figure 75 shows an example job stream to run the DFHATUP program. The job stream must include DD statements for the following data sets:

The audit log
The audit log data set to be examined to produce the output data. (Figure 75 shows a DD name of 'AUDITLOG'.)

If you do not specify the BLKSIZE parameter its value defaults to 80, which causes audit records to be truncated.

STEPLIB
A partitioned data set (DSORG=PO) that contains the DFHATUP program module. If the module is in a library in the link list, this statement is not required.

SYSIN
The input control data set. This file must be in 80 - byte record format. The control statements that you can use in this data set are described in "SYSIN control statements" on page 178.

Control statements can be continued on to the next line by including any non-blank character in column 72. If the line that follows a continuation character is empty or contains control arguments which conflict with those control arguments that make up the preceding part of the control statement, an error is reported and execution of the utility ends. Any characters which occur beyond column 72 are ignored.

SYSPRINT
The output data set which the formatted audit records and control messages are sent to.

```
//*******************************************************************
//* RUN DFHATUP (AUDIT LOG UTILITY PROGRAM)
//*
//*
//*******************************************************************
//ATUP     EXEC PGM=DFHATUP,PARM='N(EN),P(30),T(M)'
//STEPLIB    DD DSN=CTS130.CICS530.SDFHLOAD,DISP=SHR
//*******************************************************************
//*   The output will go to SYSPRINT
//*******************************************************************
//SYSPRINT DD SYSOUT=A,DCB=RECFM=FBA
//AUDITLOG DD DSN=CICSAA#.CICSDC1.JRNL001,
//            SUBSYS=(LOGR,DFHLGCNV),
//            DCB=BLKSIZE=32760
//SYSIN    DD *
PTYPE(SALES)                                                          +
PROCESS(CUST_SALES_1999.13872977829728.QA)
ACTIVITY(activity-name)
PROCESS(CUST_SALES_1999.11103847635637.QB)                            +
PTYPE(SALES)
/*
//*
```

Figure 75. Sample job to run the DFHATUP utility program

EXEC parameters

You can use the PARM keyword on the EXEC statement to pass national language, pagesize, or translate case parameters to the DFHATUP utility.

The form of the EXEC statement is:

```
EXEC PGM=DFHATUP,PARM='parm1,...,parmn'
```

NATLANG({EN|*CS*v*KA***})**
The language in which messages are to be issued.

The minimum abbreviation of this parameter is **N**. The possible values are:

CS Traditional Chinese

EN English. This is the default.

KA Kanji.

PAGESIZE({60|*nn***})**
The number of lines to be printed per page, when the output from the utility is sent to a printer. Valid values are in the range 20–99. The default is 60.

The minimum abbreviation of this parameter is **P**.

TRANSLATE({MIXEDCASE|*UPPERCASE***})**
Whether the output from the utility is to be in mixed-case or uppercase. The default is mixed-case.

The minimum abbreviation of this parameter is **T**. The minimum abbreviations of MIXEDCASE and UPPERCASE are M and U respectively.

SYSIN control statements

You can use the SYSIN data set to pass information to the audit trail utility program, DFHATUP. For example, you can include statements to select specific sets of records to be formatted.

Comments are identified by an asterisk (*) in the first position, anything entered on the SYSIN card after the asterisk is ignored by DFHATUP. The SYSIN data set must be defined.

Format of the SYSIN control statements:

Use the SYSIN control statements to specify the name of an activity, the audit data log, the BTS process, and the process type.

```
SYSIN DD *
[AUDITLOG(name)]
[PTYPE(name) <PROCESS(name)>]
[PROCESS(name)]
[ACTIVITY(name)]
```

An AUDITLOG statement cannot contain additional arguments. Other statements might consist of multiple arguments. When using multiple arguments, put each argument on a separate line; use a non-blank character in column 72 to indicate that this argument and the following one are to be treated as a single control statement. An illegal combination of arguments generates an error message and the utility is not run against the log stream.

ACTIVITY(*name***)**
The 1-16 name of an activity. Records for this activity are formatted. No further arguments are needed to make up a control statement; if none are provided, all

audit records containing this activity name are selected. To limit the scope of the search, you can add a PTYPE argument, a PROCESS argument, or both on adjoining lines, using a continuation character in column 72.

AUDITLOG(*name***)**
The 1-8 character DD name that identifies the audit log data set to be searched. The default is 'AUDITLOG'. This argument must not be specified more than once. It cannot be used with any other in a control statement.

If the specified audit log cannot be located or connected to, or if more than one AUDITLOG statement is found in the SYSIN data set, an error occurs and DFHATUP terminates.

PROCESS(*name***)**
The 1-36 character name of a BTS process. No further arguments are needed to make up a control statement; if none are provided, all audit records containing this process name are selected. To limit the scope of the search, you can add a PTYPE argument, an ACTIVITY argument, or both on adjoining lines, using a continuation character in column 72.

PTYPE(*name***)**
The 1-8 character name of a BTS process-type. No additional arguments are needed; if none are provided, all audit records containing this process-type are selected. To limit the scope of the search, you can add a PROCESS argument, an ACTIVITY argument, or both on adjoining lines, using a continuation character in column 72.

Example output from the DFHATUP utility

Follow these example outputs of different levels of audit trail auditing from the DFHATUP audit utility.

CICS writes records to an audit log in chronological order. Particularly on busy systems in a sysplex, records from different processes or from different activities in the same process are likely to become interleaved. In order to find out what has taken place during the execution of a specific process, you might want to select particular sets of records.

"Audit trail examples" on page 173 shows the points at which records are written to an audit log, depending on the level of auditing specified for the relevant process-type. The example control statements in Figure 76 format all the records written to the audit log for the SALES1234567890 process (which is of the SALES process-type).

```
//SALESLOG DD DSN=CICSAA#.CICSDC1.JRNL001,
//            SUBSYS=(LOGR,DFHLGCNV),
//            DCB=BLKSIZE=32760
//SYSIN    DD *
AUDITLOG(SALESLOG)
PTYPE(SALES)                                                    +
PROCESS(SALES1234567890)
/*
//*
```

Figure 76. Example control statements, to format all the records for the SALES1234567890 process

Example audit trail process-level auditing

Follow this example audit trail to see the output that is produced if the AUDITLEVEL attribute of the PROCESSTYPE definition for the SALES process-type is set to 'PROCESS'.

Extending our previous example, Figure 77 shows the output that would be produced if:

- On both regions SYS1 and SYS2, the AUDITLEVEL attribute of the PROCESSTYPE definition for the SALES process-type is set to 'PROCESS'
- The control statements in the SYSIN data set specify that all records for the SALES1234567890 process must be formatted.

```
CBTS Audit Trail Utility - Parameter Validation                                  Date : 29/01/1999 Time : 15:59:30 Page 000001

Exec Parm Options: Natlang   (EN)
                   Translate (mixedcase)
                   Pagesize  (60)



CBTS Audit Trail Utility - Audit Print                                           Date : 29/01/1999 Time : 15:59:30 Page 000002

Ptype(SALES   ) Function(Define Process  ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000072) Activity(DFHROOT        ) Transid(R   ) Program(R       ) Userid(CICSUSER)
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39....;f..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB3235800CCCDDDE444444444)
                                                                  (21469351A172924981B98363339AA51E60146896630000000000)
  Current: Transid(P   ) Program(P       ) Userid(CICSUSER) Date(1999.029) Time(15:59:20.798300)

Ptype(SALES   ) Function(Run Process     ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000072) Activity(DFHROOT        ) Asynchronous
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39....;f..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB3235800CCCDDDE444444444)
                                                                  (21469351A172924981B98363339AA51E60146896630000000000)
  Current: Transid(P   ) Program(P       ) Userid(CICSUSER) Date(1999.029) Time(15:59:20.798565)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000073) Activity(DFHROOT        ) Event(DFHINITIAL      )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39....;f..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB3235800CCCDDDE444444444)
                                                                  (21469351A172924981B98363339AA51E60146896630000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(15:59:20.865320)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000082) Activity(DFHROOT        ) Event(C               )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39....;f..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB3235800CCCDDDE444444444)
                                                                  (21469351A172924981B98363339AA51E60146896630000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(15:59:25.978683)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000087) Activity(DFHROOT        ) Event(D               )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39....;f..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB3235800CCCDDDE444444444)
                                                                  (21469351A172924981B98363339AA51E60146896630000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(15:59:26.824560)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000087) Activity(DFHROOT        ) Compstatus(Normal    )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39....;f..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB3235800CCCDDDE444444444)
                                                                  (21469351A172924981B98363339AA51E60146896630000000000)
  Current: Transid(R   ) Userid(CICSUSER) Date(1999.029) Time(15:59:26.849330)
CBTS Audit Trail Utility - Selection Results                                     Date : 29/01/1999 Time : 15:59:30 Page 000003

Number of Audit records read :         6
Number of records selected   :         6
  Processing Complete
```

Figure 77. Example audit trail, showing the types of record written for process-level auditing

Example audit trail activity-level auditing

Follow this example audit trail to see the output that is produced if the AUDITLEVEL attribute of the PROCESSTYPE definition for the SALES process-type is set to 'ACTIVITY'.

Figure 78 shows the output that is produced if:

- On both regions SYS1 and SYS2, the AUDITLEVEL attribute of the PROCESSTYPE definition for the SALES process-type was set to 'ACTIVITY'
- The control statements in the SYSIN data set specify that all records for the SALES1234567890 process must be formatted.

```
CBTS Audit Trail Utility - Parameter Validation                                  Date : 29/01/1999 Time : 15:24:02 Page 000001

Exec Parm Options: Natlang   (EN)
                   Translate (mixedcase)
                   Pagesize  (60)
```

Figure 78. Example audit trail, showing the types of record written for activity-level auditing (Part 1)

```
CBTS Audit Trail Utility - Audit Print                                           Date : 29/01/1999 Time : 15:24:02 Page 000002

Ptype(SALES   ) Function(Define Process  ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000053) Activity(DFHROOT         ) Transid(R   ) Program(R       ) Userid(CICSUSER)
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....F..DFHROOT         )
                                                                    (CCDCCDCF11CCCCDCEC4CECEECFFB3349C00CCCDDDE444444444)
                                                                    (21469351A172924981B98363339A28606014689663000000000)
  Current: Transid(P   ) Program(P       ) Userid(CICSUSER) Date(1999.029) Time(15:23:53.323766)

Ptype(SALES   ) Function(Run Process     ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000053) Activity(DFHROOT         ) Asynchronous
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....F..DFHROOT         )
                                                                    (CCDCCDCF11CCCCDCEC4CECEECFFB3349C00CCCDDDE444444444)
                                                                    (21469351A172924981B98363339A28606014689663000000000)
  Current: Transid(P   ) Program(P       ) Userid(CICSUSER) Date(1999.029) Time(15:23:53.324025)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000054) Activity(DFHROOT         ) Event(DFHINITIAL      )
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....F..DFHROOT         )
                                                                    (CCDCCDCF11CCCCDCEC4CECEECFFB3349C00CCCDDDE444444444)
                                                                    (21469351A172924981B98363339A28606014689663000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(15:23:53.433036)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000055) Activity(B               ) Event(DFHINITIAL      )
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2........B               )
                                                                    (CCDCCDCF11CCCCDCEC4CEDFECEFB3360A00C444444444444444)
                                                                    (21469351A172924981B98229672A283CE012000000000000000)
  Current: Transid(B   ) Program(B       ) Userid(CICSUSER) Date(1999.029) Time(15:23:53.440627)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000055) Activity(B               ) Compstatus(Normal    )
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2........B               )
                                                                    (CCDCCDCF11CCCCDCEC4CEDFECEFB3360A00C444444444444444)
                                                                    (21469351A172924981B98229672A283CE012000000000000000)
  Current: Transid(B   ) Userid(CICSUSER) Date(1999.029) Time(15:23:53.440834)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000056) Activity(C               ) Event(DFHINITIAL      )
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....<-..C               )
                                                                    (CCDCCDCF11CCCCDCEC4CEDFECEFB3364600C444444444444444)
                                                                    (21469351A172924981B98229672A286C0013000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(15:23:53.549149)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000057) Activity(D               ) Event(DFHINITIAL      )
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2.....x..D               )
                                                                    (CCDCCDCF11CCCCDCEC4CEDFECEFB336FA00C444444444444444)
                                                                    (21469351A172924981B98229672A287C0140000000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(15:23:54.116600)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000058) Activity(E               ) Event(DFHINITIAL      )
                                                          ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..."_K..E               )
                                                                    (CCDCCDCF11CCCCDCEC4CEDFECEFB3376D00C444444444444444)
                                                                    (21469351A172924981B98229672A28FD2015000000000000000)
  Current: Transid(E   ) Program(E       ) Userid(CICSUSER) Date(1999.029) Time(15:23:54.185211)
```

Figure 79. Example audit trail, showing the types of record written for activity-level auditing (Part 2)

```
CBTS Audit Trail Utility - Audit Print                                    Date : 29/01/1999 Time : 15:24:02 Page 000003

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000058) Activity(E               ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..."_K..E               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3376D00C444444444444444)
                                                            (21469351A17292498 1B98229672A28FD20150000000000000)
  Current: Transid(E   ) Userid(CICSUSER) Date(1999.029) Time(15:23:54.185619)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000059) Activity(F               ) Event(DFHINITIAL      )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2...f."..F               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3381700C444444444444444)
                                                            (21469351A17292498 1B98229672A2865F016000000000000000)
  Current: Transid(F   ) Program(F       ) Userid(CICSUSER) Date(1999.029) Time(15:23:54.198352)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000059) Activity(F               ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2...f."..F               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3381700C444444444444444)
                                                            (21469351A17292498 1B98229672A2865F016000000000000000)
  Current: Transid(F   ) Userid(CICSUSER) Date(1999.029) Time(15:23:54.198609)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000060) Activity(G               ) Event(DFHINITIAL      )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....0...G               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB330FC00C444444444444444)
                                                            (21469351A17292498 1B98229672A2990D017000000000000000)
  Current: Transid(G   ) Program(G       ) Userid(CICSUSER) Date(1999.029) Time(15:23:58.581394)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000061) Activity(C               ) Event(E               )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....<-..C               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3364600C444444444444444)
                                                            (21469351A17292498 1B98229672A286C0013000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(15:23:58.591807)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000062) Activity(C               ) Event(F               )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....<-..C               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3364600C444444444444444)
                                                            (21469351A17292498 1B98229672A286C0013000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(15:23:58.620666)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000062) Activity(C               ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....<-..C               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3364600C444444444444444)
                                                            (21469351A17292498 1B98229672A286C0013000000000000000)
  Current: Transid(C   ) Userid(CICSUSER) Date(1999.029) Time(15:23:58.636578)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000063) Activity(DFHROOT         ) Event(C               )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....F..DFHROOT         )
                                                            (CCDCCDCF11CCCCDCEC4CECEECFFB3349C00CCCDDDE444444444)
                                                            (21469351A17292498 1B98363339A2860601468966300000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(15:23:58.661620)
```

