

CICS Transaction Server for z/OS  
Version 4 Release 1



# Business Transaction Services



CICS Transaction Server for z/OS  
Version 4 Release 1



# Business Transaction Services

**Note**

Before using this information and the product it supports, read the information in "Notices" on page 231.

This edition applies to Version 4 Release 1 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, 2010.**

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

---

# Contents

<b>Dedication</b> . . . . .	<b>vii</b>
-----------------------------	------------

<b>Preface</b> . . . . .	<b>ix</b>
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

<b>Changes in CICS Transaction Server for z/OS, Version 4 Release 1</b> . . . . .	<b>xi</b>
---	-----------

---

## 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 problems . . . . .	4
The solution . . . . .	5

### 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
Requirements for BTS . . . . .	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 syncpoints . . . . .	19
BTS data-containers . . . . .	20
Lifetime of data-containers . . . . .	20
BTS timers. . . . .	20

Timer management tips . . . . .	21
BTS events . . . . .	21
Atomic events . . . . .	21
Composite events . . . . .	22
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
Retrying failed activities . . . . .	52

### Chapter 6. Running parallel BTS activities . . . . . 53

An example of parallel activities . . . . .	53
Data flow . . . . .	53
The root activity. . . . .	54

### Chapter 7. Interacting with BTS processes and activities . . . . . 61

Introduction . . . . .	61
Acquiring processes and activities. . . . .	61
Using client/server processing . . . . .	63
A client/server example . . . . .	63
Acquiring an activity . . . . .	70
A user-related example . . . . .	71
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 . . . . .	100
Running more complex transactions . . . . .	101
Intermediate output messages . . . . .	101
Conversational transactions . . . . .	101
Pseudoconversational transactions . . . . .	103
Things to consider . . . . .	105
Using timers . . . . .	105
Abend processing . . . . .	105
Transaction restart . . . . .	106
Sample programs . . . . .	106

---

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

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

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

<b>Chapter 12. Security in BTS . . . . . 123</b>	
Resource security in BTS . . . . .	123
Process and activity userids . . . . .	123
Attach-time security for processes and activities . . . . .	124
Command security in BTS . . . . .	124

<b>Chapter 13. BTS operator commands . . . . . 125</b>	
CBAM—BTS browser . . . . .	125
Running the transaction . . . . .	125
CEMT INQUIRE PROCESSTYPE . . . . .	133
CEMT INQUIRE TASK . . . . .	136
CEMT SET PROCESSTYPE . . . . .	138

<b>Chapter 14. Using BTS in a sysplex . . . . . 141</b>	
The scope of a BTS-set . . . . .	141
A note about audit logs . . . . .	141
Dynamic routing of BTS activities . . . . .	142
Which BTS activities can be dynamically routed? . . . . .	142
Understanding distributed routing . . . . .	143
Controlling BTS dynamic routing . . . . .	146
Creating a BTS-set . . . . .	146
Naming the routing program . . . . .	148
Using a CICS distributed routing program . . . . .	148
How the distributed routing program relates to the dynamic routing program . . . . .	148
Writing a distributed routing program . . . . .	149
Using CICSplex SM with BTS . . . . .	151
Overview of CICSplex SM Workload Management . . . . .	151
Using CICSplex SM to route BTS activities . . . . .	151

---

## **Part 4. BTS problems and performance . . . . . 153**

<b>Chapter 15. Having problems? . . . . . 155</b>	
Dealing with stuck processes . . . . .	155
Application design errors . . . . .	155
Restarting stuck processes . . . . .	156
Dealing with activity abends . . . . .	158
Dealing with unserviceable requests . . . . .	158
Unserviceable routing requests . . . . .	158
How CICS handles unserviceable requests . . . . .	159
Resolving unserviceable requests . . . . .	159
Dealing with CICS failures . . . . .	160
Emergency starts . . . . .	160
Initial and cold starts . . . . .	160

<b>Chapter 16. Creating a BTS audit trail . . . . . 161</b>	
Introduction to BTS audit trails . . . . .	161
Specifying the level of audit logging . . . . .	162
Audit trail constraints—using DASD-only logstreams . . . . .	164
Audit trail examples . . . . .	165
Process-level auditing . . . . .	167
Activity-level auditing . . . . .	167
Full auditing . . . . .	168
Using the audit trail utility program, DFHATUP . . . . .	168
Using DFHATUP to read audit logs . . . . .	168
Sample job stream to run the DFHATUP program . . . . .	168
Example output from the DFHATUP utility . . . . .	171

<b>Chapter 17. Examining BTS repository records . . . . . 181</b>	
Introduction to the repository utility program, DFHBARUP . . . . .	181
The repository utility program, DFHBARUP . . . . .	181
Using DFHBARUP . . . . .	181
Sample job stream to run the DFHBARUP program . . . . .	182
Example output from the DFHBARUP utility . . . . .	184

<b>Chapter 18. BTS messages, trace, and dump . . . . . 193</b>	
BTS messages . . . . .	193
Setting trace levels for BTS . . . . .	193
Defining tracing levels at system initialization . . . . .	193
Defining tracing levels when CICS is running . . . . .	194
The BTS trace points . . . . .	194
Extracting BTS information from a CICS system dump . . . . .	195

---

## **Chapter 19. Tuning BTS performance . . . . . 197**

---

## **Part 5. BTS Application Programming Reference . . . . . 199**

<b>Chapter 20. Overview of BTS API commands</b>	<b>201</b>
Process- and activity-related commands	201
Creating, activating, and terminating processes and activities	201
Retrieving information about activities	202
Relating UOWs and activities	202
Container commands	202
Event-related commands	203
Terminology	203
The event-related commands	204
Browsing and inquiry commands	205
Browsing commands	206
Inquiry commands	206
Tokens and identifiers	207
Commands which take identifiers returned by browse operations	207
Browsing examples	208
System events	211
<b>Chapter 21. BTS application programming commands</b>	<b>213</b>
<hr/>	
<b>Part 6. BTS System Programming Reference</b>	<b>215</b>
<hr/>	
<b>Chapter 22. Overview of BTS system programming commands</b>	<b>217</b>
Control commands	217
Inquiry command	217

<b>Chapter 23. BTS system programming commands</b>	<b>219</b>
CREATE PROCESSTYPE	219
DISCARD PROCESSTYPE	221
INQUIRE PROCESSTYPE	222
Browsing	224
SET PROCESSTYPE	225
<hr/>	
<b>Part 7. Appendixes</b>	<b>227</b>
<hr/>	
<b>Appendix. The BTS sample application</b>	<b>229</b>
Running the sample application from the WWW	230
<b>Notices</b>	<b>231</b>
Programming interface information	232
Trademarks	232
<b>Bibliography</b>	<b>233</b>
CICS books for CICS Transaction Server for z/OS	233
CICSplex SM books for CICS Transaction Server for z/OS	234
Other CICS publications	234
<b>Accessibility</b>	<b>235</b>
<b>Index</b>	<b>237</b>





---

## 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.*



---

## 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 1.

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 each command’s keyword 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 <i>must</i> code.
	A set of alternatives—one of which you <i>may</i> code.

Symbol	Action
	A set of alternatives—any of which you may code.
	Alternatives where <b>A</b> is the default.
<p><b>Name:</b></p>	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 should 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 1

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 3.2*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1*
- *CICS Transaction Server for z/OS Upgrading from CICS TS Version 2.3*



---

## Part 1. Overview of BTS

This part of the manual contains introductory information about CICS business transaction services (BTS).

*Table 1. Concepts road map*

<b>If you want to...</b>	<b>Refer to...</b>
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





---

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

CICS has always provided a robust transaction processing environment. For example, it:

- Allows you to create transactions with the ACID properties of atomicity, consistency, isolation, and durability.
- Allows transactions to continue to run under all sorts of conditions.

In recent years, much emphasis has been placed on continuous operation and high availability of CICS. Use of sophisticated technologies, such as the Parallel Sysplex<sup>®</sup>, 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 (BTS)** bring a similar sophistication to the CICS application programming interface (API), making it better able to model complex business transactions.

---

### Business transactions and CICS transactions

This section examines the ways in which business transactions have traditionally been modeled by CICS transactions, and some of the shortcomings of the traditional approach.

#### Business transactions

A **business transaction** is a self-contained business deal—for example, buying a theatre ticket. Some business transactions—for example, buying a newspaper—are simple and short-lived. However, many are not. Many involve multiple actions that take place over an extended period. For example, selling a vacation may 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 may be taken days, weeks, or even months after the transaction was started—arranging foreign currency, in this example.
- Some of the actions may 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's not possible to book a hotel. In this case, the action must be retried. Or the customer might fail to meet his final payment; this 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 and the unit of work (UOW). Typically, a UOW is short-lived, because it is undesirable for it to hold locks for long periods, thus causing other UOWs to wait on resources and possibly abend. A CICS transaction consists of one or more UOWs. It provides the environment in which its associated UOWs will run—for example, the transid, program name, and userid. Typically, like the UOWs of which it consists, a CICS transaction is short-lived, because the aim should be 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 problems

Traditionally, application programmers have modeled business transactions using the basic CICS building blocks, transactions and units of work. However, there are problems. Here are some of them:

### Application design

Typically, the individual actions that make up a complex business transaction are mapped on to CICS transactions. Usually, 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 reinvoked 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 may or may not be planned). To survive restarts, state data relating to the business transaction's flow of control 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.

### **The solution**

CICS business transaction services extend the CICS API and provides support services that make it easier to model complex business transactions. How it does this is the subject of Chapter 2, "What are CICS business transaction services?," on page 7.



---

## Chapter 2. What are CICS business transaction services?

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

CICS business transaction services consist of an application programming interface and support services that simplify the development of business transactions. As the vacation example in the previous chapter illustrates, business transactions are often made up of multiple actions, that may be spread over hours, days, or even months.