Figure 80. Example audit trail, showing the types of record written for activity-level auditing (Part 3)

```
CBTS Audit Trail Utility - Audit Print                                    Date : 29/01/1999 Time : 15:24:02 Page 000004

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000064) Activity(G               ) Compstatus(Forced    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....0...G              )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB330FC00C444444444444444)
                                                            (21469351A172924981B98229672A2990D017000000000000000)
  Current: Transid(I   ) Program(I       ) Userid(CICSUSER) Date(1999.029) Time(15:24:00.664584)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000065) Activity(D               ) Event(G              )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2.....x..D              )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB336FA00C444444444444444)
                                                            (21469351A172924981B98229672A287C7014000000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(15:24:00.725741)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000066) Activity(H               ) Event(DFHINITIAL     )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....l...H              )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3359000C444444444444444)
                                                            (21469351A172924981B98229672A2F737018000000000000000)
  Current: Transid(H   ) Program(H       ) Userid(CICSUSER) Date(1999.029) Time(15:24:00.784073)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000066) Activity(H               ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2....l...H              )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB3359000C444444444444444)
                                                            (21469351A172924981B98229672A2F737018000000000000000)
  Current: Transid(H   ) Userid(CICSUSER) Date(1999.029) Time(15:24:00.784346)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000067) Activity(D               ) Event(H              )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2.....x..D              )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB336FA00C444444444444444)
                                                            (21469351A172924981B98229672A287C7014000000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(15:24:00.813682)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000067) Activity(D               ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2.....x..D              )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB336FA00C444444444444444)
                                                            (21469351A172924981B98229672A287C7014000000000000000)
  Current: Transid(D   ) Userid(CICSUSER) Date(1999.029) Time(15:24:02.478498)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000068) Activity(DFHROOT         ) Event(D              )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....F..DFHROOT        )
                                                            (CCDCCDCF11CCCCDCEC4CECEECFFB3349C00CCCDDDE444444444)
                                                            (21469351A172924981B98363339A2860601468966300000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(15:24:02.511054)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000068) Activity(DFHROOT         ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....F..DFHROOT        )
                                                            (CCDCCDCF11CCCCDCEC4CECEECFFB3349C00CCCDDDE444444444)
                                                            (21469351A172924981B98363339A2860601468966300000000)
  Current: Transid(R   ) Userid(CICSUSER) Date(1999.029) Time(15:24:02.571838)

CBTS Audit Trail Utility - Selection Results                              Date : 29/01/1999 Time : 15:24:02 Page 000005


Number of Audit records read :        24
Number of records selected   :        24
  Processing Complete
```

Figure 81. Example audit trail, showing the types of record written for activity-level auditing (Part 4)

Example audit trail full auditing

Follow this example audit trail to see the output that is produced if the AUDITLEVEL attribute of the PROCESSTYPE definition for the SALES process-type is set to 'FULL'.

Figure 82 on page 184 shows the output that is produced if:

- the AUDITLEVEL attribute of the PROCESSTYPE definition for the SALES process-type is set to 'FULL'
- The control statements in the SYSIN data set specify that all records for the SALES1234567890 process must be formatted.

```
CBTS Audit Trail Utility - Parameter Validation                                  Date : 29/01/1999 Time : 14:39:04 Page 000001

Exec Parm Options: Natlang   (EN)
                   Translate (mixedcase)
                   Pagesize  (60)
```

Figure 82. Example audit trail, showing the types of record written for full auditing (Part 1)

```
CBTS Audit Trail Utility - Audit Print                                          Date : 29/01/1999 Time : 14:39:04 Page 000002

Ptype(SALES   ) Function(Define Process  ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000033) Activity(DFHROOT        ) Transid(R   ) Program(R       ) Userid(CICSUSER)
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....v..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB2902A00CCCDDDE444444444)
                                                                  (21469351A17292498 1B98363339A709F5014689663000000000)
  Current: Transid(P   ) Program(P       ) Userid(CICSUSER) Date(1999.029) Time(14:36:12.557162)

Ptype(SALES   ) Function(Run Process     ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000033) Activity(DFHROOT        ) Asynchronous
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....v..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB2902A00CCCDDDE444444444)
                                                                  (21469351A17292498 1B98363339A709F5014689663000000000)
  Current: Transid(P   ) Program(P       ) Userid(CICSUSER) Date(1999.029) Time(14:36:13.921790)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000034) Activity(DFHROOT        ) Event(DFHINITIAL      )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....v..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB2902A00CCCDDDE444444444)
                                                                  (21469351A17292498 1B98363339A709F5014689663000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.142640)

Ptype(SALES   ) Function(Define Activity ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000034) Activity(B              ) CompletionEvent(B               ) Transid(B   ) Program(B       ) Userid(CICSUSER)
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.....B               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29B0300C444444444444444)
                                                                  (21469351A17292498 1B98229672A7111C012000000000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.295419) Activity(DFHROOT         )

Ptype(SALES   ) Function(Run Activity    ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000034) Activity(B              ) Synchronous
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.....B               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29B0300C444444444444444)
                                                                  (21469351A17292498 1B98229672A7111C012000000000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.295549) Activity(DFHROOT         )

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000035) Activity(B              ) Event(DFHINITIAL      )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.....B               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29B0300C444444444444444)
                                                                  (21469351A17292498 1B98229672A7111C012000000000000000)
  Current: Transid(B   ) Program(B       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.296323)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000035) Activity(B              ) Compstatus(Normal    )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.....B               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29B0300C444444444444444)
                                                                  (21469351A17292498 1B98229672A7111C012000000000000000)
  Current: Transid(B   ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.408739)

Ptype(SALES   ) Function(Define Activity ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000034) Activity(C              ) CompletionEvent(C               ) Transid(C   ) Program(C       ) Userid(CICSUSER)
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.6...C               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29DF500C444444444444444)
                                                                  (21469351A17292498 1B98229672A71C690130000000000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.472960) Activity(DFHROOT         )
```

Figure 83. Example audit trail, showing the types of record written for full auditing (Part 2)

```
CBTS Audit Trail Utility - Audit Print                                    Date : 29/01/1999 Time : 14:39:04 Page 000003

Ptype(SALES   ) Function(Run Activity    ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000034) Activity(C              ) Asynchronous
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.6...C               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB29DF500C444444444444444)
                                                              (21469351A172924981B98229672A71C690130000000000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.473066) Activity(DFHROOT         )

Ptype(SALES   ) Function(Define Activity ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000034) Activity(D              ) CompletionEvent(D              ) Transid(D   ) Program(D       ) Userid(CICSUSER)
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j..9..D               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB29DBF00C444444444444444)
                                                              (21469351A172924981B98229672A71EE901400000000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.482228) Activity(DFHROOT         )

Ptype(SALES   ) Function(Run Activity    ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000034) Activity(D              ) Asynchronous
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j..9..D               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB29DBF00C444444444444444)
                                                              (21469351A172924981B98229672A71EE901400000000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.482346) Activity(DFHROOT         )

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000036) Activity(C              ) Event(DFHINITIAL      )
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.6...C               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB29DF500C444444444444444)
                                                              (21469351A172924981B98229672A71C690130000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.556761)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000037) Activity(D              ) Event(DFHINITIAL      )
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j..9..D               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB29DBF00C444444444444444)
                                                              (21469351A172924981B98229672A71EE901400000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.569775)

Ptype(SALES   ) Function(Define Activity ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000036) Activity(E              ) CompletionEvent(E              ) Transid(E   ) Program(E       ) Userid(CICSUSER)
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..o..E               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB2906900C444444444444444)
                                                              (21469351A172924981B98229672A72946015000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.656929) Activity(C               )

Ptype(SALES   ) Function(Run Activity    ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000036) Activity(E              ) Asynchronous
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..o..E               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB2906900C444444444444444)
                                                              (21469351A172924981B98229672A72946015000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.657049) Activity(C               )

Ptype(SALES   ) Function(Define Activity ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000036) Activity(F              ) CompletionEvent(F              ) Transid(F   ) Program(F       ) Userid(CICSUSER)
                                                    ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..I..F               )
                                                              (CCDCCDCF11CCCCDCEC4CEDFECEFB290AC00C444444444444444)
                                                              (21469351A172924981B98229672A72BA901600000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.668485) Activity(C               )
```

Figure 84. Example audit trail, showing the types of record written for full auditing (Part 3)

```
CBTS Audit Trail Utility - Audit Print                                    Date : 29/01/1999 Time : 14:39:04 Page 000004

Ptype(SALES   ) Function(Run Activity    ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000036) Activity(F              ) Asynchronous
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..I..F               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB290AC00C444444444444444)
                                                           (21469351A17292498 1B98229672A72BA90160000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.668584) Activity(C              )

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000038) Activity(E              ) Event(DFHINITIAL      )
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..o..E               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB2906900C444444444444444)
                                                           (21469351A17292498 1B98229672A729460150000000000000000)
  Current: Transid(E   ) Program(E       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.757748)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000039) Activity(F              ) Event(DFHINITIAL      )
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..I..F               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB290AC00C444444444444444)
                                                           (21469351A17292498 1B98229672A72BA90160000000000000000)
  Current: Transid(F   ) Program(F       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.790932)

Ptype(SALES   ) Function(Define Activity ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000037) Activity(G              ) CompletionEvent(G              ) Transid(G   ) Program(G       ) Userid(CICSUSER)
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k.....G               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB2925000C444444444444444)
                                                           (21469351A17292498 1B98229672A72F75017000000000000000)
  Current: Transid(D   ) Program(D       ) User
Ptype(SALES   ) Function(Run Activity    ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000037) Activity(G              ) Asynchronous
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k.....G               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB2925000C444444444444444)
                                                           (21469351A17292498 1B98229672A72F75017000000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.811377) Activity(D              )

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000040) Activity(G              ) Event(DFHINITIAL      )
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k.....G               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB2925000C444444444444444)
                                                           (21469351A17292498 1B98229672A72F75017000000000000000)
  Current: Transid(G   ) Program(G       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.844281)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000039) Activity(F              ) Compstatus(Normal    )
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..I..F               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB290AC00C444444444444444)
                                                           (21469351A17292498 1B98229672A72BA90160000000000000000)
  Current: Transid(F   ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.887329)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000041) Activity(C              ) Event(F               )
                                                 ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.6...C               )
                                                           (CCDCCDCF11CCCCDCEC4CEDFECEFB29DF500C444444444444444)
                                                           (21469351A17292498 1B98229672A71C690130000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.979781)
```

Figure 85. Example audit trail, showing the types of record written for full auditing (Part 4)

```
CBTS Audit Trail Utility - Audit Print                                    Date : 29/01/1999 Time : 14:39:04 Page 000005

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000038) Activity(E              ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..o..E               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB2906900C444444444444444)
                                                            (21469351A17292498 1B98229672A72946015000000000000000)
  Current: Transid(E   ) Userid(CICSUSER) Date(1999.029) Time(14:36:15.070372)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000042) Activity(C              ) Event(E              )
                                                  ActivityId(BAMFILE1..GB
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB29DF500C444444444444444)
                                                            (21469351A17292498 1B98229672A71C690130000000000000000)
  Current: Transid(C   ) Program(C       ) Userid(CICSUSER) Date(1999.029) Time(14:36:15.117121)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000042) Activity(C              ) Compstatus(Normal    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.6...C               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB29DF500C444444444444444)
                                                            (21469351A17292498 1B98229672A71C690130000000000000000)
  Current: Transid(C   ) Userid(CICSUSER) Date(1999.029) Time(14:36:15.135971)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000043) Activity(DFHROOT        ) Event(C              )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....v..DFHROOT         )
                                                            (CCDCCDCF11CCCCDCEC4CECEECFFB2902A00CCCDDDE444444444)
                                                            (21469351A17292498 1B98363339A709F501468966300000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:15.169265)

Ptype(SALES   ) Function(Acquire ActId   ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000045) Activity(G              )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k.....G               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB2925000C444444444444444)
                                                            (21469351A17292498 1B98229672A72F750170000000000000000)
  Current: Transid(I   ) Program(I       ) Userid(CICSUSER) Date(1999.029) Time(14:36:21.922942)

Ptype(SALES   ) Function(Cancel Activity ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000045) Activity(G              )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k.....G               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB2925000C444444444444444)
                                                            (21469351A17292498 1B98229672A72F750170000000000000000)
  Current: Transid(I   ) Program(I       ) Userid(CICSUSER) Date(1999.029) Time(14:36:21.923045)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                    ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000045) Activity(G              ) Compstatus(Forced    )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k.....G               )
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB2925000C444444444444444)
                                                            (21469351A17292498 1B98229672A72F750170000000000000000)
  Current: Transid(I   ) Program(I       ) Userid(CICSUSER) Date(1999.029) Time(14:36:21.923093)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                    ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000046) Activity(D              ) Event(G              )
                                                  ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j..9..D
                                                            (CCDCCDCF11CCCCDCEC4CEDFECEFB29DBF00C444444444444444)
                                                            (21469351A17292498 1B98229672A71EE901400000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(14:36:21.948512)
```

Figure 86. Example audit trail, showing the types of record written for full auditing (Part 5)

```
CBTS Audit Trail Utility - Audit Print                                    Date : 29/01/1999 Time : 14:39:04 Page 000006

Ptype(SALES   ) Function(Define Activity ) Process(SALES1234567890                   ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000046) Activity(H              ) CompletionEvent(H              ) Transid(H    ) Program(H       ) Userid(CICSUSER)
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..q.X...H               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29FED00C444444444444444)
                                                                  (21469351A17292498 1B98229672A78F7F018000000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(14:36:21.990993) Activity(D              )

Ptype(SALES   ) Function(Run Activity    ) Process(SALES1234567890                   ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000046) Activity(H              ) Asynchronous
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..q.X...H               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29FED00C444444444444444)
                                                                  (21469351A17292498 1B98229672A78F7F018000000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(14:36:21.991119) Activity(D              )

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                   ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000047) Activity(H              ) Event(DFHINITIAL      )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..q.X...H               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29FED00C444444444444444)
                                                                  (21469351A17292498 1B98229672A78F7F018000000000000000)
  Current: Transid(H   ) Program(H       ) Userid(CICSUSER) Date(1999.029) Time(14:36:22.052659)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                   ) System(SYS2) Auditlog(BAMAUDIT)
  Taskno(0000047) Activity(H              ) Compstatus(Normal    )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..q.X...H               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29FED00C444444444444444)
                                                                  (21469351A17292498 1B98229672A78F7F018000000000000000)
  Current: Transid(H   ) Userid(CICSUSER) Date(1999.029) Time(14:36:22.123737)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                   ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000048) Activity(D              ) Event(H              )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j..9..D               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29DBF00C444444444444444)
                                                                  (21469351A17292498 1B98229672A71EE901400000000000000)
  Current: Transid(D   ) Program(D       ) Userid(CICSUSER) Date(1999.029) Time(14:36:22.147332)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                   ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000048) Activity(D              ) Compstatus(Normal    )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j..9..D               )
                                                                  (CCDCCDCF11CCCCDCEC4CEDFECEFB29DBF00C444444444444444)
                                                                  (21469351A17292498 1B98229672A71EE901400000000000000)
  Current: Transid(D   ) Userid(CICSUSER) Date(1999.029) Time(14:36:22.162148)

Ptype(SALES   ) Function(Activation      ) Process(SALES1234567890                   ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000049) Activity(DFHROOT        ) Event(D              )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....v..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB2902A00CCCDDDE444444444)
                                                                  (21469351A17292498 1B98363339A709F5014689663000000000)
  Current: Transid(R   ) Program(R       ) Userid(CICSUSER) Date(1999.029) Time(14:36:22.185932)

Ptype(SALES   ) Function(Completion      ) Process(SALES1234567890                   ) System(SYS1) Auditlog(BAMAUDIT)
  Taskno(0000049) Activity(DFHROOT        ) Compstatus(Normal    )
                                                        ActivityId(BAMFILE1..GBIBMIYA.IYCWTC39.....v..DFHROOT         )
                                                                  (CCDCCDCF11CCCCDCEC4CECEECFFB2902A00CCCDDDE444444444)
                                                                  (21469351A17292498 1B98363339A709F5014689663000000000)
  Current: Transid(R   ) Userid(CICSUSER) Date(1999.029) Time(14:36:22.482472)




CBTS Audit Trail Utility - Selection Results                              Date : 29/01/1999 Time : 14:39:04 Page 000007

Number of Audit records read :        40
Number of records selected   :        40
  Processing Complete
```

Figure 87. Example audit trail, showing the types of record written for full auditing (Part 6)

Note:

1. All times in the audit trails refer to Greenwich Mean Time (GMT).
2. As the example audit trails show, the detailed information within the audit report varies according to the audit point taken.
3. When an activity is activated, in some cases the name of the event that caused the activation is not available. In these cases, the request type and reason for the activation are reported. The possible request types are:
   - Dispatch
   - Cancel
   - Delete

The possible reasons are:

- Fire complete
- Fire input
- Fire timer
- Delete command
- Delete complete
- Delete reset
- Delete tree
- Cancel command
- Cancel complete
- Cancel force
- Reattach acq
- Unknown. Unknown applies only to dispatch requests. The unknown reason means that the activation has not been triggered by a specific event. This can happen, for example, in any of the following cases:
  - An application issues a RESUME command against a child activity. In this case, BTS does a speculative dispatch, to see if there are any events to be serviced; it does not know, at the time the activation is started, whether there are any.
  - An activation terminates but there are several more fired events that it needs to service. BTS reactivates the activity immediately, but does not regard the activation as being caused by any particular event.
  - A timer is forced. Although a particular timer event fires, this firing occurs in the application that issued the FORCE TIMER command; it is not part of the request that starts the activation.

Chapter 17. Examining BTS repository records

You can examine records on a BTS repository data set with the DFHBARUP utility, which takes a "snapshot" of your BTS system.

About this task

It contains:

- "The repository utility program, DFHBARUP"
- "Using DFHBARUP."

The repository utility program, DFHBARUP

You can use the DFHBARUP utility to take a "snapshot" of your BTS system at the time the utility is run. You can filter to print records for a specific process or activity.

By default, DFHBARUP prints all the records currently on the specified repository. If you have more than one repository, it is a snapshot of the processes served by the specified repository.

The state of a repository can change from moment to moment, especially if it is shared across a busy sysplex. For example, records for new processes and activities can be added constantly; conversely, as processes complete and events are deleted their associated records disappear from the repository.

Using DFHBARUP you can filter selected records. For example, you could print only the records associated with a specific process. Doing so would give you the current state of:

- The activities that have been defined to the process, and have not yet been deleted
- The containers associated with the activities - that is, the data they contain
- The events in the event pools of the activities.

Alternatively, you could print only the records associated with a specific activity. Doing so would give you the current state of:

- The activity itself
- The containers associated with the activity
- The events in the event pool of the activity.

DFHBARUP formats the records it extracts, to make them easier to interpret.

Using DFHBARUP

Use the repository utility program, DFHBARUP, to read, format, and print selected records from a specified repository data set.

About this task

Run DFHBARUP as a batch job.

© Copyright IBM Corp. 1999, 2011

DFHBARUP reads the records in the order they are stored on the repository - that is, in keyed-sequence order. To select the records that DFHBARUP writes to the output data set, SYSPRINT, you include control statements in the SYSIN data set. By default, DFHBARUP prints all records currently on the data set. DFHBARUP formats the selected records before writing them to SYSPRINT.

Sample job stream to run the DFHBARUP program

To help you develop your own job to run the DFHBARUP utility program, use this sample job.

Figure 88 shows an example job stream to run the DFHBARUP program. The job stream includes DD statements for the following data sets:

The repository
The repository data set to be examined to produce the output data. (Figure 88 shows a DD name of 'REPOS'.)

STEPLIB
A partitioned data set (DSORG=PO) that contains the DFHBARUP program module. If the module is in a library in the link list, this statement is not required.

SYSIN
The input control data set. This file must be in 80- byte record format. The control statements that you can use in this data set are described in "SYSIN control statements" on page 193.

Control statements can be continued on to the next line by including any non-blank character in column 72. If the line that follows a continuation character is empty or contains control arguments which conflict with those that make up the preceding part of the control statement, an error is reported and execution of the utility ends. Any characters which occur beyond column 72 are ignored.

SYSPRINT
The output data set to which the formatted audit records and control messages are to be sent.

```
//*******************************************************************
//* RUN DFHBARUP (REPOSITORY UTILITY PROGRAM)
//*
//*
//*******************************************************************
//ARUP     EXEC PGM=DFHBARUP,PARM='N(EN),P(60),T(M)'
//STEPLIB    DD DSN=CTS130.CICS530.SDFHLOAD,DISP=SHR
//*******************************************************************
//*   The output will go to SYSPRINT
//*******************************************************************
//SYSPRINT DD SYSOUT=A,DCB=RECFM=FBA
//REPOS DD DISP=SHR,DSN=CICS530.CBTS.SALESREP
//SYSIN    DD *
PTYPE(SALES)                                                          +
PROCESS(CUSTSALES1999.13872977829728.QA)                              +
ACTIVITY(ORDER)
/*
//*
```

Figure 88. Sample job to run the DFHBARUP utility program. This job prints all the records for the ORDER activity of the CUSTSALES1999.13872977829728.QA process.

EXEC parameters

You can use the PARM keyword on the EXEC statement to pass the NATLANG, PAGESIZE, and TRANSLATE parameters to the DFHBARUP utility. Using these parameters you can control the content and format of the DFHBARUP utility output.

The form of the EXEC statement is:

```
EXEC PGM=DFHBARUP,PARM='parm1,...,parmn'
```

NATLANG({EN|*CS∨KA***})**
The language in which messages are to be issued.

The minimum abbreviation of this parameter is **N**. The possible values are:

CS Traditional Chinese

EN English. This is the default.

KA Kanji.

PAGESIZE({60|*nn***})**
The number of lines to be printed per page, when the output from the utility is sent to a printer. Valid values are in the range 20–99. The default is 60.

The minimum abbreviation of this parameter is **P**.

TRANSLATE({MIXEDCASE|*UPPERCASE***})**
Whether the output from the utility is to be in mixed-case or uppercase. The default is mixed-case.

The minimum abbreviation of this parameter is **T**. The minimum abbreviations of MIXEDCASE and UPPERCASE are M and U respectively.

SYSIN control statements

The SYSIN data set passes information to DFHBARUP. You can include statements to select specific sets of records to be formatted. Define the SYSIN data set with these control statements.

Format of the SYSIN control statements

```
SYSIN DD *
[REPOSITORY(name)]
[PTYPE(name)]
[PROCESS(name)]
[ACTIVITY(name)]
```

Comments are identified by an asterisk in the first position.

The REPOSITORY statement cannot contain additional arguments. Other statements can consist of multiple arguments. When using multiple arguments, put each argument on a separate line; use a non-blank character in column 72 to indicate that this argument and the following one are to be treated as a single control statement. An illegal combination of arguments generates an error message and the utility is not run against the log stream.

ACTIVITY(*name***)**
The 1-16 character name of an activity. Only records for activities of this name are formatted. To limit the scope of the search, specify a PROCESS or PTYPE argument with ACTIVITY.

PROCESS(*name*)
The 1-36 character name of a BTS process. No further arguments are needed to make up a control statement; if none are provided, all records containing this process name are selected. To limit the scope of the search, you can add a PTYPE argument on an adjoining line, using a continuation character in column 72.

PTYPE(*name*)
The 1-8 character name of a BTS process-type. No additional arguments are needed; if none are provided, all records containing this process-type are selected. To limit the scope of the search, you can add a PROCESS argument on an adjoining line, using a continuation character in column 72.

REPOSITORY(*name*)
The 1-8 character DD name that identifies the repository data set to be searched. The default is 'REPOS'. This argument must not be specified more than once. It cannot be used with any other in a control statement.

If the specified repository file cannot be opened, or if more than one REPOSITORY statement is found in the SYSIN data set, an error occurs and DFHBARUP terminates.

Example output from the DFHBARUP utility

Follow this example of output produced by the DFHBARUP utility. The example control statements format all the records currently on the SALEREP repository for the SALES1234567890 process, which is of the SALES process-type.

```
.
.
//SALESREP DD DISP=SHR,DSN=CICS530.CBTS.SALESREP
//SYSIN      DD *
REPOSITORY(SALESREP)
PTYPE(SALES)                                                    +
PROCESS(SALES1234567890)
/*
```

Figure 89. Example control statements, to format all records on the SALEREP repository for the SALES1234567890 process

Figure 90 shows the output that might be produced by the control statements in Figure 89.

```
CICS Business Transaction Services - Parameter Validation                    Date : 29/01/1999 Time : 14:39:04 Page 0001
Exec Parm Options: Natlang   (EN)
                   Translate (mixedcase)
                   Pagesize  (60)

REPOSITORY(SALEREP)
```

Figure 90. Example output from the DFHBARUP utility (Part 1)

```
CICS Business Transaction Services - Repository File Report                          Date : 29/01/1999 Time : 14:39:04 Page 0002
Activity Name : DFHROOT          Id : ..GBIBMIYA.IYCWTC37........DFHROOT           Generation : 0000001
                                      11CCCCDCEC4CECEECFF44042F00CCCDDDE4444444444
                                      A172924981B98363337BB0C1B0146896630000000000
Definitional Attributes
  Program    : ABU081D
  Transid    : RUP4
  Userid     : CICSUSER
  Comp Event :
Current State
  Mode       : Dormant    (Initial, Active, Dormant, Cancelling, Complete)
  Suspended  : No  (Yes, No)
  Generation : 0000001
  Child Count : 0000002
Completion Status
  Completion Response : Incomplete

000000   C1401A11 C7C2C9C2 D4C9E8C1 4BC9E8C3 E6E3C3F3 F74B4B00 4C21FB00 01C4C6C8  *A ..GBIBMIYA.IYCWTC37...<....DFH*
000020   D9D6D6E3 40404040 40404040 40400000 00000004 00004000 000005E0 01500000  *ROOT          ........ ....\.&..*
000040   6EC4C6C8 C2C1C1C3 E3C9E5C9 00000000 FFFFFFFF 01500001 00000000 D740D7E3  *>DFHBAACTIVI.........&......P SA*
000060   E8D7C5F1 4040D7D9 D6C36DC6 D6E4D940 40404040 40404040 40404040 40404040  *LES   SALES1234567890           *
000080   40404040 40404040 40400000 00000000 00000000 00000000 00000000 00000000  *          ......................*
0000A0   00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000  *................................*
0000C0   1A11C7C2 C9C2D4C9 E8C14BC9 E8C3E6E3 C3F3F74B 4B004C21 FB000103 00000000  *..GBIBMIYA.IYCWTC37...<.........*
0000E0   00000000 00000000 00000000 00000000 00000002 00000000 00000000 000003C4  *............................D*
000100   00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000  *................................*
000120   00000000 10C2011C 10C2011C 00000000 00000000 10C2012C 10C2012C C1C2E4F0  *.....B...B..........B...B..ABU0*
000140   F8F1C440 00000000 00000000 D9E4D7F4 C3C9C3E2 E4E2C5D9 40404040 40404040  *81D ........RUP4CICSUSER        *
000160   40404040 40404040 01404040 40404040 40404040 4003C2C1 D4C1E4C4 C9E30000  *        .            .BAMAUDIT..*
000180   00000000 0000FFFF FFFFFFFF                                               *............                    *
Related BTS Objects
  Process Type : SALES   Name : SALES1234567890
  No Parent
  Child Name : ACT_3            Id : ..GBIBMIYA.IYK2ZFX2........ACT_3             Generation : 0000001
                                     11CCCCDCEC4CEDFECEF440F3A00CCE6F444444444444
                                     A172924981B98229672BBCF9F01133D3000000000000
  Child Name : ACT_ONE          Id : ..GBIBMIYA.IYK2ZFX2........ACT_ONE           Generation : 0000001
                                     11CCCCDCEC4CEDFECEF44068A00CCE6DDC4444444444
                                     A17292498 1B98229672BBB35A01133D6550000000000
Eventpool
  Event : (Reattach)
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
```

Figure 91. Example output from the DFHBARUP utility (Part 2)

```
CICS Business Transaction Services - Repository File Report                          Date : 29/01/1999 Time : 14:39:04 Page 0003
  Event : DFHINITIAL
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT_ONE
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT_3
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No

Containers

  No Containers
```