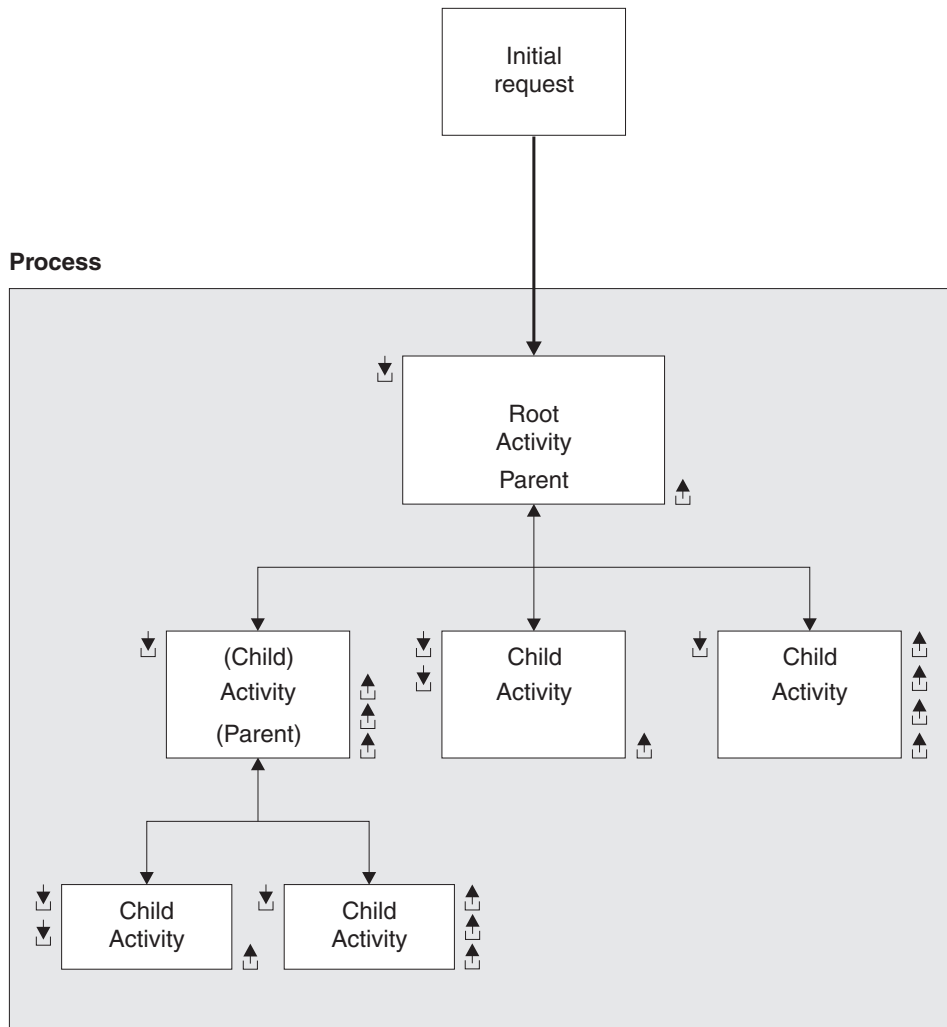
CICS business transaction services allow you to control the execution of complex business transactions. Using BTS, each action that makes up the business transaction is implemented as one or more CICS transactions, as in the traditional approach. However, a top-level program is used to control the overall 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 brings a number of benefits:

- Management and control is 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 simplifies the development of such transactions and makes it easier to reuse them.

---

### What is a BTS application?

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 shall refer to an application that uses the CICS business transaction services API as “a BTS application”.)



**Key:** ↓ Input data-container    ↑ Output data-container

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 may 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 current 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**

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 reinvoked when one of its child activities completes. It determines which event caused it to be reinvoked—that is, the completion of the activity that it started earlier. It retrieves, from the completed activity's output data-containers, any return data that the completed activity has placed there.
5. Steps 3 on page 9 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

The CICS Web Interface allows Internet users to 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 browser invokes a CICS transaction that does some setup work, returns a page of HTML to the 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 particularly useful if the business transaction is long-lived.

---

## Support for existing code in BTS

BTS supports the 3270 bridge function, which is described in the *CICS External Interfaces Guide*. This means that BTS applications can be integrated with, and make use of, existing 3270-based applications.

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. 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 may execute. For example, the activities that constitute a single process may execute on several regions.

### Dynamic routing of BTS activities

In a *BTS-set*, the CICS transactions that implement your BTS activities can be routed dynamically across the participating regions. When an event is signalled, an activity is activated in the most appropriate region in the *BTS-set*, based on one or more of the following:

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

You can control the dynamic routing of your BTS activities by either of the following means:

1. Using the CICSplex<sup>®</sup> 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 balancing of the transactions that implement BTS activities.
2. Writing a CICS **distributed routing program**.

Dynamic routing of BTS activities is described in Chapter 14, “Using BTS in a sysplex,” on page 141.

### Audit trails

You can create an audit trail for the BTS processes and activities that run in your CICS regions. Doing so allows you to, 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 may 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. This means, for example, that you can request information about a business transaction's use of resources 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 19, “Tuning BTS performance,” on page 197.

---

## **The BTS sample application**

As well as the fragments of example code in this book, CICS supplies a sample BTS application. The sample is a basic sales application, consisting of order, credit check, stock check, delivery note, invoice, and payment-reminder activities. See “The BTS sample application,” on page 229.

---

## **Requirements for BTS**

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

---

## 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*

<b>If you want to...</b>	<b>Refer to...</b>
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, "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



---

## 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 20
- “BTS events” on page 21.

---

### BTS activities and processes

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 re-instantiated 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

When a program defines a process, it gives the process a name (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 will use to reference the child. This user-assigned name, which can be up to 16 characters long, only needs to be unique within the 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 by means of its identifier.

## Activation sequences

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

Each activation is “triggered” by a BTS *event*, and consists of a single transaction. An activity's first activation 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 63. ) When the last activation ends, the *activity completion event* is “fired”, which may, in turn, trigger another activity's activation. 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 may 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.

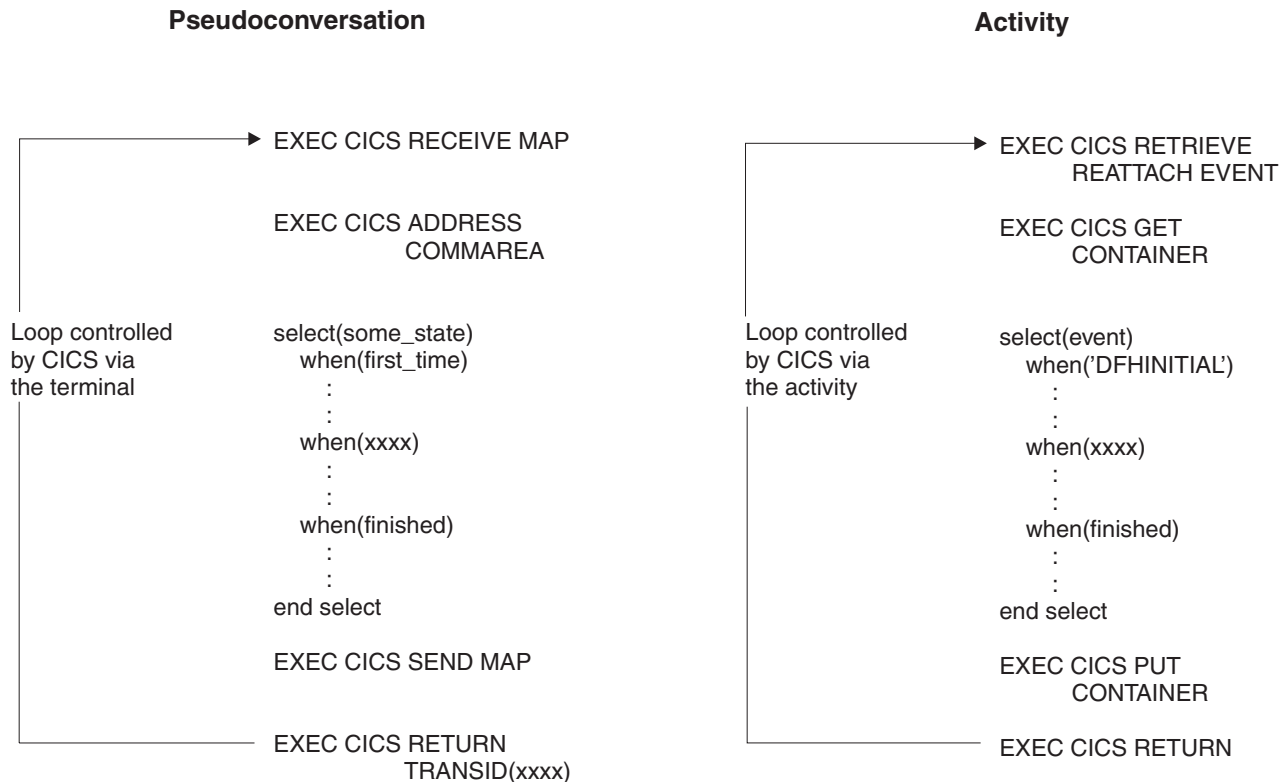


Figure 3. Comparison between a terminal-related pseudoconversation and a BTS activity that is activated multiple times

**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

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 (TRANSID or USERID) 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 that of the requestor, and with the transaction attributes (TRANSID and USERID) 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 that of the requestor, and with the transaction attributes (TRANSID and USERID) 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. This is because 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 firing of the child's completion event. (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 will not be 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 destroyed:

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

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

## **Processing modes**

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 activity's descendants are inaccessible. This can happen if, for example, one of the activity's children 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 by means of 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.