Figure 92. Example output from the DFHBARUP utility (Part 3)

```
CICS Business Transaction Services - Repository File Report                        Date : 29/01/1999 Time : 14:39:04 Page 0004
Activity Name : ACT_ONE          Id : ..GBIBMIYA.IYK2ZFX2........ACT_ONE          Generation : 0000001
                                      11CCCCDCEC4CEDFECEF44068A00CCE6DDC4444444444
                                      A172924981B98229672BBB35A01133D6550000000000
Definitional Attributes
  Program    : ABU081E
  Transid    : RUP5
  Userid     : CICSUSER
  Comp Event : ACT_ONE
Current State
  Mode       : Dormant    (Initial, Active, Dormant, Cancelling, Complete)
  Suspended  : No  (Yes, No)
  Generation : 0000001
  Child Count : 0000000
Completion Status
  Completion Response : Incomplete

000000   C1401A11 C7C2C9C2 D4C9E8C1 4BC9E8D2 F2E9C6E7 F24B4B0B 6385AA00 01C1C3E3  *A ..GBIBMIYA.IYK2ZFX2....e...ACT*
000020   6DD6D5C5 40404040 40404040 40400000 00008081 00004000 00003E88 01500000  *_ONE          .....a.. ....h.&..*
000040   6EC4C6C8 C2C1C1C3 E3C9E5C9 00000000 FFFFFFFF 01500001 00000000 D740D7E3  *>DFHBAACTIVI.........&......P SA*
000060   E8D7C5F1 4040D7D9 D6C36DC6 D6E4D940 40404040 40404040 40404040 40404040  *LES   SALES1234567890           *
000080   40404040 40404040 40400000 0000C140 1A11C7C2 C9C2D4C9 E8C14BC9 E8C3E6E3  *          ....A ..GBIBMIYA.IYCWT*
0000A0   C3F3F74B 4B004C21 FB0001C4 C6C8D9D6 D6E34040 40404040 40404040 00000000  *C37...<....DFHROOT          ....*
0000C0   1A11C7C2 C9C2D4C9 E8C14BC9 E8C3E6E3 C3F3F74B 4B004C21 FB000103 D9E4D7F4  *..GBIBMIYA.IYCWTC37...<.....RUP4*
0000E0   C3C9C3E2 E4E2C5D9 00000000 00000001 00000000 00000000 00000000 000003C4  *CICSUSER.......................D*
000100   00000001 00000003 000084A9 00000000 00000000 00000000 00000000 00000000  *.........dz....................*
000120   00000000 108F911C 108F911C 00000000 00000000 10C20C10 101465E0 C1C2E4F0  *......j...j.........B.....\ABU0*
000140   F8F1C540 00000000 00000000 D9E4D7F5 C3C9C3E2 E4E2C5D9 C1C3E36D D6D5C540  *81E ........RUP5CICSUSERACT_ONE *
000160   40404040 40404040 01404040 40404040 40404040 4003C2C1 D4C1E4C4 C9E30000  *        .            .BAMAUDIT..*
000180   00000000 0000FFFF FFFFFFFF                                               *............                    *
Related BTS Objects
  Proce  Parent Name : DFHROOT          Id : ..GBIBMIYA.IYCWTC37........DFHROOT          Generation : 0000001
                                      11CCCCDCEC4CECEECFF44042F00CCCDDDE4444444444
                                      A172924981B98363337BB0C1B0146896630000000000
No Children
Eventpool
  Event : (Reattach)
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
```

Figure 93. Example output from the DFHBARUP utility (Part 4)

```
CICS Business Transaction Services - Repository File Report                    Date : 29/01/1999 Time : 14:39:04 Page 0005
  Event : DFHINITIAL
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT1_CONTINUE
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT1_END
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT2_DEF
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT2_CAN
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT2_SUS
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : ACT2_RES
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No

Containers

Container Name : ACT_CONT_1      Container Length : x'00008000'

000000   FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF  *................................*
   .
   lines omitted
   .
07FE0   FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF  *................................*
Container Name : ACT_CONT_2      Container Length : x'00000400'

000000   22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222  *................................*
   .
   lines omitted
   .
0003E0   22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222  *................................*
Container Name : ACT_CONT_3      Container Length : x'00000019'

000000   10101010 10101010 10101010 10101010 10101010 10101010 10                 *.........................      *
```

Figure 94. Example output from the DFHBARUP utility (Part 5)

```
CICS Business Transaction Services - Repository File Report                         Date : 29/01/1999 Time : 14:39:04 Page 0006
Activity Name : ACT_3            Id : ..GBIBMIYA.IYK2ZFX2........ACT_3            Generation : 0000001
                                      11CCCCDCEC4CEDFECEF440F3A00CCE6F444444444444
                                      A172924981B98229672BBCF9F01133D3000000000000
Definitional Attributes
  Program    : ABU081Z
  Transid    : RUPZ
  Userid     : CICSUSER
  Comp Event : ACT_3
Current State
  Mode        : Dormant    (Initial, Active, Dormant, Cancelling, Complete)
  Suspended   : No  (Yes, No)
  Generation  : 0000001
  Child Count : 0000000
Completion Status
  Completion Response : Incomplete

000000   C1401A11 C7C2C9C2 D4C9E8C1 4BC9E8D2 F2E9C6E7 F24B4B0C FF39AF00 01C1C3E3  *A ..GBIBMIYA.IYK2ZFX2........ACT*
000020   6DF34040 40404040 40404040 40400000 00000000 00004000 00000550 01500000  *_3            ........ ....&.&..*
000040   6EC4C6C8 C2C1C1C3 E3C9E5C9 00000000 D9D4E4E6 01500001 00000000 D740D7E3  *>DFHBAACTIVI....RMUW.&......P SA*
000060   E8D7C5F1 4040D7D9 D6C36DC6 D6E4D940 40404040 40404040 40404040 40404040  *LES   SALES1234567890           *
000080   40404040 40404040 40400000 0000C140 1A11C7C2 C9C2D4C9 E8C14BC9 E8C3E6E3  *          ....A ..GBIBMIYA.IYCWT*
0000A0   C3F3F74B 4B004C21 FB0001C4 C6C8D9D6 D6E34040 40404040 40404040 00000000  *C37...<....DFHROOT          ....*
0000C0   1A11C7C2 C9C2D4C9 E8C14BC9 E8C3E6E3 C3F3F74B 4B004C21 FB000103 D9E4D7F4  *..GBIBMIYA.IYCWTC37...<.....RUP4*
0000E0   C3C9C3E2 E4E2C5D9 00000000 00000001 00000000 00000000 00000000 000003C4  *CICSUSER.......................D*
000100   00000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000  *................................*
000120   00000000 10C2911C 10C2911C 00000000 00000000 10C2912C 10C2912C C1C2E4F0  *.....Bj..Bj..........Bj..Bj.ABU0*
000140   F8F1E940 00000000 00000000 D9E4D7E9 C3C9C3E2 E4E2C5D9 C1C3E36D F3404040  *81Z ........RUPZCICSUSERACT_3   *
000160   40404040 40404040 01404040 40404040 40404040 4003C2C1 D4C1E4C4 C9E30000  *        .            .BAMAUDIT..*
000180   00000000 00000000 00000000                                              *............                    *
Related BTS Objects
  Process Type : SALES   Name : SALES1234567890
  Parent Name : DFHROOT          Id : ..GBIBMIYA.IYCWTC37........DFHROOT          Generation : 0000001
                                      11CCCCDCEC4CECEECFF44042F00CCCDDDE4444444444
                                      A172924981B98363337BB0C1B0146896630000000000
No Children
Eventpool
  Event : (Reattach)
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
```

Figure 95. Example output from the DFHBARUP utility (Part 6)

```
CICS Business Transaction Services - Repository File Report                          Date : 29/01/1999 Time : 14:39:04 Page 0007
  Event : DFHINITIAL
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
  Event : T1
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
    Timer     : TIMER_ONE
  Event : T2
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
    Timer     : TIMER_TWO
  Event : T3
    Type      : Activity
    Fired     : No
    Reattach  : Yes
    Retrieve  : No
    Subevent  : No
    Timer     : TIMER_3
  Timer : TIMER_ONE
    Status    : Unexpired
    Date      : 05/11/1998 Time : 10:23:46
    Event     : T1
  Timer : TIMER_TWO
    Status    : Unexpired
    Date      : 08/11/1998 Time : 10:23:49
    Event     : T2
  Timer : TIMER_3
    Status    : Unexpired
    Date      : 27/11/1998 Time : 10:23:52
    Event     : T3

Containers

  No Containers
```

Figure 96. Example output from the DFHBARUP utility (Part 7)

```
CICS Business Transaction Services - Repository File Report                          Date : 29/01/1999 Time : 14:39:04 Page 0008
Process : SALES1234567890                          Process Type : SALES
Root Id : ..GBIBMIYA.IYCWTC37........DFHROOT
          11CCCCDCEC4CECEECFF44042F00CCCDDDE4444444444
          A172924981B98363337BB0C1B01468966300000000000
Audit Level : Full (Off, Pro, Act, Full)
Audit Log   : BAMAUDIT
000000   D740D7E3 E8D7C5F1 4040D7D9 D6C36DC6 D6E4D940 40404040 40404040 40404040  *P SALES   SALES1234567890       *
000020   40404040 40404040 40404040 40400000 00008004 00004000 00003E14 00A00000  *              ........ .........*
000040   6EC4C6C8 C2C1D7D9 D6C3C5E2 00000000 00000000 00000000 00000000 C1401A11  *>DFHBAPROCES.................A ..*
000060   C7C2C9C2 D4C9E8C1 4BC9E8C3 E6E3C3F3 F74B4B00 4C21FB00 01C4C6C8 D9D6D6E3  *GBIBMIYA.IYCWTC37...<....DFHROOT*
000080   40404040 40404040 40400000 0000776F 10C2003C 00000003 000084A9 00000000  *           .....?.B........dz....*
0000A0   00000000 00000000 00000000 00000000 00000000 108F90AC 108F90AC 00000000  *.............................*
0000C0   00000000 10146C90 10146710 03C2C1D4 C1E4C4C9 E3000001 00000000          Containers
Container Name : Container_one   Container Length : x'00008000'

000000   FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF  *.............................*
       .
       .
       .
        lines omitted
       .
       .
       .

007FE0   FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF  *.............................*
```

Figure 97. Example output from the DFHBARUP utility (Part 8)

```
CICS Business Transaction Services - Repository File Report                          Date : 29/01/1999 Time : 14:39:04 Page 0009
Container Name : Container_two   Container Length : x'00000400'

000000   22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222  *................................*
       .
       lines omitted
       .
0003E0   22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222  *................................*
Container Name : Container_three Container Length : x'00000019'

000000   10101010 10101010 10101010 10101010 10101010 10101010 10                *.........................      *
```

Figure 98. Example output from the DFHBARUP utility (Part 9)

Note: A DFHBARUP report shows activity identifiers in the form they are stored on the repository. Unlike the activity identifiers returned by commands such as ASSIGN and GETNEXT ACTIVITY, those shown by DFHBARUP are not prefixed with the CICS file name of the repository.

Chapter 18. BTS messages

You can use BTS messages to help you understand and respond to BTS activities and issues.

BTS messages are identified by the following prefixes:

- DFHBA - Business application manager messages.
- DFHEM - Event manager messages.
- DFHSH - Scheduler services messages.

All CICS messages, including BTS messages, are listed in *CICS Messages and Codes*. To discover the meaning of a particular message, refer to that book.

© Copyright IBM Corp. 1999, 2011

Chapter 19. Setting trace levels for BTS

Using system initialization parameters or the CETR transaction, you can use the CICS component codes to specify the level of standard and special tracing that you require for your BTS processes.

About this task

BTS consists of three CICS domains:

Domain name	CICS Component code
Business application manager	BA
Event manager	EM
Scheduler services	SH

BTS trace points are listed in the CICS *Trace Entries* manual.

For detailed information about using component codes to set the level of tracing to be applied to particular CICS components, see the *CICS Problem Determination Guide*.

Defining tracing levels at system initialization

You can use these system initialization parameters to define, at system startup, the level of tracing that you require for BTS.

You can code any of the following parameters to define, at CICS system initialization time, the level of tracing required for BTS:

- *SPCTR*, to indicate the level of special tracing required for CICS as a whole.
- *SPCTRBA*, to specify the level of special tracing required for the BTS business application manager domain.
- *SPCTREM*, to specify the level of special tracing required for the BTS event manager domain.
- *SPCTRSH*, to specify the level of special tracing required for the BTS scheduler services domain.
- *STNTR*, to indicate the level of standard tracing required for CICS as a whole.
- *STNTRBA*, to specify the level of standard tracing required for the BTS business application manager domain.
- *STNTREM*, to specify the level of standard tracing required for the BTS event manager domain.
- *STNTRSH*, to specify the level of standard tracing required for the BTS scheduler services domain.

For more information about system initialization parameters, see *Specifying CICS system initialization parameters*, in the *CICS System Definition Guide*.

© Copyright IBM Corp. 1999, 2011

Defining tracing levels when CICS is running

You can use the CETR transaction to dynamically define, on a running CICS system, the level of tracing that you require for BTS. Use this CETR screen as an example.

Figure 99 shows you what the CETR Component Trace Options screen looks like. To change trace options, you overtype the settings shown on the screen, and then press ENTER.

```
CETR                   Component Trace Options
Overtype where required and press ENTER.                    PAGE 1 OF 2
Component Standard                          Special
--------  --------------------------------  --------------------------
  AP      1                                 1-2
  BA      1                                 1-2
  BF      1                                 OFF
  BM      1                                 OFF
  BR      1                                 1-2
  CP      1                                 1-2
  DC      1                                 OFF
  DD      1                                 1-2
  DI      1                                 1
  DM      1                                 1-2
  DS      1                                 1-2
  DU      1                                 1-2
  EI      1                                 1
  EM      1                                 1-2
  FC      1                                 1-2
  GC      1                                 1-2
  IC      1                                 1

PF:  1=Help   3=Quit   7=Back  8=Forward  9=Messages   ENTER=Change
```

Figure 99. CETR screen for specifying component trace options

With the settings shown, BTS trace entries are made as follows:

- With standard task tracing in effect, from level-1 trace points.
- With special task tracing in effect, from both level-1 and level-2 trace points.

For detailed information about the CETR transaction, see *CETR - trace control*, in the *CICS Supplied Transactions*.

Chapter 20. Extracting BTS information from a CICS system dump

You can use dump formatting keywords to extract BTS information from a CICS system dump.

For information about the dump formatting keywords used to extract BTS information from a CICS system dump, see the CICS Problem Determination Guide.

© Copyright IBM Corp. 1999, 2011

Chapter 21. Tuning BTS performance

You can use the monitoring data produced by CICS to help you improve the performance of BTS.

For introductory information about CICS monitoring, see *The CICS monitoring facility*, in the *CICS Performance Guide*.