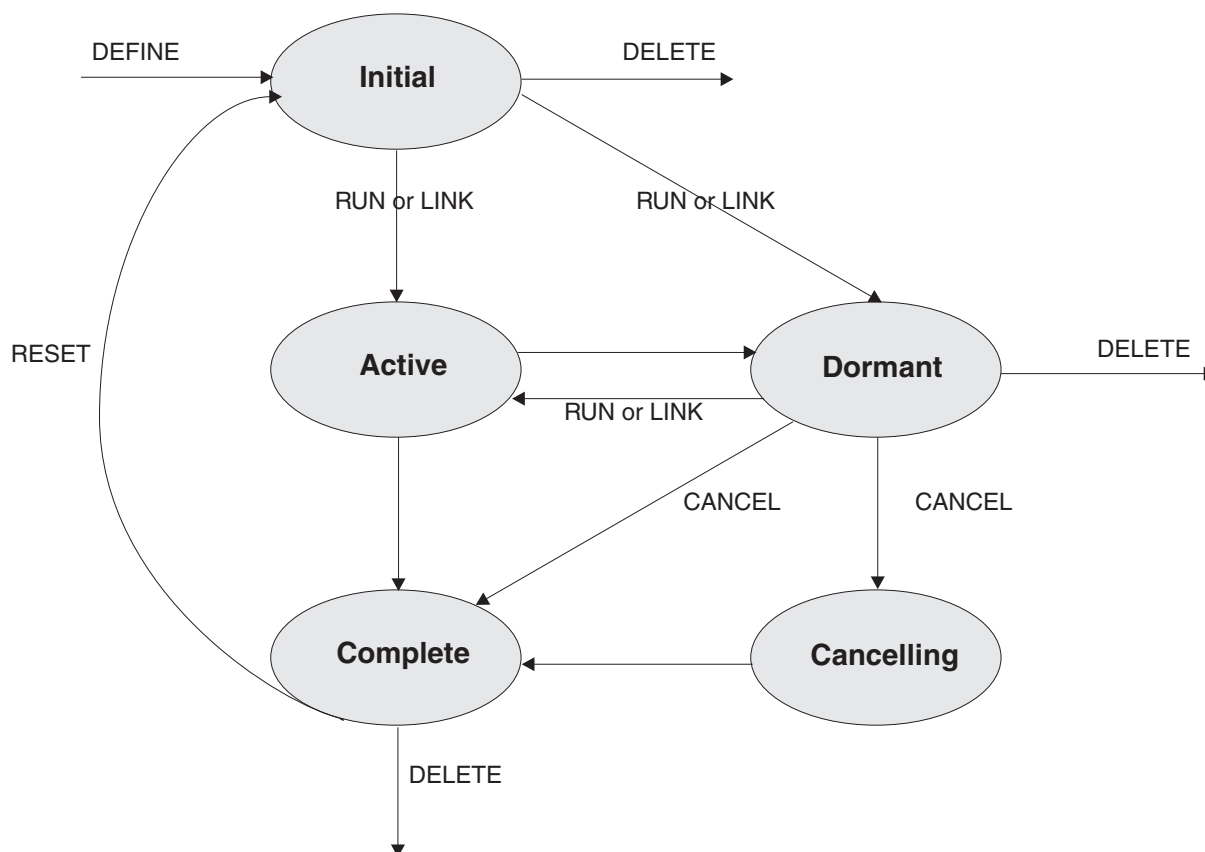


Figure 4. Activity mode transitions. The words in uppercase are the commands which cause the transitions.

To discover the current mode of an activity, use the CHECK ACTIVITY or INQUIRE ACTIVITYID command.

## User syncpoints

A program that is running as the activation of a BTS *process* cannot issue user syncpoints (EXEC CICS SYNCPOINT commands). Some examples of programs that may issue user syncpoints are:

- A “top-level” transaction that defines and runs a BTS process
- A program that runs as the activation of a child activity

- A program executing outside the BTS environment that acquires a process or activity by means of 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.

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”.

An activity's data-containers serve as its working storage. They can be read and updated by the activity itself, by the activity's parent, or by a program that has “acquired” the activity. Because they 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, being written to disk as necessary, and restored at system restart.

Just like an activity, a process may 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 root activity's containers.

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

### Related tasks

“Acquiring processes and activities” on page 61

## Lifetime of data-containers

An activity's data-containers 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 will destroy the activity's containers. It's usually best to allow activities to be deleted automatically by CICS. For child activities, this happens when the activity's parent completes. At this stage, the parent no longer needs access to its children's containers. If the parent is reset and re-run, it will re-create its child activities.

---

## 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 can use a timer to, for example, 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—see “Timer 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

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

---

## BTS events

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. 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 may 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 (see “Composite events” on page 22) can be constructed.

## Input events

**Input events** tell 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 may 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 invoked with DFHINITIAL), it may use the DEFINE INPUT EVENT command to define some input events for which it can be activated subsequently.

**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 63.

## 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 fires.

**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

All the other types of event described in this section (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. BTS system events, on the other hand, are defined by BTS. They are a special kind of input event.

There is only one type of BTS system event—DFHINITIAL. See “System events” on page 211.

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, non-system) 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.

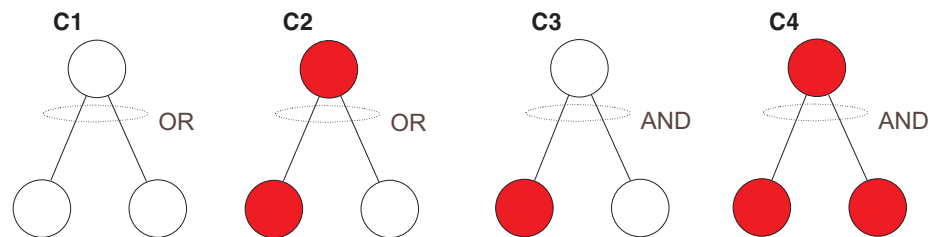


Figure 5. Composite events. An OR composite event fires when any of its sub-events fires. An AND composite event fires when all of its sub-events have fired.

**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 sub-event queue**

The names of sub-events that fire are placed on the composite event's **sub-event queue**—from where they can be retrieved by issuing one or more RETRIEVE SUBEVENT commands. Each composite event has a sub-event queue associated with it. The sub-event queue:

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

Figure 6 on page 24 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.

### Event pool

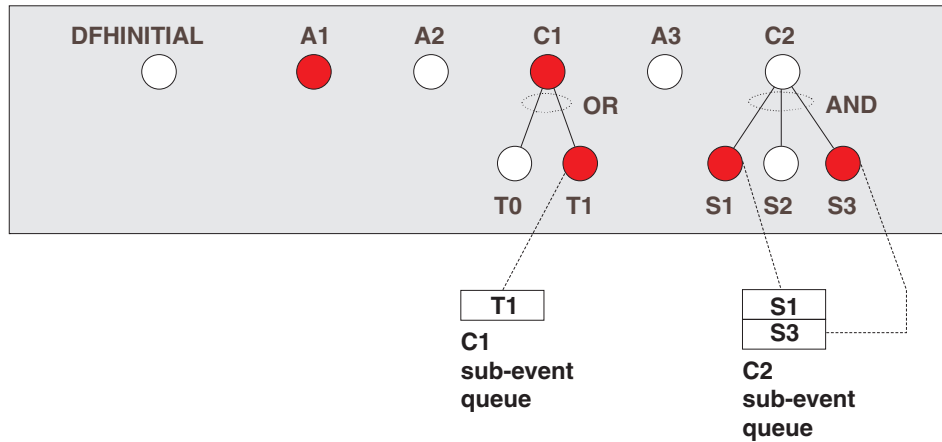


Figure 6. Sub-event queues. The sub-event queue for composite event C1 contains the name T1. The sub-event queue for composite event C2 contains the names S1 and S3.

## Event pools

Events are defined within **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 by means of:
  - DEFINE COMPOSITE EVENT
  - DEFINE INPUT EVENT
  - DEFINE TIMER
  - The EVENT option of the DEFINE ACTIVITY command.
2. 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 described in "Event-related commands" on page 203, except FORCE TIMER, operate on the event pool associated with the *current* activity.

Figure 7 shows an activity's event pool.

### Event pool

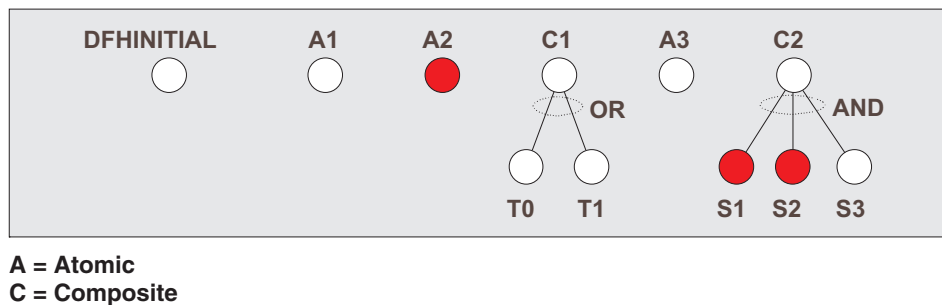


Figure 7. An event pool

## Deleting events

### About this task

You can delete an event (that is, discard both the event and its name). If the event is a sub-event, the value of the composite event will be that of its predicate, after the sub-event has been removed from the predicate's Boolean expression.

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. Note that deleting a composite event does not delete the composite's sub-events.
- 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 activity's parent; 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 by means of `CHECK ACTIVITY` or `DELETE ACTIVITY` commands—in the activity's event pool. 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	<ol style="list-style-type: none"><li>1. <code>CHECK ACTIVITY</code> (if the activity has completed)</li><li>2. <code>DELETE ACTIVITY</code></li></ol>
Composite	<ol style="list-style-type: none"><li>1. <code>DELETE EVENT</code></li><li>2. <code>RETURN ENDACTIVITY</code></li></ol>
Input	<ol style="list-style-type: none"><li>1. <code>DELETE EVENT</code></li><li>2. <code>RETURN ENDACTIVITY</code></li></ol>
System	Cannot be deleted
Timer	<ol style="list-style-type: none"><li>1. <code>CHECK TIMER</code> (if the timer has expired)</li><li>2. <code>DELETE TIMER</code></li><li>3. <code>RETURN ENDACTIVITY</code></li></ol>

Before it can complete normally, an activity must have deleted all the activity completion events in its event pool. (That is, 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 an event (other than a sub-event) 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 may 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 sub-event never directly causes an activity to be reattached—it is the firing of the associated composite event that does so. Therefore, a sub-event can never be a reattachment event.

### Handling reattachment events

#### About this task

When an activity is reattached, it should use the RETRIEVE REATTACH EVENT command to discover the event that caused reattachment. If the event that caused it to be reattached is composite, the activity may also need to issue one or more RETRIEVE SUBEVENT commands to discover the sub-event or sub-events that fired.

At times reattachment may 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 by means of 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 will be 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 on page 27 shows the event pool and reattachment queue for a particular activity. The reattachment queue contains the names A1 and C1.



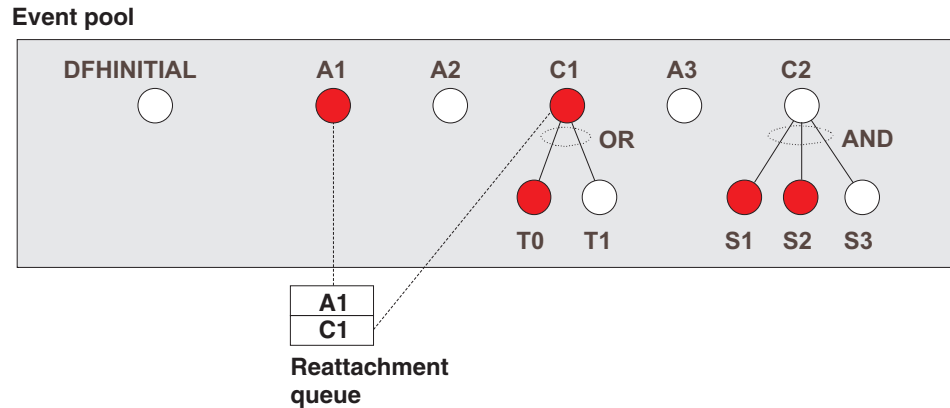


Figure 8. A reattachment queue. The queue contains the names A1 and C1.

**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 made no progress—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 syncpoint does not occur until the activation returns.

### Resetting and deleting reattachment events: About this task

Retrieving an atomic event from the reattachment queue automatically causes the event's fire status to be reset to NOTFIRED.

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.

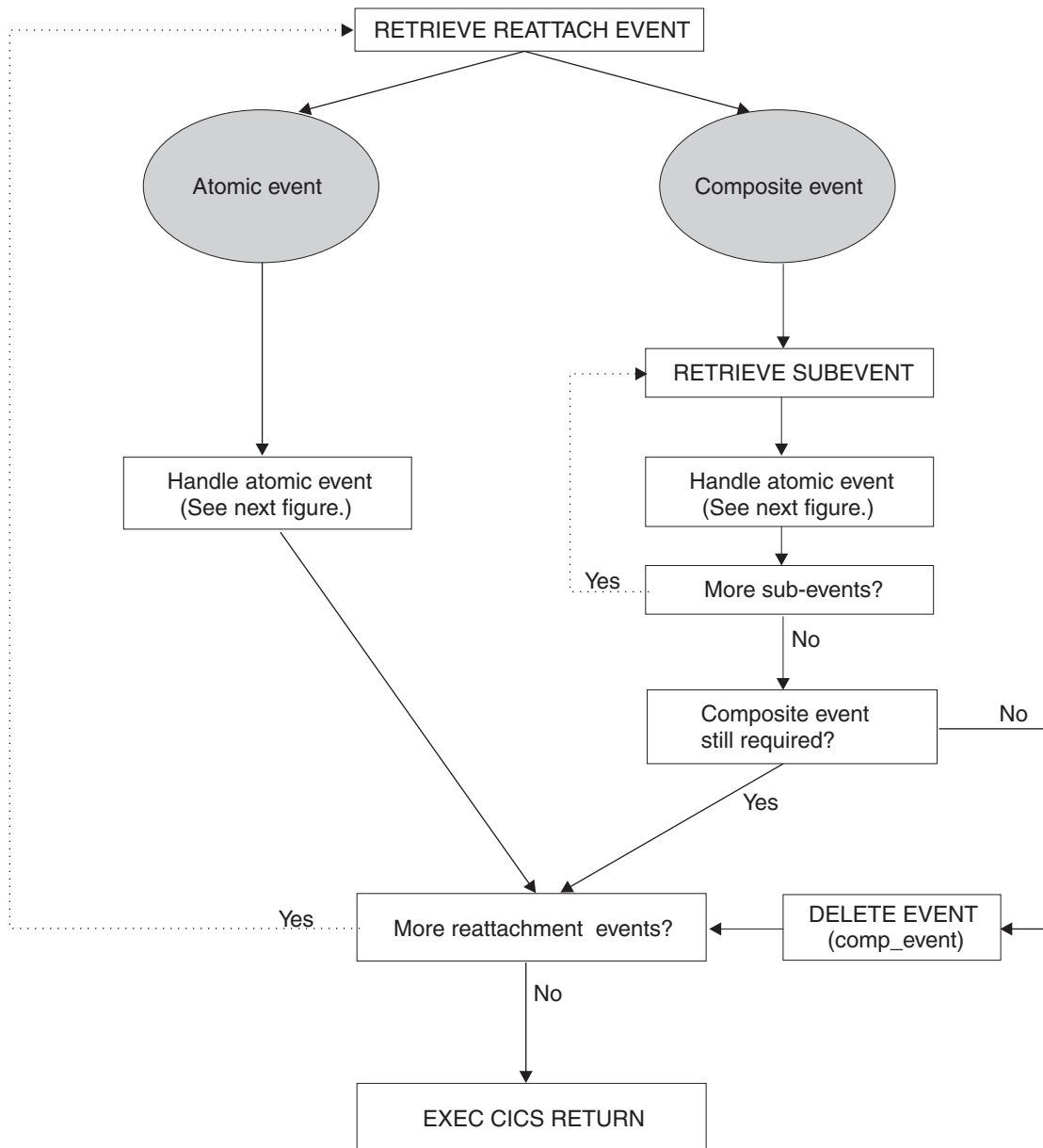


Figure 9. Handling reattachment events. The “Handle atomic event” box is expanded in Figure 10 on page 30. The figure shows multiple reattachment events being handled in a single activation—you may prefer to handle one per activation.

## HANDLE ATOMIC EVENT

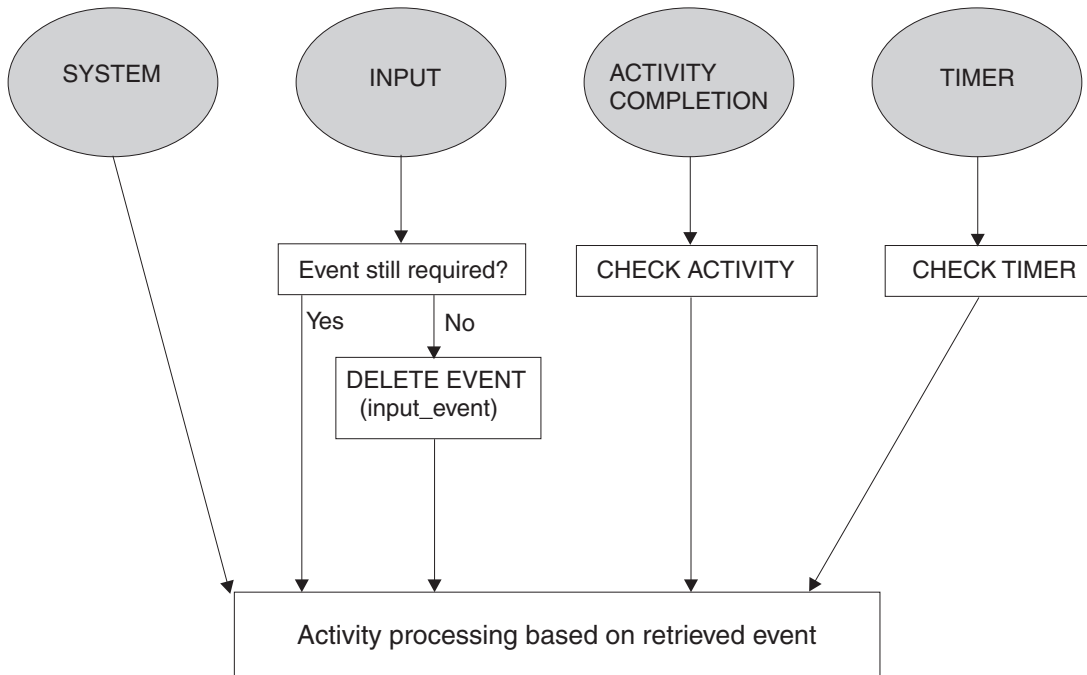


Figure 10. Handling atomic events

### Note:

- 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.)
- 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):

- 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:**

**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 is not to 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 activity's event pool**

The activity completes normally.

**If there are user events (fired or unfired) in the activity's event pool**

- 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

The Sale example implements a *Sale* business transaction that is made up of four basic actions:

- Order entry
- Delivery
- Invoice
- Payment.