The CICS-defined fields in performance class monitoring records are organized in groups. The BTS-related fields are in group DFHCBTS. All the CICS-defined fields, including those in the DFHCBTS group, are listed in *Performance data in group DFHCBTS*, in the *CICS Performance Guide*.

You can print monitoring data using the CICS-supplied sample utility program DFH$MOLS. You can use the IGNORE and SELECT control statements to specify that DFH$MOLS should include or exclude BTS-related monitoring records in the set of records to be printed. For information about DFH$MOLS, see *Sample monitoring data print program (DFH$MOLS)*, in the *CICS Operations and Utilities Guide*.

© Copyright IBM Corp. 1999, 2011

Part 5. BTS Application Programming Reference

Use this programming information about the CICS business transaction services application programming interface (API) to control the way that BTS works.

Table 12. API reference road map

If you want to...	Refer to...
Understand how the BTS API commands fall into functional groups	Chapter 22, "Overview of BTS API commands," on page 211

© Copyright IBM Corp. 1999, 2011

Chapter 22. Overview of BTS API commands

Use the CICS business transaction services application programming interface (API) commands to work on processes and activities, data-containers, events, and objects.

It contains:

- "Process- and activity-related commands"
- "Container commands" on page 213
- "Event-related commands" on page 213
- "Browsing and inquiry commands" on page 216
- "BTS system events" on page 221.

This section groups the API commands by function, giving a brief overview of what each can be used for. For an alphabetic listing of the commands, or for detailed programming information, see BTS application programming commands.

Process- and activity-related commands

For processes and activities, use these CICS business transaction services commands to create, activate, and stop processes and activities, to retrieve status information, and to give a unit of work access to an activity.

Creating, activating, and terminating processes and activities

You can use these commands to create, activate, return to initial state, control the progress of, end, or delete processes and activities.

Use these commands to create processes and activities:

DEFINE PROCESS
Creates a new process.

DEFINE ACTIVITY
Creates a new child activity.

Use these commands to activate a process or activity:

RUN Invokes a program that implements a process or activity. Runs it synchronously or asynchronously with the requestor, in a separate unit of work, and with the transaction attributes specified on the DEFINE PROCESS or DEFINE ACTIVITY command.

LINK ACTIVITY
Invokes a program that implements an activity. Runs it synchronously with the requestor, in the same unit of work, and with the same transaction attributes as the requestor.

LINK ACQPROCESS
Invokes the program that implements the process that is currently acquired by the requestor. Runs the program synchronously with the requestor, in the same unit of work, and with the same transaction attributes as the requestor.

© Copyright IBM Corp. 1999, 2011

Use these commands to return a process or activity to its initial state:

RESET ACQPROCESS
Resets the currently acquired process to its initial state - used before trying the process again.

RESET ACTIVITY
Resets an activity to its initial state - used before trying an activity again.

Use these commands to control the progress of a process or activity:

SUSPEND (BTS)
Prevents a process or activity being reattached if events in its event pool fire.

RESUME
Allows a suspended process or activity to be reattached if events in its event pool fire.

Use these commands to terminate an activity:

RETURN ENDACTIVITY
Indicates that a process or activity is complete.

CANCEL (BTS)
Forces a process or activity to complete.

Use this command to destroy an activity:

DELETE ACTIVITY
Removes a child activity from the BTS repository data set where it is defined.

Retrieving information about activities

You can use the **ASSIGN**, **CHECK ACQPROCESS**, or **CHECK ACTIVITY** commands to retrieve status information for your activities.

Use this command to discover the activity the current unit of work is acting for:

ASSIGN
Returns information about the activity the current unit of work is acting for.

Use these commands to check the response from a process or activity:

CHECK ACQPROCESS
Returns the completion status of the process that is currently acquired by the requestor.

CHECK ACTIVITY
Returns the completion status of an activity.

See also “Browsing and inquiry commands” on page 216.

Relating UOWs and activities

You can use the **ACQUIRE** command to give a unit of work access to an activity.

ACQUIRE
Allows a unit of work executing outside a BTS process to gain access to an activity within the process.

Container commands

You can use CICS business transaction services commands to perform actions on data-containers. For example, **PUT CONTAINER (BTS)**, **GET CONTAINER (BTS)**, **MOVE CONTAINER (BTS)**, and **DELETE CONTAINER (BTS)**.

The CICS business transaction services commands that perform actions on data-containers are:

PUT CONTAINER (BTS)
Use this command to save data in a data-container associated with a specified BTS activity or process. If the named container does not exist, it is created. If the named container exists, its previous contents are overwritten.

GET CONTAINER (BTS)
Use this command to read data from a data-container associated with a specified BTS activity or process.

MOVE CONTAINER (BTS)
Use this command, instead of GET CONTAINER (BTS) and PUT CONTAINER (BTS), as a more efficient way of transferring data between activities. Using GET CONTAINER and PUT CONTAINER, you must:

1. Issue a GET CONTAINER NODATA command to retrieve the length of the data in the source container.
2. Allocate an area of working storage sufficient to hold the data.
3. Issue a GET CONTAINER command to retrieve the data into working storage.
4. Issue a PUT CONTAINER command to store the data in the target container.

Using MOVE CONTAINER, only one command is required and no working storage needs to be allocated. No data is moved; only CICS internal references are changed.

Use MOVE CONTAINER, rather than GET CONTAINER and PUT CONTAINER, if you have no need to keep the source container.

DELETE CONTAINER (BTS)
Use this command to delete a BTS data-container and discard any data that it contains.

Event-related commands

Use the terms that describe BTS events and the event-related commands to perform actions on your event pool.

Terminology

These terms are used to describe CICS business transaction services (BTS) events.

For a more detailed introduction to BTS events, see “BTS events” on page 21.

Event states

An event can be in one of two states: FIRED (true) or NOTFIRED (false). Which state it is in is known as the event's *fire status*.

Atomic events

Atomic events are simple, "low-level" events.

The BTS atomic events are:

Activity completion events
: Events that fire on completion of an activity.

Input events
: Events delivered to an activity when it is activated, conveying the reason for its attachment.

Timer events
: Events that fire when a timer expires.

System events
: Input events defined by the BTS system.

Note: Activity, input, timer, and composite events are referred to as *user-defined events*, because they are defined by the programmer. System events are defined by BTS.

Composite events

A composite event is a method of grouping user-defined (that is, non-system) atomic events in a logical expression, which is named.

An atomic event that is included in a composite event is known as a *subevent*. Subevents that fire are placed on the *subevent queue* of the composite event. Each composite event has a subevent queue associated with it. The subevent queue:

- May be empty
- Contains only those subevents that have fired and not been retrieved.

Reattachment events

An event that fires, and causes an activity to be reattached is known as a *reattachment event*.

All user-defined events except subevents cause the activity to which they are defined to be reattached when they fire. Thus, all user-defined events (both atomic and composite, but excluding subevents) are potentially reattachment events. All system events are reattachment events.

At times, reattachment might occur because of the firing of more than one event. Reattachment events are placed on the *reattachment queue* of the activity, from which they can be retrieved. Each activity has a reattachment queue, which:

- May be empty
- Contains only those reattachment events that have fired and not been retrieved.

Timers

A timer is a BTS object that expires when the system time becomes greater than a specified time or after a specified period has elapsed. When you define a timer, a timer event is automatically associated with it. When the timer expires, its associated event fires.

Event pools

Events are defined within *event pools*.

Each activity has an event pool, which contains the set of events that it recognizes (that is, events that have been defined to it, and system events). An activity's event pool is initialized when the activity is created, and deleted when the activity is deleted. All the event-related commands except FORCE TIMER operate on the event pool associated with the *current* activity.

The event-related commands

You can use event-related commands to perform actions on your event pool of the current activity. You can define input and composite events, control timers and timer events, manipulate events, and check whether an event has fired.

Use these commands to define user events other than activity completion and timer events:

DEFINE INPUT EVENT
Defines an input event.

DEFINE COMPOSITE EVENT
Defines a composite event.

Use these commands to control timers and timer events:

DEFINE TIMER
Defines a timer, and associates an event with it.

FORCE TIMER
Forces early expiry of a timer, and causes the associated event of the timer to fire.

CHECK TIMER
Returns the status of a timer and, if the timer has expired, deletes its associated event.

DELETE TIMER
Deletes a timer and its associated event (if any).

Use these commands to manipulate events:

ADD SUBEVENT
Adds a subevent to a composite event.

REMOVE SUBEVENT
Removes a subevent from a composite event.

DELETE EVENT
Removes an input or composite event from the event pool of the current activity.

Note: DELETE EVENT cannot be used to delete activity completion events (which are implicitly deleted when a response from the completed activity has been acknowledged by a CHECK ACTIVITY command, or when a DELETE ACTIVITY command is issued), timer events, or system events.

RETRIEVE REATTACH EVENT
Retrieves the name of an event that caused the current activity to be reattached and, if the event is atomic, resets its fire status to NOTFIRED.

RETRIEVE SUBEVENT
Retrieves the name of the next subevent in a subevent queue of the composite event, and resets the retrieved fire status of the sub-event to NOTFIRED.

Use this command to check whether an event has fired:

TEST EVENT
Tests whether an event has fired.

Browsing and inquiry commands

Important: The API commands described here are different in kind from the commands described previously in this chapter.

The commands described previously in this chapter are the basic commands used by application programmers to create BTS applications.

The commands described here have more specialized uses. They might be used, for example, in a utility program written to investigate a stuck process. A typical BTS business application does not need to inquire on or browse the objects it creates, and therefore does not use these commands.

You can use CICS business transaction services commands in your programs to search for and examine BTS objects. These commands can be summarized as browsing commands and inquiry commands.

The object-identifiers and names retrieved from the browsing or inquiry commands can be specified on a subset of the other BTS API commands. By using this method you can start actions against specified activities, processes, and data-containers.

Browsing commands

You can use the browsing commands to locate BTS objects and examine their relationships to each other.

The objects that you can browse include:

- Activities
- Data-containers
- Events
- Processes
- Process-types.

Process-types are a special case. They are browsed using the START, NEXT, and END options of the INQUIRE PROCESSTYPE command. INQUIRE PROCESSTYPE is described in Chapter 24, “BTS system programming commands,” on page 227.

Each browse has three commands associated with it:

STARTBROWSE
STARTBROWSE:

1. Tells CICS to begin a browse of a specified type of BTS object.
2. Defines the scope of the browse. Except for browses of processes and process-types, an absence of additional arguments indicates that the browse is to have the scope of the current activity.
3. Returns a browse token which must be included on the remaining commands within the browse.

GETNEXT
Locates the next object within the scope of the browse, or returns the END condition if there are no more objects found.

GETNEXT always returns sufficient information to allow additional actions to be taken. For example, in a browse of the children of a specified parent activity, the GETNEXT ACTIVITY command returns both the name and the identifier (ACTIVITYID) of the next child activity that it finds. The name could be used to decide where the current browse should be paused. The identifier could be used to start a new browse - which might be containers of the child activity, for instance.

ENDBROWSE
Ends the browse.

Inquiry commands

You can use a program to get details of a specific BTS object by issuing an **INQUIRE** command instead of browsing for the object. Note some limitations about using the **INQUIRE** command.

The objects that you can inquire about include:

- Activities
- Data-containers
- Events
- Processes
- Process-types
- Timers.

The object inquired upon might have been located by browsing. For example, a program can use a browse to locate an activity, then issue an INQUIRE ACTIVITYID command to find out the name of the program associated with the activity, the userid under whose authority it runs, or its current completion status.

Note: All INQUIRE commands try to locate a record on a BTS repository data set, and to read information from it if found. This operation does not obtain an exclusive control lock for the record; therefore the data in the record might change while the operation is taking place.

Be careful when issuing INQUIRE commands from within programs that execute as part of an activity, if the commands refer to records which might be modified by the same program. The INQUIRE command always goes to the repository for the record it needs, and might not see changes made by the program. This can lead to unexpected results. For example, a program might define a new activity and then issue a command to inquire upon it, only to be told that the activity does not exist (because the activity-record has not yet been committed to the repository).

Tokens and identifiers

A *browse token* uniquely identifies a browse within a CICS region. An *activity identifier* is a means of uniquely referring to an instance of an activity that has been retrieved from a BTS repository data set.

The browse token that is returned on a **STARTBROWSE** command must be supplied on the corresponding **GETNEXT** and **ENDBROWSE** commands. CICS discards it after the **ENDBROWSE** command.

The lifetime of a browse token is from a **STARTBROWSE** command to an **ENDBROWSE** command or sync point, whichever comes first. Therefore, your applications:

- Should not attempt to use a token after the browse has ended

- Should not attempt to use a token if a sync point is encountered before the browse has completed

When an activity identifier or a process name is known, it can be used as a scoping argument to a new browse. It can also be specified on certain API commands which cause actions to be taken against existing activities or processes, or their containers and events - see "Commands that take identifiers returned by browse operations." The lifetime of an activity identifier is the same as the lifetime of the activity it refers to. Thus, it can be used after an ENDBROWSE and after a sync point.

A data-container or an event cannot be identified in the same way as an activity or a process, because it forms part of a record on a BTS repository data set. Instead, it must be referenced through the activity or process to which it belongs.

Commands that take identifiers returned by browse operations

You can specify the activity identifier or the process name returned by your browse operations on various commands; for example, **GETNEXT ACTIVITY**, **GETNEXT PROCESS**, **INQUIRE PROCESS**, or **INQUIRE ACTIVITYID**. System programmers must be able to modify a business transaction after it has started, especially if your transaction gets into a state in which it cannot complete.

A user-written utility program could, for example:

1. Use a series of browses to locate a particular process or activity
2. When the process or activity is found, inquire about its state
3. Gain control of the process or activity by issuing an ACQUIRE command
4. Correct a processing problem by issuing a further command or commands.

You can specify the activity identifier returned by a GETNEXT ACTIVITY, GETNEXT PROCESS, or INQUIRE PROCESS command on any of the following commands:

- ACQUIRE
- INQUIRE ACTIVITYID
- INQUIRE CONTAINER
- INQUIRE EVENT
- INQUIRE TIMER
- STARTBROWSE ACTIVITY
- STARTBROWSE CONTAINER
- STARTBROWSE EVENT

You can specify the process name returned by a GETNEXT PROCESS (or INQUIRE ACTIVITYID) command on any of the following commands:

- ACQUIRE
- INQUIRE CONTAINER
- INQUIRE PROCESS
- STARTBROWSE ACTIVITY
- STARTBROWSE CONTAINER

After you have acquired a process or activity, you could, for example, issue one or more of the following commands against it:

- CANCEL (BTS)
- CHECK
- DELETE ACTIVITY
- DELETE CONTAINER (BTS)
- FORCE TIMER
- GET CONTAINER (BTS)
- LINK
- MOVE CONTAINER (BTS)
- PUT CONTAINER (BTS)
- RESET ACQPROCESS
- RESUME
- RUN
- SUSPEND (BTS)

Browsing examples

Follow these examples of applications to learn how to use the browsing and inquiry commands.