A Sale business transaction is started by a terminal-user selecting the Sale option from a menu of business transactions. This 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

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

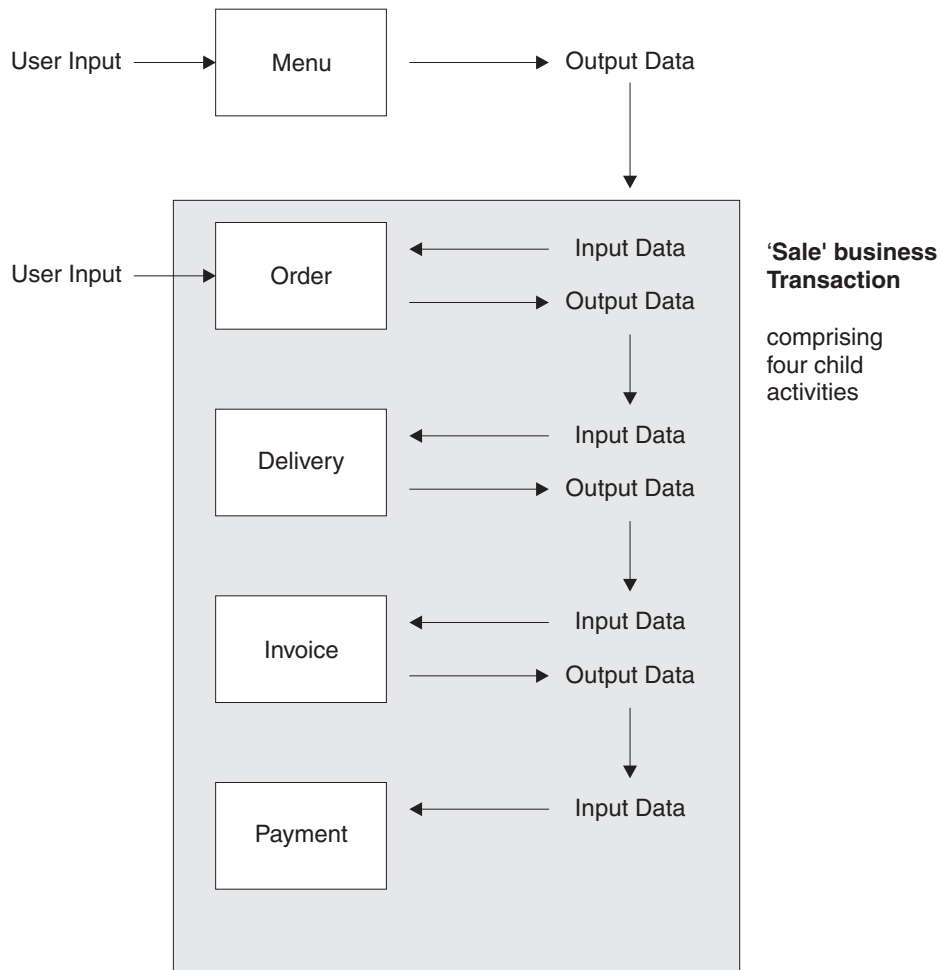


Figure 11. Data flow in the Sale example application. (The root activity is not shown.)

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

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

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.

Figure 12 on page 36 shows, in COBOL pseudocode, how DFH0SAL1 creates and starts an instance of the Sale business transaction.

```

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. 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 that 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 that 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 that of its requestor, the start and finish of the activation are related to the requestor's syncpoints. 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 will be accepted. If not, it will be rejected. “Accepted” means committed—this instance of the Sale transaction will be ready to start its next activity. “Rejected” means rolled back—this instance of the Sale transaction will no longer exist.

---

## The root activity

The SAL001 program starts a new instance of the Sale business transaction by starting the SAL002 program, running under the transid 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 (somewhat 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('Delivery-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 tells the activity that it should 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 will run 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 requestor's transaction identifier, 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 Delivery activity's completion event 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 sub-event, that is in its event pool. (An activity's event pool contains events that have



been defined to the activity, plus the DFHINITIAL system event.) Thus, the SAL002 root activity will be reattached when the Delivery activity's completion event (*Delivery-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 will always be executed *synchronously* with the parent, the EVENT option is not often used. Normally, the firing of a synchronous activity's completion event does not cause the parent to be reattached, because the event is deleted (by a CHECK ACTIVITY command) during the parent's current activation. 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 Order activity's output data-container to the Delivery activity's input data-container (both of which are named *Order*). Note that 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 will be 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 will be executed asynchronously with SAL002, and will start 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 syncpoint 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 syncpoint, the menu of business transactions is redisplayed on the user's terminal, ready for further selection. The remaining activities will be

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 *Delivery-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 Invoice activity's parameters, 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 indicates to CICS that the root activity's processing 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

### About this task

This section illustrates how to transfer data between a parent and a child activity. It uses the Sale application's Delivery activity as an example.

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('Delivery-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 Order activity's output data-container, 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
- “Retrying failed activities” on page 52

---

## Checking the response from a synchronous activity

### About this task

Figure 18 shows the Sale application's Order activity 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 command's processing. 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 may result from processing the activity itself.

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 by means of 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.

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

```
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.
.
```

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 by means of 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 activity's parent 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 "Acquiring an activity" on page 70.

---

## Checking the response from an asynchronous activity

### About this task

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

### About this task

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. For example:

```
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
```

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

**Note:** 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.

---

## Retrying failed activities

### About this task

If a child activity fails, it may be appropriate to retry it. If it has not already done so, the parent should issue a CHECK ACTIVITY command to check the current completion status of the child activity.

### Procedure

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

---

## Chapter 6. Running parallel BTS activities

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

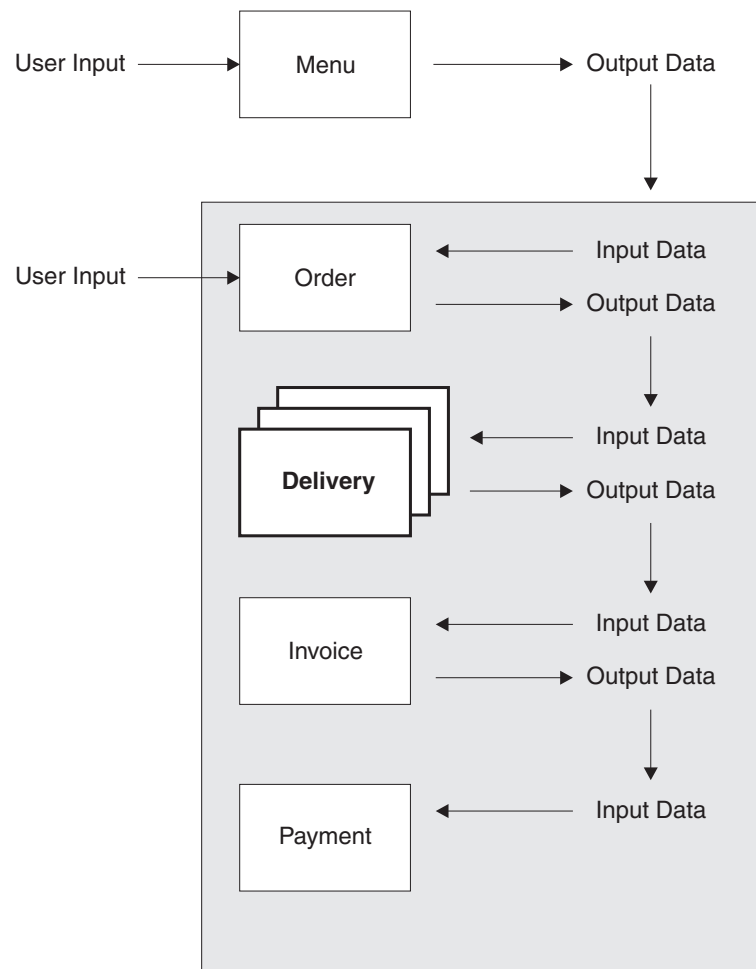
---

### An example of parallel 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 of the items have been delivered.

#### Data flow

Figure 19 shows data flows in the Sale example application when parallel activities are included.



*Figure 19. Data flow for parallel 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.

- 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.

## The root activity

Figure 20 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 20. 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 21. 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('DELO01')
      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 22. 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('Delivery-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 23. 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 24. 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 between 1 and 20 items in an order. Having first defined a composite event (*Delivery-Complete*), SAL002 requests a delivery activity to be run for each item ordered:

```

EXEC CICS DEFINE COMPOSITE EVENT('Delivery-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 will 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('DELO01')
      EVENT(Work-Event)
      RESP(data-area) RESP2(data-area) END-EXEC
.
EXEC CICS ADD SUBEVENT(Work-Event) EVENT('Delivery-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-through-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 will *not* cause SAL002 to be reattached—because the delivery activities' completion events have been specified as sub-events of the composite event *Delivry-Complete*. Instead, SAL002 will be reattached when *Delivry-Complete* fires. Because *Delivry-Complete* uses the AND Boolean operator, it will fire 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 *Delivery-Complete* composite event, it uses a succession of EXEC CICS RETRIEVE SUBEVENT commands to retrieve, in turn, each sub-event on the composite event's sub-event queue—that is, each sub-event whose firing was instrumental in the firing of the composite event. These sub-events are, of course, the completion events for each of the delivery activities. The number of each sub-event (contained in the *Event-Count* field of *Work-Event*) is used to identify the particular delivery activity for which the sub-event is the completion event.
- When all the sub-events have been retrieved, SAL002 deletes the composite event *Delivery-Complete*. This 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.

Note that deleting a composite event does *not* delete any associated sub-events. In the example in this chapter, the sub-events 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

This chapter describes ways in which BTS processes and activities can interact with the world outside the BTS environment. It contains:

- “Introduction”
- “Using client/server processing” on page 63
- “Acquiring an activity” on page 70
- “Transferring data to asynchronous activations” on page 83.

---

### Introduction

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

In practice, many business transactions require some sort of interaction with things external to themselves. For example, most business transactions include activities that require human involvement. (Such activities are known as **user-related activities**. They 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.

To permit interaction with the outside world, *BTS allows a program executing outside a process to acquire access to an activity within the process*. This means, for example, that 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.

## Acquiring processes and activities

### About this task

Before a program executing outside a process can activate an activity within the process, it must acquire access to the activity. Acquiring an activity enables the program to:

- Read and write to the activity’s containers

- 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.

To gain access to an activity from outside the process that contains it, you use the ACQUIRE command. An activity that a program accesses by means of 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 the process's root activity. (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 by means of *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*.

### **Some rules**

1. A program can acquire only one activity within the same unit of work. The activity remains acquired until the next syncpoint. 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. An acquired activity's process is accessible in the same way as the activity itself can access it. Thus, if the acquired activity is a descendant activity:
  - Its process's containers may be read but not updated.
  - The process may 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:

- Its process's containers may be both read and updated.
- The process may 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. 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. At this point, the server process is maintained only by BTS.

### A client/server example

The client/server example in this section shows:

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 may be reinvoked; 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 reinvoked, 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 should not be reinvoked.

#### The client program

Figure 25 on page 64 shows, in COBOL pseudocode, the example client program, PRG001.

```

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 25. 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 26. 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 27. 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 an 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 syncpoint 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 process's name and process-type:

```
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

Figure 28 shows, in COBOL pseudocode, the example server program, SRV001.

```
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 28. 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 29. 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 30. 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. SRV001's housekeeping includes defining two input events for which it could subsequently be reinvoked:

```

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 will fire when *either* of the two input events fires; SRV001 will be 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 sub-event *SRV-WORK*.)

Eventually, the RETRIEVE SUBEVENT command returns the sub-event *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.

---

## Acquiring an activity

### About this task

Imagine a particular activity's processing to be organized in two activations. The first activation sets up the environment. The second activation is started when a defined external interaction occurs.

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 will start 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 will start 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 will start 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. Re-activate the activity, and tell it why it is being activated—by issuing a RUN ACQACTIVITY command that specifies the input event.

Figure 31 on page 71 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

re-activates it.

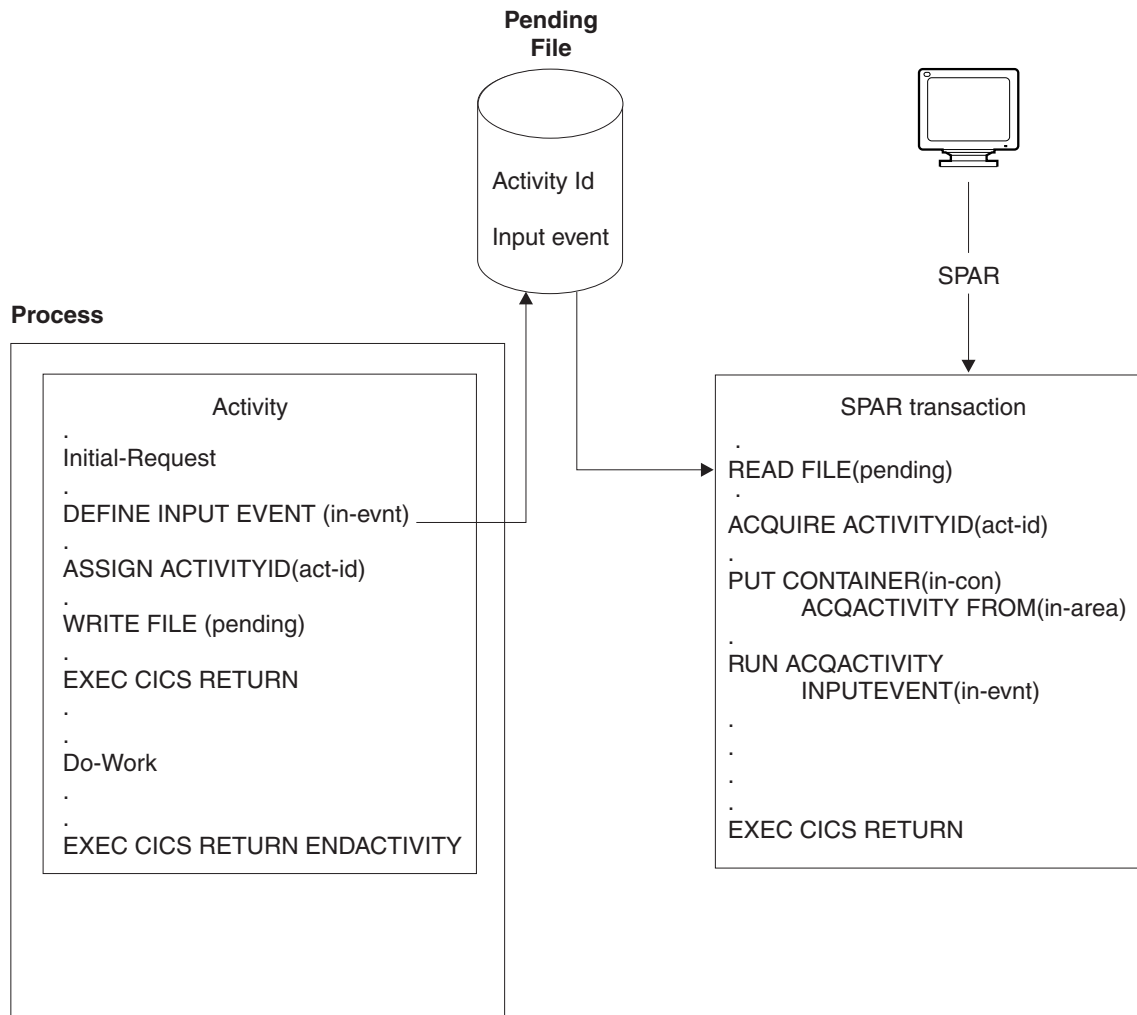


Figure 31. *Acquiring an activity.* On its initial activation, the activity sets up the environment and returns without completing. Some time later, the SPAR transaction is started from a terminal; it retrieves the activity identifier and input event, issues an ACQUIRE ACTIVITYID command to gain access to the activity, and re-activates it.

## A user-related example

The Sale example application described in Chapter 4, “The Sale example application,” on page 33 assumed that none of its later activities required human involvement. (The only child activity to require human involvement was the first (Order), and this was 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 has notified the Sale transaction that the delivery has taken place. In addition, the Payment activity requires user input.

### Data flow

Figure 32 on page 72 shows data flows in the Sale example application when the user actions described above are included.

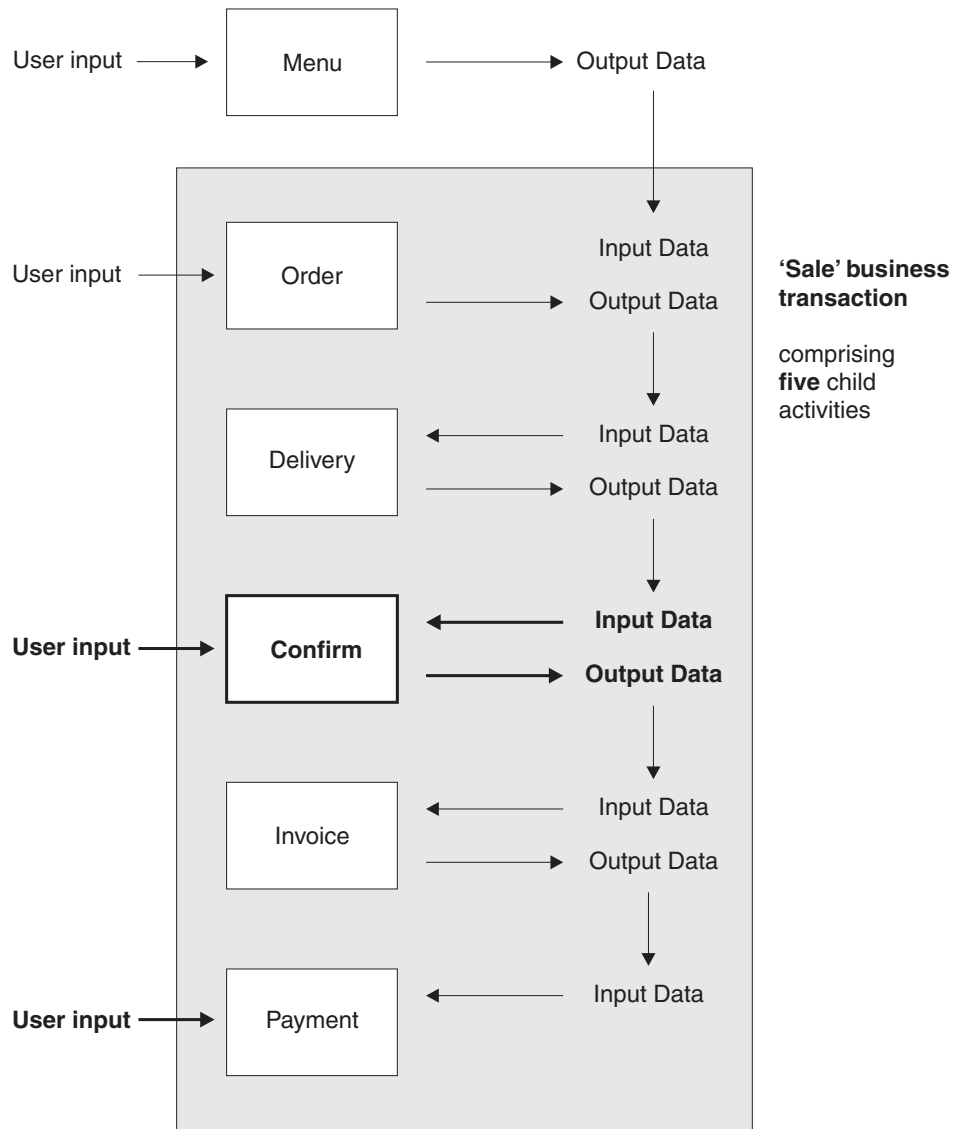


Figure 32. Data flow in the Sale example application, showing user-related activities. (The root activity is not shown.) Changes from the basic Sale example described in Chapter 4, "The Sale example application," on page 33 are shown in bold.