Example 1

An application, which has not issued any requests to BTS, needs to know whether a particular container belongs to a child of the root activity of a particular process, the name and type of which are known.

```
EXEC CICS INQUIRE PROCESS(pname)
                  PROCESSTYPE(ptype)
                  ACTIVITYID(root_id)

if process found then browse the children of its root activity
  EXEC CICS STARTBROWSE ACTIVITY
                    ACTIVITYID(root_id)
                    BROWSETOKEN(root_token)
  EXEC CICS GETNEXT ACTIVITY(child_name)
                    BROWSETOKEN(root_token)
                    ACTIVITYID(child_id)
  loop while the child is not found and there are more activities
    EXEC CICS GETNEXT ACTIVITY(child_name)
                      BROWSETOKEN(root_token)
                      ACTIVITYID(child_id)
  end child activity browse loop

  if the child we are looking for is found then browse its containers
    EXEC CICS STARTBROWSE CONTAINER
                      ACTIVITYID(child_id)
                      BROWSETOKEN(c_token)
    EXEC CICS GETNEXT CONTAINER(c_name)
                      BROWSETOKEN(c_token)
    loop while container not found and there are more containers
      EXEC CICS GETNEXT CONTAINER(c_name)
                        BROWSETOKEN(c_token)
    end container browse loop

    EXEC CICS ENDBROWSE CONTAINER BROWSETOKEN(c_token)
  EXEC CICS ENDBROWSE ACTIVITY BROWSETOKEN(root_token)
```

Figure 100. Browsing example 1

Example 2

An application, which has not issued any requests to BTS, needs to know whether a particular data-container is one of the global containers associated with a particular process.

If it is not, the program needs to know whether the container is owned by the root activity of that process.

```
EXEC CICS INQUIRE PROCESS(pname) PROCESSTYPE(ptype)
                 ACTIVITYID(root_id)

if process found then browse its containers
  EXEC CICS STARTBROWSE CONTAINER PROCESS(pname) PROCESSTYPE(ptype)
                 BROWSETOKEN(c_token_1)
  EXEC CICS GET NEXT CONTAINER(c_name)
                 BROWSETOKEN(c_token_1)
  loop while container not found and there are more containers
    EXEC CICS GET NEXT CONTAINER(c_name)
                   BROWSETOKEN(c_token_1)
  end process container browse loop

  if container not found browse the root activity's containers
    EXEC CICS STARTBROWSE CONTAINER ACTIVITYID(root_id)
                     BROWSETOKEN(c_token_2)
    EXEC CICS GETNEXT CONTAINER(c_name)
                     BROWSETOKEN(c_token_2)
   loop while container not found and there are more containers
     EXEC CICS GETNEXT CONTAINER(c_name)
                      BROWSETOKEN(c_token_2)
   end root activity's container browse loop

   EXEC CICS ENDBROWSE CONTAINER BROWSETOKEN(c_token_2)
EXEC CICS ENDBROWSE CONTAINER BROWSETOKEN(c_token_1)
```

Figure 101. Browsing example 2

Example 3

A program running as an activation of an activity needs to find whether a named event has been defined to any of its children; that is, whether the event exists in any of the children of the event pool. If the event exists, the program needs to retrieve its fire status.

Because the program starts an activity browse on which no activity identifier or process name is specified, BTS browses the current activity. The program retrieves the identifier of each child activity, and uses the identifier to browse the events child.

Activity identifiers remain valid after the browse that obtained them has ended, they are valid for the life of the activity itself. To illustrate this, the program uses the identifier of the activity whose event pool contains the named event, on an INQUIRE EVENT command, after it has ended the browse.

```
EXEC CICS STARTBROWSE ACTIVITY BROWSETOKEN(parent_token)
loop until the event is found or there are no more child activities
  EXEC CICS GETNEXT ACTIVITY(child_activity_name)
                    BROWSETOKEN(parent_token)
                    ACTIVITYID(child_activity_id)
  EXEC CICS STARTBROWSE EVENT ACTIVITYID(child_activity_id)
                    BROWSETOKEN(event_token)
  loop until event found or there are no more events
    EXEC CICS GETNEXT EVENT(event_name)
                      BROWSETOKEN(event_token)
  end event browse loop

  EXEC CICS ENDBROWSE EVENT BROWSETOKEN(event_token)

end child activity browse loop
EXEC CICS ENDBROWSE ACTIVITY BROWSETOKEN(parent_token)
EXEC CICS INQUIRE EVENT(event_name)
                  ACTIVITYID(child_activity_id)
                  FIRESTATUS(fstatus)
```

Figure 102. Browsing example 3

BTS system events

BTS produces the DFHINITIAL system events as a result of its own processing.

DFHINITIAL is the only BTS system event:

DFHINITIAL
The activity is being attached for the first time in this process, *or* it is trying again after being reset with a RESET ACTIVITY command. An activity must be coded to cope with this event, which specifies that it must perform any initial housekeeping.

Part 6. BTS System Programming Reference

Use this detailed programming information when you work with CICS business transaction services system programming commands.

Table 13. System programming reference road map

If you want to...	Refer to...
Understand how the BTS system programming commands fall into functional groups	Chapter 23, "Overview of BTS system programming commands," on page 225
Check the syntax of a system programming command	Chapter 24, "BTS system programming commands," on page 227

© Copyright IBM Corp. 1999, 2011

Chapter 23. Overview of BTS system programming commands

Use the CICS business transaction services system programming commands to work on BTS process-types.

It contains:

- "Control commands"
- "Inquire command."

You can use CICS command security to restrict access to the commands described in this chapter.

This chapter groups the commands by function, giving a brief overview of what each can be used for. For an alphabetic listing of the commands, or for detailed programming information, see Chapter 24, "BTS system programming commands," on page 227.

Control commands

You can use CICS business transaction services commands in your programs to create and modify BTS process-types.

These commands are:

CREATE PROCESSTYPE
: Builds a PROCESSTYPE definition in the local CICS region, without reference to data in the CICS system definition (CSD) file

DISCARD PROCESSTYPE
: Removes a PROCESSTYPE definition from the local CICS region

SET PROCESSTYPE
: Modifies the attributes of a PROCESSTYPE definition.

Inquire command

You can use the **INQUIRE PROCESSTYPE** command to retrieve the attributes of a process-type.

For information about the other BTS inquiry commands, see "Browsing and inquiry commands" on page 216.

© Copyright IBM Corp. 1999, 2011

Chapter 24. BTS system programming commands

This detailed programming information describes each of the CICS business transaction services (BTS) system programming commands. These commands include **CREATE PROCESSTYPE**, **DISCARD PROCESSTYPE**, **INQUIRE PROCESSTYPE**, and **SET PROCESSTYPE**.

The BTS system programming commands include:

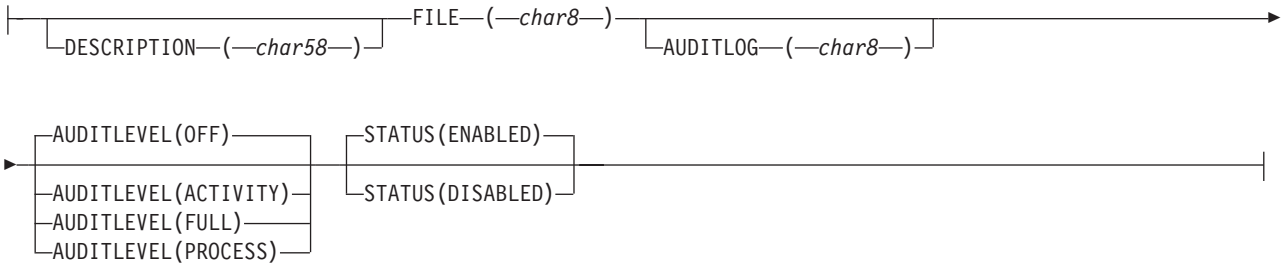
- "CREATE PROCESSTYPE" on page 124
- "DISCARD PROCESSTYPE" on page 126
- "INQUIRE PROCESSTYPE" on page 127
- "SET PROCESSTYPE" on page 130.

CREATE PROCESSTYPE

Use the **CREATE PROCESSTYPE** command to add the definition of a BTS process-type to the local CICS region.

CREATE PROCESSTYPE

ATTRIBUTES:



Conditions: ILLOGIC, INVREQ, LENGERR, NOTAUTH

Description

CREATE PROCESSTYPE adds the definition of a BTS process-type to the local CICS region. The definition is built without reference to data stored on the CSD file. If there is already a process-type by the name you specify in the local CICS region, the new definition replaces the old one; if not, the new definition is added.

A syncpoint is implicit in CREATE PROCESSTYPE processing, except when an exception condition is detected early in processing the command. Uncommitted changes to recoverable resources made up to that point in the task are committed if the CREATE executes successfully and rolled back if not. For other general rules

© Copyright IBM Corp. 1999, 2011

about CREATE commands, see *Creating resource definitions*, in the *CICS System Programming Reference*.

Options

ATTRIBUTES(data-value)
specifies the attributes of the PROCESSTYPE being added. The list of attributes must be coded as a single character string using the syntax shown in **PROCESSTYPE attributes**. For general rules for specifying attributes, see *The ATTRIBUTES option*, in the *CICS System Programming Reference*. For details of specific attributes, see "CEDA DEFINE PROCESSTYPE" on page 119.

ATTRLEN(data-value)
specifies the length in bytes of the character string supplied in the ATTRIBUTES option, as a halfword binary value. The length can be from 0 to 32767.

PROCESSTYPE(data-value)
specifies the name (1-8 characters) of the PROCESSTYPE definition to be added to the CICS region. The acceptable characters are A-Z a-z 0-9 $ @ # . / -_ % & ? ! : | " = ¬ , ; < >. Leading and embedded blank characters are not permitted. If the name supplied is less than eight characters, it is padded with trailing blanks up to eight characters.

Conditions

ILLOGIC
RESP2 values:

2 The command cannot be executed because an earlier CONNECTION or TERMINAL pool definition has not yet been completed.

INVREQ
RESP2 values:

n There is a syntax error in the ATTRIBUTES string, or an error occurred during either the discard or resource definition phase of the processing.

200 The command was executed in a program defined with an EXECUTIONSET value of DPLSUBSET or a program invoked from a remote system by a distributed program link without the SYNCONRETURN option.

LENGERR
RESP2 values:

1 The length you have specified in ATTRLEN is negative.

NOTAUTH
RESP2 values:

100 The user associated with the issuing task is not authorized to use this command.

101 The user associated with the issuing task is not authorized to create a PROCESSTYPE definition with this name.

102 The caller does not have surrogate authority to install the resource with the particular userid.

DISCARD PROCESSTYPE

Use the **DISCARD PROCESSTYPE** command to remove the definition of a specified process-type from the local CICS region.

DISCARD PROCESSTYPE

Conditions: INVREQ, NOTAUTH, PROCESSERR

Description

DISCARD PROCESSTYPE removes the definition of a specified process-type from the local CICS region.

Note:

1. Only disabled process-types can be discarded.
2. If you are using BTS in a single CICS region, you can use the DISCARD PROCESSTYPE command to remove process-types. However, if you are using BTS in a sysplex, it is strongly recommended that you use CPSM to remove them. If you don't use CPSM, problems could arise if Scheduler Services routes to this region work that requires a discarded definition.

Options

PROCESSTYPE(data-value)
specifies the name (1-8 characters) of the process-type to be removed.

Conditions

INVREQ
RESP2 values:

2 The process-type named in the PROCESSTYPE option is not disabled.

NOTAUTH
RESP2 values:

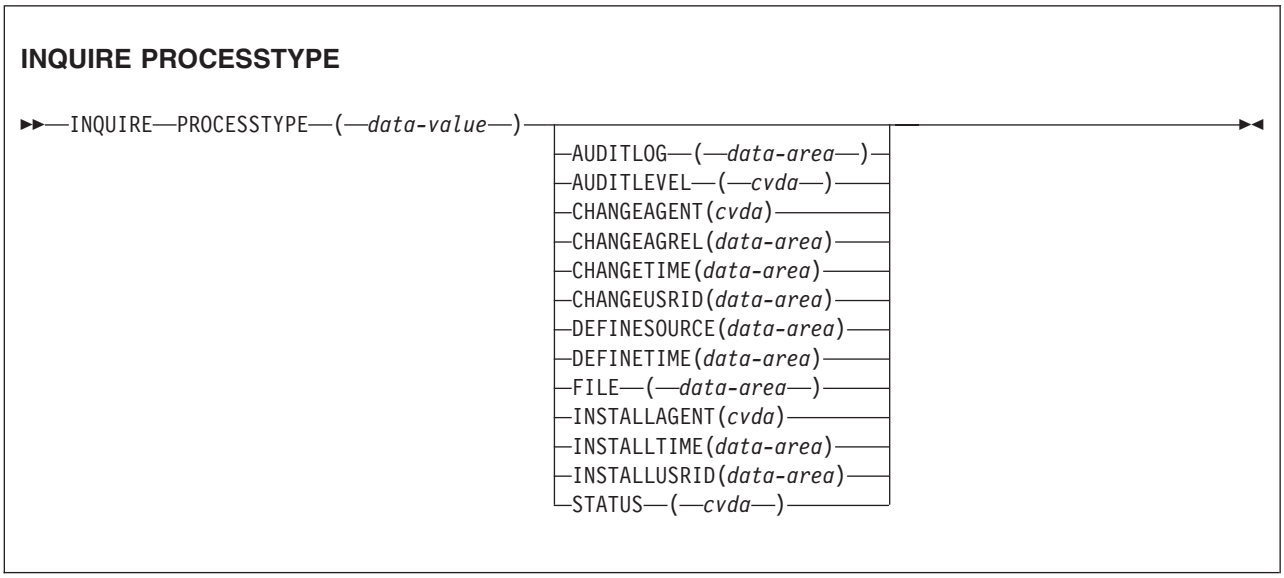
100 The user associated with the issuing task is not authorized to use this command.

PROCESSERR
RESP2 values:

1 The process-type named in the PROCESSTYPE option is not defined in the process-type table (PTT).

INQUIRE PROCESSTYPE

Use the **INQUIRE PROCESSTYPE** command to retrieve the attributes of a process-type.



Conditions: NOTAUTH, PROCESSERR

Description

INQUIRE PROCESSTYPE returns the attributes of a specified process-type.