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

Figure 33 on page 73 shows, in COBOL pseudocode, the Sale root activity, with modifications for user-related activities. The changes are in 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 'Delivery-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 33. 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 34. 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 35. 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 36. 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 37. 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 user input. The following pseudocode creates the Confirm activity:

```

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

```

Because the Confirm activity will be executed asynchronously with the root activity, the EVENT option of DEFINE ACTIVITY is used to name the activity's completion event as *Delivery-Confirmd*. CICS will reattach SAL002 when this event fires—that is, when the Confirm activity completes.

SAL002 places the input data 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 will reattach the root activity only when the Confirm activity has completed.

### Implementation of a user-related activity

The Confirm activity is used to notify the Sale business transaction that actual delivery has taken place. Figure 38 on page 78 shows, in COBOL pseudocode, how program CON001 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 38. 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 39. 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 may subsequently 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 the user is ready to confirm delivery, he or she invokes the USRX user-written transaction, which starts the USRCON program. USRCON executes outside the BTS environment—it is not part of the SAL001 process that contains the Confirm activity. Figure 40 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 40. Pseudocode for the USRCON program, that implements the USRX transaction

First, USRCON sends a map to the user's screen and requests a unique reference. This must be the same as the key used by the CON001 program. It might be the customer reference or account number that has been used throughout to identify this instance of the Sale business transaction. However, it may 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 Confirm activity's containers. USRCON creates a new 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 USRCON has acquired.

Finally, USRCON re-activates the Confirm activity, 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. Note that 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 Confirm activity's completion event. CICS deletes a completed activity's completion event only after the execution of a CHECK ACTIVITY command issued by the activity's parent.

**The second activation of the Confirm activity:** The Confirm activity is activated for a second, and final, time due to 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 respectively.
3. Stores the updated delivery details into the Confirm activity's output data-container (*Confirm*).



See Figure 38 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 Confirm activity's completion event (*Delivery-Confirmed*) is fired.

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

---

## Transferring data to asynchronous activations

### 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 may be used for both input and output data. If the activity will be activated only once, this presents no problems. Perhaps separate containers are used, one for input and one for output data. Again, if the activity will be activated only once, this presents no problems. However, if the activity may be activated, asynchronously, multiple times, you must take care that the contents of containers are not over-written inadvertently. You should 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 the normal case, the activity is dormant—awaiting the activation and ready to perform its function. The activation occurs almost immediately, the activity program executes and places any results in a container. In a client/server application, the activity may then be left dormant, ready for the next request. If the activation is triggered by an external interaction, it's likely that the activity will complete; the firing of its completion event causes its parent to be reactivated.

However, you should take account of the fact that, when the RUN ASYNCHRONOUS command is executed, the target activity may not be dormant, waiting for work—it may 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 will not happen immediately. Thus, in a client/server application, it's 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 would be 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

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 usually 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 may need to be reversed, or possibly modified. Similarly, if application logic determines that the business transaction should be terminated, changes made by activities that have already completed may need to be reversed.

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

---

### Implementing compensation

#### About this task

Compensation is the act of modifying (“compensating for”) the effects of a completed activity. How compensation is implemented is decided by the designer of the business transaction. 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 activity's parent. 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. Re-run the activity.

To do this, 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. (Note that 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.

This is the more straightforward method. 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 that used for the execution of the activity that is compensated.

The compensation example in this section uses this method.

---

## **A compensation example**

In this section, 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**

Figure 41 on page 87 shows, in schematic form, the Sale example application when compensation actions are included.

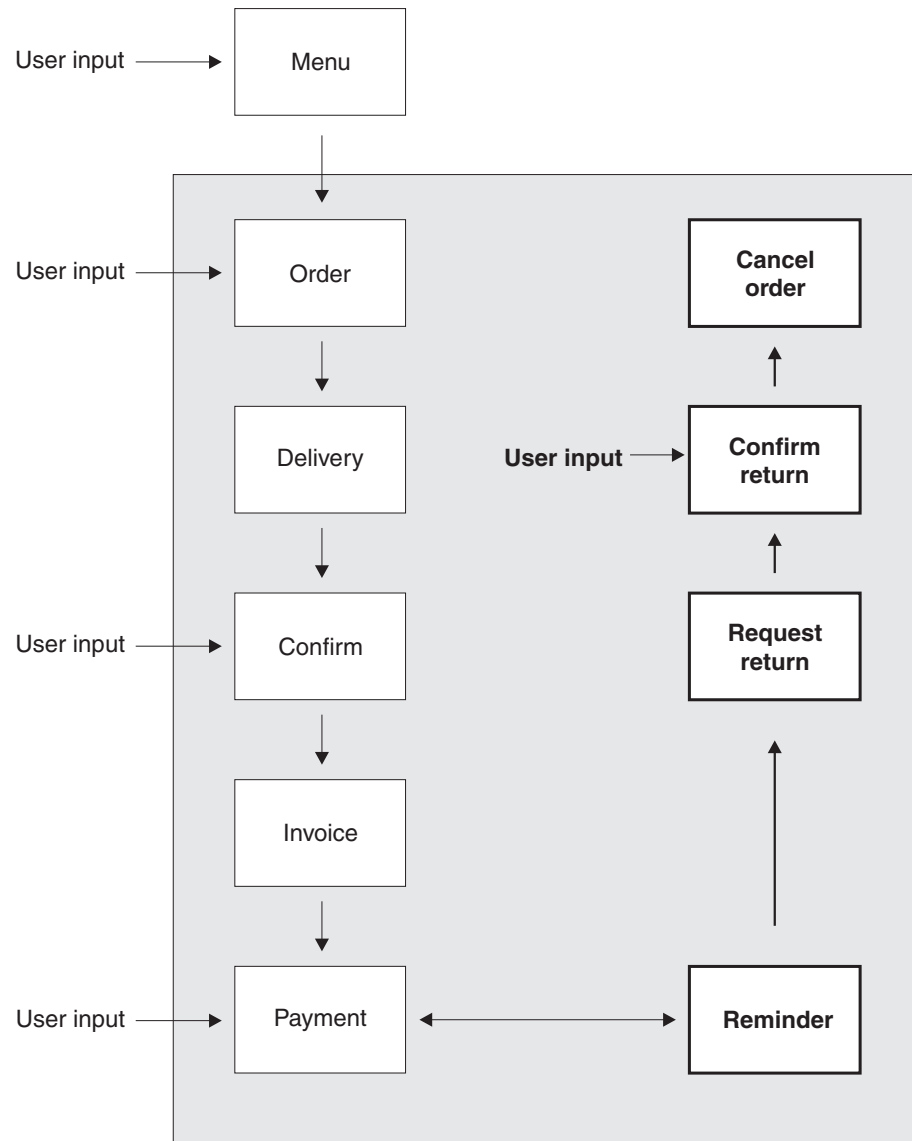


Figure 41. Process flow with compensation actions included. (The root activity is not shown.) Changes from the user-related example described in Chapter 7, “Interacting with BTS processes and activities,” on page 61 are shown in bold.

- 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

Figure 42 shows, in COBOL pseudocode, the Sale root activity, modified to include compensation actions. The changes are in 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 Remindr-Expired               value 'Remindr-Expired'.
   88 Remindr-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 42. 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 43. 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 44. 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 45. 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 46. 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 47. 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 48. 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 49. 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

### About this task

When an activity completes, any updates 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 which are part of other business applications. 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 may 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 may 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 may need to include logic to provide logical record locking. This “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 may 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, 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

This section describes how you can use BTS's support for the 3270 bridge to integrate existing transactions into BTS applications.

The 3270 bridge is described in the *CICS External Interfaces Guide*.

---

### Running a 3270 transaction from BTS

#### About this task

BTS supports the 3270 bridge function. This means that BTS applications can be integrated with, and make use of, existing 3270-based applications.