The resource signature

You can use this command to retrieve the resource signature fields. You can use these fields to manage resources by capturing details of when the resource was defined, installed, and last changed. For more information, see Auditing resources in the Resource Definition Guide. The resource signature fields are CHANGEAGENT, CHANGEAGREL, CHANGETIME, CHANGEUSRID, DEFINESOURCE, DEFINETIME, INSTALLAGENT, INSTALLTIME, and INSTALLUSRID. See Summary of the resource signature field values in the Resource Definition Guide for detailed information about the content of the resource signature fields.

Options

AUDITLEVEL(*cvda***)**
Indicates the level of audit currently active for processes of the specified type. CVDA values are as follows:

ACTIVITY
Activity-level auditing. Audit records are written from the following points:

- The process audit points
- The activity primary audit points.

FULL Full auditing. Audit records are written from the following points:

- The process audit points
- The activity primary *and* secondary audit points.

OFF No audit trail records are written.

PROCESS
Process-level auditing. Audit records are written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see "Specifying the level of audit logging" on page 170.

AUDITLOG(*data-area***)**
Returns the 8-character name of the CICS journal used as the audit log for processes of the specified type.

CHANGEAGENT(*cvda***)**
Returns a CVDA value that identifies the agent that made the last change to the resource definition. The possible values are as follows:

CREATESPI
The resource definition was last changed by an **EXEC CICS CREATE** command.

CSDAPI
The resource definition was last changed by a CEDA transaction or the programmable interface to DFHEDAP.

CSDBATCH
The resource definition was last changed by a DFHCSDUP job.

DREPAPI
The resource definition was last changed by a CICSPlex SM BAS API command.

CHANGEAGREL(*data-area***)**
Returns a 4-digit number of the CICS release that was running when the resource definition was last changed.

CHANGETIME(*data-area***)**
Returns an ABSTIME value that represents the time stamp when the resource definition was last changed. For more information about the format of the ABSTIME value, see FORMATTIME in CICS Application Programming.

CHANGEUSRID(*data-area***)**
Returns the 8-character user ID that ran the change agent.

DEFINESOURCE(*data-area***)**
Returns the 8-character source of the resource definition. The DEFINESOURCE value depends on the CHANGEAGENT value. For more information, see Summary of the resource signature field values in the Resource Definition Guide.

DEFINETIME(*data-area***)**
Returns an ABSTIME value that represents the time stamp when the resource definition was created.

FILE(*data-area***)**
Returns the 8-character name of the CICS file associated with the process-type.

INSTALLAGENT(*cvda***)**
Returns a CVDA value that identifies the agent that installed the resource. The possible values are as follows:

CREATESPI
The resource was installed by an **EXEC CICS CREATE** command.

CSDAPI
The resource was installed by a CEDA transaction or the programmable interface to DFHEDAP.

GRPLIST
The resource was installed by **GRPLIST INSTALL**.

INSTALLTIME(*data-area***)**
Returns an ABSTIME value that represents the time stamp when the resource was installed.

INSTALLUSRID(*data-area***)**
Returns the 8-character user ID that installed the resource.

PROCESSTYPE(*data-value***)**
Specifies the name (1 - 8 characters) of the process-type being inquired on.

STATUS(*cvda***)**
Indicates whether new processes of the specified type can currently be defined. CVDA values are as follows:

DISABLED
The installed definition of the process-type is disabled. New processes of this type cannot be defined.

ENABLED
The installed definition of the process-type is enabled. New processes of this type can be defined.

Conditions

NOTAUTH
RESP2 values:

100 The user associated with the issuing task is not authorized to use this command.

PROCESSERR
RESP2 values:

1 The process-type specified on the PROCESSTYPE option could not be found.

Browsing process-type definitions

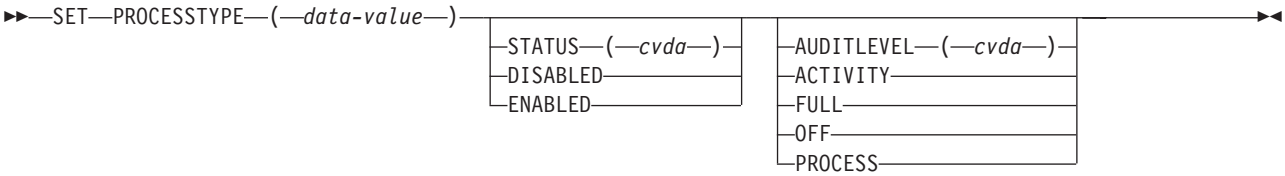
You can browse through all the process-type definitions in your system by using the browse options (START, NEXT, and END) on **INQUIRE PROCESSTYPE** commands.

In browse mode, the definitions are returned in alphabetic order. For general information about browsing, syntax, exception conditions, and examples, see *Browsing resource definitions*, in the *CICS System Programming Reference*.

SET PROCESSTYPE

Use the **SET PROCESSTYPE** command to change the attributes of a process-type.

SET PROCESSTYPE



Conditions: INVREQ, NOTAUTH, PROCESSERR

Description

SET PROCESSTYPE allows you to change the current state of audit logging and the enablement status of PROCESSTYPE definitions installed on this CICS region.

Note: Process-types are defined in the process-type table (PTT). CICS uses the entries in this table to maintain its records of processes (and their constituent activities) on external data sets. If you are using BTS in a single CICS region, you can use the SET PROCESSTYPE command to modify your process-types. However, if you are using BTS in a sysplex, it is strongly recommended that you use CPSM to make such changes. This is because it is essential to keep resource definitions in step with each other, across the sysplex.

Options

AUDITLEVEL(cvda)
specifies the level of audit logging to be applied to processes of this type.

Note: If the AUDITLOG attribute of the installed PROCESSTYPE definition is not set to the name of a CICS journal, an error is returned if you try to specify any value other than OFF.

The CVDA values are:

ACTIVITY
Activity-level auditing. Audit records will be written from:

1. The process audit points
2. The activity primary audit points.

FULL Full auditing. Audit records will be written from:

1. The process audit points
2. The activity primary *and* secondary audit points.

OFF No audit trail records will be written.

PROCESS
Process-level auditing. Audit records will be written from the process audit points only.

For details of the records that are written from the process, activity primary, and activity secondary audit points, see “Specifying the level of audit logging” on page 170.

PROCESSTYPE(value)
specifies the 8-character name of a process-type defined in the process-type table (PTT), whose attributes are to be changed.

STATUS(cvda)
specifies whether new processes of this type can be created. The CVDA values are:

DISABLED
The installed definition of the process-type is disabled. New processes of this type cannot be defined.

ENABLED
The installed definition of the process-type is enabled. New processes of this type can be defined.

Conditions

INVREQ
RESP2 values:

2 The process-type is not disabled, and therefore cannot be enabled.

3 You have specified an invalid CVDA value on the AUDITLEVEL option.

5 You have specified an invalid CVDA value on the STATUS option.

6 You have specified a value of FULL, PROCESS, or ACTIVITY on the AUDITLEVEL option, but the AUDITLOG attribute of the PROCESSTYPE definition does not specify an audit log.

NOTAUTH
RESP2 values:

100 The user associated with the issuing task is not authorized to use this command.

PROCESSERR
RESP2 values:

1 The process-type named in the PROCESSTYPE option is not defined in the process-type table (PTT).

Part 7. Appendixes

© Copyright IBM Corp. 1999, 2011

Appendix. The BTS sample application

The CICS-supplied sample BTS application is a skeleton sales application, consisting of order, credit check, stock check, delivery note, invoice, and payment-not-received-reminder activities. You can use the sample as the basis of your own sales application. However, before using it in a production environment you must add installation-specific code.

The user enters an order, and is checked for credit-worthiness. If all is well, and the required goods are in stock, a delivery note and invoice are produced. A timer is set, to check for payment of the invoice. If the timer expires, a reminder is produced and the timer is reset. A container is used to track the number of reminders sent. If payment is received, the timer is canceled and the process completes.

The sample is implemented as a set of COBOL programs. These programs are supplied, in source code, in the SDFHSAMP library, together with copybooks and BMS maps and map sets. Resource definitions are in RDO group DFH$CBTS.

The sample uses a repository file called DFHBARF. RDO group DFH$BARF contains a definition of DFHBARF. For an example of the JCL you might use to create a repository, see Figure 53 on page 112.

Note: The supplied definition of DFHBARF uses LSRPOOL 1. Ensure that LSRPOOL 1 is defined on your system with a MAXKEYLENGTH of at least 50.

Table 14 shows the transactions and programs that make up the sample application.

Table 14. Transactions and programs in the BTS sample application

Transaction	Program	Function
SALM	DFH0SAL0	Order entry. The order entry transaction is a regular CICS transaction running outside the BTS environment.
SAL1	DFH0SAL1	Accepts the order and defines and runs the SALE process.
SALE	DFH0SAL2	The root activity of the SALE process. This is the main logic-control program. It defines and runs the following activities: • Customer credit-check, run synchronously with the root activity • Stock check, run synchronously • Delivery note, run asynchronously • Invoice, run asynchronously • Payment-not-received reminder, run asynchronously (if payment is not received before the timer expires).
RED1	DFH0RED1	Implements the customer credit-check activity.
STOC	DFH0STOC	Implements the stock-level check activity.
DEL1	DFH0DEL1	Implements the delivery note activity.

© Copyright IBM Corp. 1999, 2011

Table 14. Transactions and programs in the BTS sample application (continued)

Transaction	Program	Function
INV1	DFH0INV1	Implements the invoice activity.
REM1	DFH0REM1	Implements the payment-not-received reminder activity.
PAYM	DFH0PAY0	Payment input. The payment input transaction is a regular CICS transaction running outside the BTS environment.
PAY1	DFH0PAY1	Accepts the payment information, then acquires and runs the appropriate SALE process. This causes the SALE process to complete.

Table 15 shows the copybooks, maps, and map sets supplied with the sample application.

Table 15. Copybooks, maps and map sets supplied with the BTS sample application

Type	Member	Description
Copybook	DFH0CONT	Container definitions.
Copybook	DFH0SALC	Defines BMS map fields for the SALM transaction.
Copybook	DFH0PAYC	Defines BMS map fields for the PAYM transaction.
Map/Mapset	DFH0SALM	Source of the BMS map for the SALM transaction. The map set is DFH0SAS. The name of the map is ORDER.
Map/Mapset	DFH0PAYM	Source of the BMS map for the PAYM transaction. The map set is DFH0PAS. The name of the map is PAYMENT.

The source code modules contain explanatory comments.

You can use the sample as the basis of your own sales application. However, before using it in a production environment you must add installation-specific code.

Important: Sample program DFH0SAL2 contains the following command:

```
EXEC CICS DEFINE ACTIVITY('CREDIT-CHECK')
                 TRANSID('RED1')
                 PROGRAM('DFH0RED1')
                 USERID('USER0001')
                 RESP(RESP-AREA)
                 RESP2(RESP2-AREA)
END-EXEC.
```

You must change the value of the `USERID` option to specify a user ID that is valid in your installation. You must enter the customer number and order number because they are used by the **`DEFINE PROCESS`** command, the **`PROCESS`** must have a unique name.

Running the sample application from the WWW

You can run the sample application from a Web browser. The DFH0SAL0 program source code contains detailed instructions about how you can put a Web front end on to the two BTS processes used in the sample.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply in the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN.

© Copyright IBM Corp. 1999, 2011

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Programming interface information

This book is intended to help you use CICS business transaction services. This book documents General-use Programming Interface and Associated Guidance Information provided by CICS.

General-use programming interfaces allow you to write programs that obtain the services of CICS.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Bibliography

CICS books for CICS Transaction Server for z/OS

General

CICS Transaction Server for z/OS Program Directory, GI13-0565
CICS Transaction Server for z/OS What's New, GC34-7192
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1, GC34-7188
CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2, GC34-7189
CICS Transaction Server for z/OS Upgrading from CICS TS Version 4.1, GC34-7190
CICS Transaction Server for z/OS Installation Guide, GC34-7171

Access to CICS

CICS Internet Guide, SC34-7173
CICS Web Services Guide, SC34-7191

Administration

CICS System Definition Guide, SC34-7185
CICS Customization Guide, SC34-7161
CICS Resource Definition Guide, SC34-7181
CICS Operations and Utilities Guide, SC34-7213
CICS RACF Security Guide, SC34-7179
CICS Supplied Transactions, SC34-7184

Programming

CICS Application Programming Guide, SC34-7158
CICS Application Programming Reference, SC34-7159
CICS System Programming Reference, SC34-7186
CICS Front End Programming Interface User's Guide, SC34-7169
CICS C++ OO Class Libraries, SC34-7162
CICS Distributed Transaction Programming Guide, SC34-7167
CICS Business Transaction Services, SC34-7160
Java Applications in CICS, SC34-7174

Diagnosis

CICS Problem Determination Guide, GC34-7178
CICS Performance Guide, SC34-7177
CICS Messages and Codes Vol 1, GC34-7175
CICS Messages and Codes Vol 2, GC34-7176
CICS Diagnosis Reference, GC34-7166
CICS Recovery and Restart Guide, SC34-7180
CICS Data Areas, GC34-7163
CICS Trace Entries, SC34-7187
CICS Supplementary Data Areas, GC34-7183
CICS Debugging Tools Interfaces Reference, GC34-7165

Communication

CICS Intercommunication Guide, SC34-7172
CICS External Interfaces Guide, SC34-7168

© Copyright IBM Corp. 1999, 2011

Databases

CICS DB2 Guide, SC34-7164

CICS IMS Database Control Guide, SC34-7170

CICS Shared Data Tables Guide, SC34-7182

CICSPlex SM books for CICS Transaction Server for z/OS

General

CICSPlex SM Concepts and Planning, SC34-7196
CICSPlex SM Web User Interface Guide, SC34-7214

Administration and Management

CICSPlex SM Administration, SC34-7193
CICSPlex SM Operations Views Reference, SC34-7202
CICSPlex SM Monitor Views Reference, SC34-7200
CICSPlex SM Managing Workloads, SC34-7199
CICSPlex SM Managing Resource Usage, SC34-7198
CICSPlex SM Managing Business Applications, SC34-7197

Programming

CICSPlex SM Application Programming Guide, SC34-7194
CICSPlex SM Application Programming Reference, SC34-7195

Diagnosis

CICSPlex SM Resource Tables Reference Vol 1, SC34-7204
CICSPlex SM Resource Tables Reference Vol 2, SC34-7205
CICSPlex SM Messages and Codes, GC34-7201
CICSPlex SM Problem Determination, GC34-7203

Other CICS publications

The following publications contain further information about CICS, but are not provided as part of CICS Transaction Server for z/OS, Version 4 Release 2.

Designing and Programming CICS Applications, SR23-9692

CICS Application Migration Aid Guide, SC33-0768

CICS Family: API Structure, SC33-1007

CICS Family: Client/Server Programming, SC33-1435

CICS Family: Interproduct Communication, SC34-6853

CICS Family: Communicating from CICS on System/390, SC34-6854

CICS Transaction Gateway for z/OS Administration, SC34-5528

CICS Family: General Information, GC33-0155

CICS 4.1 Sample Applications Guide, SC33-1173

CICS/ESA 3.3 XRF Guide , SC33-0661

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully.

You can perform most tasks required to set up, run, and maintain your CICS system in one of these ways:

- using a 3270 emulator logged on to CICS
- using a 3270 emulator logged on to TSO
- using a 3270 emulator as an MVS system console

IBM Personal Communications provides 3270 emulation with accessibility features for people with disabilities. You can use this product to provide the accessibility features you need in your CICS system.

© Copyright IBM Corp. 1999, 2011

Index

Numerics

3270 bridge support
  conversational transactions 102
  introduction 10, 99
  pseudoconversational transactions 103
  resource definition 101
  running a 3270 transaction 99
  sample programs 106

A

abends, of activities 166
acquiring a process 61
acquiring an activity 61, 69
activities
  abends 166
  acquiring access to 61
  activation of 17
  asynchronous 17
    checking response from 18
  auditing of
    introduction 169
    specifying the logging level 170
  browsing with CBAM 133
  checking response from
    asynchronous 51
    synchronous 49
  child 15
  compared with terminal-related pseudoconversations 17
  compensation 85
  data-containers 20
  described 15
  destruction of 18, 44
  identifiers 217
  implementation by existing 3270–based transactions 99
  lifetime of 18
  modes 18
  parallel 53
  parent 15
  processing states 18
  root 15
  security of 123, 124
  synchronous 17
    checking response from 18
  syncpoint 20
  transferring input and output data 44
  unserviceable requests 167
activity abends 166
activity completion events 22
activity identifiers
  described 217
activity-related commands
  INQUIRE PROCESSTYPE 127, 230
  overview 211
administration
  controlling BTS
    operator commands 133
  resource definition 117
  security 123
  sysplex considerations
    dealing with affinities 160
    using CPSM 159
  system definition
    defining local request queue data set 113
    defining repository data sets 111
    naming the distributed routing program 115
affinities, in a sysplex 160
ALL
  CEMT INQUIRE TASK 144
API commands
  activity-related 211
  browse tokens 217
  browsing commands 216
  container 213
  event-related 213
  examples 219
  inquiry commands 217
  overview 211
  that take activity identifiers 218
asynchronous activities
  checking response from 51
  how invoked 17
Atomic events 22
attach-time security 124
audit commands
  introduction to 169
audit trail
  examples 173
  introduction to 169
  sharing a logstream between CICS regions 173
  specifying the logging level 170
  utility program, DFHATUP 176
audit trail utility program, DFHATUP 176
AUDITLEVEL attribute
  PROCESSTYPE definition 120
AUDITLOG attribute
  PROCESSTYPE definition 121

B

browse tokens 217
browsing commands
  CBAM 133
  INQUIRE PROCESSTYPE 127, 230
BTS messages 201
BTS sample application 237
BTS-set
  dealing with affinities 160
  how to create 154
  introduction to 149
  scope of 149
business transaction
  described 3
  Sale example application 33
  sample BTS application 237

C

CBAM, CICS-supplied transaction 133
CEDA DEFINE PROCESSTYPE command 119
CEMT INQUIRE PROCESSTYPE
  ALL 141
  AUDITLEVEL 142
  AUDITLOG 142
  FILE 143
  STATUS 143
  value 141
CEMT INQUIRE TASK
  ACTIVITY 145
  ACTIVITYID 145
  PROCESS 145
  PROCESSTYPE 145
CEMT SET PROCESSTYPE
  (value) 147
  ACTIVITY 146
  ALL 146
  DISABLED 147
  ENABLED 147
  FULL 146
  OFF 146
  PROCESS 146
CEMT transaction
  PROCESSTYPE 140, 146
  TASK 144
child activity
  described 15
CICS business transaction services
  3270 bridge support 99
  administration
    controlling BTS 133
    defining local request queue data set 113
    defining repository data sets 111
    performance tuning 207
    resource definition 117
    security 123
    sysplex considerations 149
    system definition 111
  browsing BTS objects 133
  client/server processing 10
  components
    activities 15
    data-containers 20
    events 21
    introduction to 7
    processes 15
  external interactions
    acquiring activities 61
    acquiring an activity 69
    acquiring processes 61
    client/server processing 63

© Copyright IBM Corp. 1999, 2011

CICS business transaction services *(continued)*
introduction to 7
monitoring 207
parallel activities 53
problem determination 163
recovery and restart 10
reusing existing code 99
sample application 237
security
attach-time 124
command-level 124
resource-level 123
sysplex support 11
user-related activities 70
Web Interface support 10
CICS-supplied transactions
CBAM 133
CEMT INQUIRE PROCESSTYPE 140
CEMT INQUIRE TASK 144
CEMT SET PROCESSTYPE 146
CICSPlex SM
use with BTS 11, 159
client/server processing
example 63
introduction 10
cold start, of CICS 168
command-level security 124
compensation
example 86
how to implement 85
introduction to 85
components of BTS
activities 15
data-containers 20
events 21
introduction to 7
processes 15
composite event
described 23
conditions
CREATE PROCESSTYPE command 126, 228
DISCARD PROCESSTYPE command 127, 229
INQUIRE PROCESSTYPE command 129, 232
SET PROCESSTYPE command 131, 234
container commands
overview 213
controlling BTS
operator commands 133
CPSM, use with BTS 159
CREATE PROCESSTYPE command 124, 227
conditions 126, 228

D

DASD-only logstreams, restrictions on sharing 173
data flow
in basic Sale application 33
in parallel activities example 53
in user-related example 71
data sets
local request queue 113
repository 111
data-container
described 20
data-containers
destruction of 20, 44
lifetime of 20
DEFINE ACTIVITY 22
DEFINE COMPOSITE EVENT 22
DEFINE INPUT EVENT 22
DEFINE TIMER 22
defined activity userid 123
defined process userid 123
defining BTS resources to CICS
local request queue data set 113
process-types 119
repository data sets 111
deleting an event 25
DESCRIPTION attribute
PROCESSTYPE definition 121
destruction of activities 18, 44
destruction of data-containers 20, 44
DFH$MOLS, sample monitoring data print program 207
DFH0CBAC, sample client activity program for 3270 bridge 106
DFH0CBAE, sample bridge exit program 106
DFHATUP, audit trail utility program 176
DFHBARUP, repository utility program 191
DFHDSRP, distributed routing program 115
how to write 157
relation to dynamic routing program 156
DFHINITIAL 22
DFHINITIAL system event 221
DISCARD PROCESSTYPE command 126, 229
conditions 127, 229
distributed routing
introduction to 151
of BTS activities 156, 157
creating a BTS-set 154
which activities can be dynamically routed? 150
routing program, DFHDSRP 156, 157
distributed routing program, DFHDSRP 115
how to write 157
relation to dynamic routing program 156
DSRTPGM, system initialization parameter 115
dump formatting keywords, for BTS 205
dynamic routing
of BTS activities
creating a BTS-set 154
naming the distributed routing program 115
using a distributed routing program 156
using CPSM 159
dynamic routing *(continued)*
of BTS activities *(continued)*
which activities can be dynamically routed? 150
Dynamic routing 167

E

emergency restart, of CICS 168
errors
checking response from asynchronous activities 51
checking response from synchronous activities 49
event
composite 23
deleting 25
described 21
reattaching an activity on firing of 26
event-related commands
overview 213
examples
API commands 219
attach-time security 124
audit trails 173
basic Sale application
data flow 33
initial request 35
overview 33
root activity 38
transferring data to activities 44
browsing 219
client/server processing
client program 63
server program 66
compensation 86
error handling 51
output from DFHATUP 180, 181
output from DFHBARUP 194
parallel activities
data flow 53
root activity 54
RACF commands 123, 124
surrogate security checking 123
user-related activities
data flow 71
implementation of activity 77
root activity 72
exceptional conditions
checking response from asynchronous activities 51
checking response from synchronous activities 49
existing code, reuse in BTS applications
3270 bridge support
conversational transactions 102
introduction 10, 99
pseudoconversational transactions 103
resource definition 101
running a 3270 transaction 99
sample programs 106

F

failure, of CICS 168

FILE attribute
PROCESSTYPE definition 121

I

initial request
in basic Sale application 35
initial start, of CICS 168
input and output data, transfer of 44
input events 22
INQUIRE PROCESSTYPE
command 127, 230
conditions 129, 232
interacting with non-BTS code
acquiring activities 61
acquiring an activity 69
acquiring processes 61
client/server processing 63
introduction to BTS 7

L

lifetime of activities 18
lifetime of data-containers 20
local request queue
data set, definition of 113

M

messages, BTS-related 201
modes, of an activity
described 18
monitoring
performance class data 207
using DFH$MOLS 207
monitoring data print program,
DFH$MOLS 207

N

notation, syntax ix

O

operator commands
CBAM 133
CEMT INQUIRE PROCESSTYPE 140
CEMT INQUIRE TASK 144
CEMT SET PROCESSTYPE 146

P

parallel activities
data flow 53
example 54
introduction 53
parent activity
described 15
transferring data to child
activities 44
performance class monitoring data 207
performance tuning
introduction 207
problem determination
activity abends 166
BTS-related messages 201
CICS failures
cold starts 168
emergency starts 168
initial starts 168
dump formatting keywords 205
examining repository records
utility program, DFHBARUP 191
introduction to 163
stuck processes
due to application errors 163
due to unserviceable requests 167
trace levels 203
unserviceable requests 167
using an audit trail
examples 173
introduction 169
sharing a logstream between CICS
regions 173
specifying the logging level 170
utility program, DFHATUP 176
process
acquiring access to 61
auditing of
introduction 169
specifying the logging level 170
browsing with CBAM 133
categorizing 15
data-containers 20
described 15
identifier 217
security of 123, 124
stuck 163
unserviceable requests 167
process identifiers 217
process-type
browsing with CBAM 133
CBAM requests 133
CEMT INQUIRE requests 140
CEMT SET requests 146
PROCESSTYPE attribute
PROCESSTYPE definition 121
PROCESSTYPE command
CEMT INQUIRE transaction 140
CEMT SET transaction 146
PROCESSTYPE definition
AUDITLEVEL attribute 120
AUDITLOG attribute 121
DESCRIPTION attribute 121
FILE attribute 121
PROCESSTYPE attribute 121
STATUS attribute 121
pseudoconversational
terminal-related pseudoconversation
comparison with multiple
activations of an activity 17

R

RACF
example commands 123, 124
RDO commands
PROCESSTYPE 119
reattaching an activity on firing of an
event 26
recovery and restart
introduction to 10
repository
data sets, definition of 111
examining records on 191
utility program, DFHBARUP 191
repository utility program,
DFHBARUP 191
Resolving unserviceable requests 167
resource definition
defining BTS resources to CICS 117
RDO commands
CEDA DEFINE
PROCESSTYPE 119
resource-control commands
CREATE PROCESSTYPE 124, 227
DISCARD PROCESSTYPE 126, 229
SET PROCESSTYPE 130, 233
resource-level security 123
RETRIEVE REATTACH EVENT 22
reusing existing code
3270 bridge support
conversational transactions 102
introduction 10, 99
pseudoconversational
transactions 103
resource definition 101
running a 3270 transaction 99
sample programs 106
root activity
described 15
in basic Sale application 38
in compensation example 88
in parallel activities example 54
in user-related example 72
routing of BTS activities
naming the distributed routing
program 115
unserviceable requests 167

S

Sale example application
compensation 86
data flow 33
error handling 51
initial request 35
overview 33
parallel activities 53
root activity 38
transferring data to activities 44
user-related activities 70
sample BTS application 237
sample programs
BTS application 237
for 3270 bridge support
DFH0CBAC, client activity 106
DFH0CBAE, bridge exit 106
security
of activities
attach-time 124
defined activity userid 123
resource-level 123
of BTS commands 124
of processes
attach-time 124
defined process userid 123

security *(continued)*
of processes *(continued)*
resource-level 123
SET PROCESSTYPE command 130, 233
conditions 131, 234
sharing an audit logstream between CICS regions 173
SPCTR, system initialization parameter 203
special trace, setting the level of 203
standard trace, setting the level of 203
Static routing 167
STATUS attribute
PROCESSTYPE definition 121
stuck processes 163
surrogate security checking 123
synchronous activities
checking response from 49
how invoked 17
syncpoint
user, issued by an activity 20
syntax notation ix
sysplex considerations
BTS's sysplex support 11, 149
dealing with affinities 160
introduction 11
using CPSM 159
system definition 111
system events 22
DFHINITIAL 221
system initialization parameters
DSRTPGM 115
SPCTR 203
SPCTRBA 203
STNTR 203
STNTRBA 203
system programming commands
control commands 225
CREATE PROCESSTYPE 124, 227
DISCARD PROCESSTYPE 126, 229
inquire command 225
INQUIRE PROCESSTYPE 127, 230
overview 225
SET PROCESSTYPE 130, 233

T

task
CEMT INQUIRE requests 144
TASK command
CEMT INQUIRE transaction 144
TCLASS
CEMT INQUIRE TASK 144
terminal-related pseudoconversation
comparison with an activity that is activated multiple times 17
timer
described 21
timer events 22
trace
special, setting the level of 203
standard, setting the level of 203
trademarks 240
transaction affinities, in a sysplex 160
transferring data to child activities 44

U

unserviceable requests 167
Unserviceable routing requests 167
user-related activities
example 70
utility programs
audit trail utility, DFHATUP 166, 176
repository utility, DFHBARUP 166, 191

V

value
CEMT INQUIRE TASK 144

W

Web Interface
introduction 10

Readers' Comments — We'd Like to Hear from You

CICS Transaction Server for z/OS
Version 4 Release 2
Business Transaction Services

Publication No. SC34-7160-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +44 1962 816151
- Send your comments via email to: idrcf@uk.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC34-7160-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM United Kingdom Limited
User Technologies Department (MP095)
Hursley Park
Winchester
Hampshire
United Kingdom
SO21 2JN

Fold and Tape | **Please do not staple** | Fold and Tape

SC34-7160-00

Cut or Fold
Along Line

SC34-7160-00