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

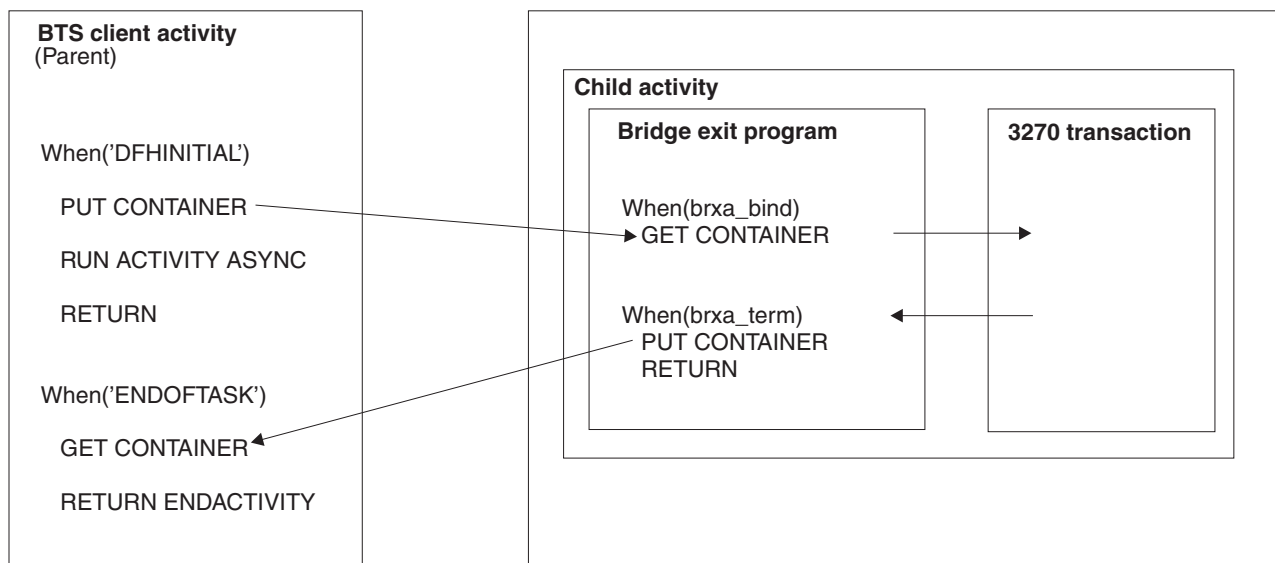


Figure 50. Running a 3270-based transaction as a BTS activity

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.
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 child activity's completion event 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
<pre> 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 </pre>	<pre> 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 </pre>

Note that 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

The basic mechanism described in “Running a 3270 transaction from BTS” on page 99 assumed 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. For example, you might want to run 3270 transactions that:

1. Output intermediate messages
2. Are conversational in design
3. Are pseudoconversational.

### Intermediate output messages

For a non-conversational 3270 transaction, the bridge exit program could be called to write an intermediate message for either of two reasons:

1. The 3270 transaction has specified WAIT on the EXEC CICS SEND command.
2. 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 its storage—including that 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

This section describes how to run a conversational 3270 transaction.

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'll refer to such child activities as “conversational activities”.) Figure 51 illustrates this approach.

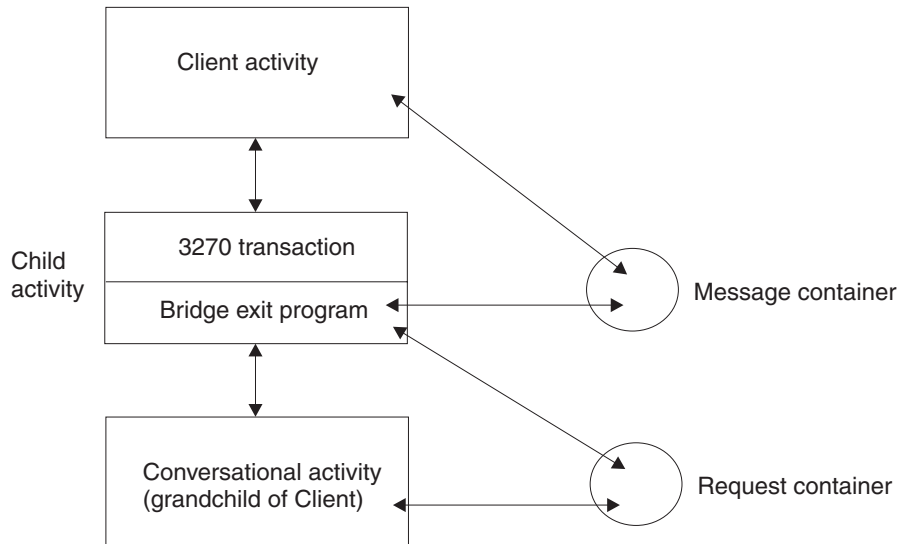


Figure 51. Running a 3270 conversational transaction as a BTS activity. The bridge exit program creates a child activity to deal with each map sent by the 3270 transaction.

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's important that intermediate messages should be 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
<pre> 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 . </pre>	<pre> 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 </pre>

Note that 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

This section describes 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 client's responsibility to check the appropriate field in the output message and to start the next transaction.

The following contains example 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
<pre> 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       FACILITYTKN(8-byte token)     RESP(data-area) RESP2(data-area) END-EXEC     .     EXEC CICS RETURN END-EXEC   End-If. </pre>	<pre> 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 </pre>

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 will have been deleted from the client's event pool 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 FACILITYTOKEN 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, it is essential that all its constituent transactions run in the same CICS region. If one of the transactions is routed to a different region, an ABRH abend occurs.

One way to ensure that all the transactions execute 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 (though, of course, it must be a CICS TS OS/390, Version 1 Release 3 or later 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. This 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 is the preferred option.

---

## Things to consider

This section discusses things that you need to consider when using the 3270 bridge from BTS.

### Using timers

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 hasn't started, or a SET TASK PURGE command if it has.

### Abend processing

If the 3270 transaction ends abnormally, an abend call is made to the bridge exit. Note that 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 a non-recoverable 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
<pre> 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 </pre>	<pre> 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 </pre>

Note that 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

This part of the manual tells you how to define and control the CICS business transaction services environment.

*Table 10. Administration road map*

If you want to...	Refer to...
<ul style="list-style-type: none"><li>• Define BTS data sets to CICS</li><li>• Specify the program used to route BTS activities dynamically around a BTS-set</li></ul>	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 125
Learn how to operate BTS in a sysplex	Chapter 14, "Using BTS in a sysplex," on page 141



---

## Chapter 10. System definition for BTS

This chapter contains:

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

---

### Defining BTS data sets

#### About this task

You need to define two types of BTS data set:

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

Use the IDCAMS program to define your BTS data sets to MVS™.

#### Repository data sets

When a process is not executing 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 doing this 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 don't 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 141.

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.

#### **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, will be 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, 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 52 shows example JCL for defining a BTS repository data set. Note that the 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 ('CICSTS41.CICS.BTS') PURGE CLUSTER
DEFINE CLUSTER (
    NAME( CICSTS41.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( CICSTS41.CICS.BTS.INDEX ) -
)
DATA
(
    NAME( CICSTS41.CICS.BTS.DATA ) -
)
/*
//
```

Figure 52. 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 is used to store 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 53 on page 114 shows example JCL for defining a BTS local request queue data set.

```

//SMITHGOT JOB (WINVMC,SMITH),CLASS=E,USER=username
//IJMLRQ EXEC PGM=IDCAMS,REGION=6144K
//SYSPRINT DD SYSOUT=A
//AMSDUMP DD SYSOUT=A
//SYSIN DD *
  DEFINE CLUSTER (
    NAME( CICSTS41.CICS.LRQ ) -
    LOG(UNDO) -
    CYL(2,1) -
    VOLUME (SYSDAV) -
    KEYS( 40 0 ) -
    INDEXED -
    RECORDSIZE( 2024 2024 ) -
    FREESPACE( 0 10 ) -
    SHAREOPTIONS( 2 3 ) -
  )
DATA (
  NAME( CICSTS41.CICS.LRQ.DATA ) -
  CISZ(2048) -
)
INDEX (
  NAME( CICSTS41.CICS.LRQ.INDEX ) -
)
/*
//

```

Figure 53. 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.

## Sizing and maintaining the LRQ data set

### About this task

**Sizing:** The space you allocate for the LRQ data set depends on how often you plan to trim the file (see “Maintenance”) 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 2KB in length.

You are recommended to 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. And even after 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 will never be reused because they represent time in the past, and their control interval space can only be recovered by periodic maintenance of the data set (see below).

**Maintenance:** 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, unless the LRQ data set is regularly maintained, it may become very large.

You can use an AMS command such as REPRO to shrink the data set—that is, to remove the records that have been logically-deleted by VSAM. Before doing this,



you must quiesce CICS. (Disabling the LRQ file definition is not sufficient—if you do only this, activations may 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 within the sysplex will be uninterrupted.

When reorganizing the data set, consider the percentage of control areas specified on the FREESPACE parameter on the cluster definition. If there are a large number of 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 may occur. If the majority of new timer events are added for times later than existing events, the freespace percentage for control areas may be set to 0.

---

## Naming the routing program

If you are using BTS in a sysplex (see Chapter 14, “Using BTS in a sysplex,” on page 141), you must name the routing program that will be used to dynamically route BTS activities around the BTS-set. To do this, you use the DSRTPGM system initialization parameter.

Table 11. The DSRTPGM system initialization parameter

DFHSIT	[TYPE={ <u>CSECT</u>   DSECT}]
	:
	:
	[,DSRTPGM={ <u>NONE</u>   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 142.
- Eligible non-terminal-related EXEC CICS START requests.  
For information about which non-terminal-related START requests are eligible for dynamic routing, see 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.

---

## Chapter 11. Resource definition for BTS

Most BTS resources (processes, activities, events, and containers) are defined at run time, using BTS API commands. The only BTS resources that must be defined on the CICS system definition file (CSD) are:

### Process-types

See *CEDA DEFINE PROCESSTYPE*.

**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 may want to do this 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*.

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*.

For information about how to define repository files to the CSD, see *Defining repository files to the CSD*.

### 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

### About this task

There is a default FILE definition, DFHLRQ, for the local request queue data set in the CICS-supplied RDO group DFHCBTS. DFHCBTS is included in the default CICS startup group list, DFHLIST. Figure 54 on page 118 shows the default definition.

```

DEFINE FILE(DFHLRQ) GROUP(DFHCBTS)
DESCRIPTION(Scheduler Services - Local Request Queue)
    RLSACCESS(NO) LSRPOOLID(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 54. The supplied, default, FILE definition for the LRQ

For performance reasons, you may want to change the default definition. For example, because the LRQ is frequently accessed it may be sensible to put the file in its own LSR pool, or to define it to use VSAM nonshared resources. This way, BTS operations will not be affected by other file accesses.

If it's possible that, in your BTS system, many activities may be initiated at the same time (consider a banking application that produces a statement for each customer at the same time each month), it may be a good idea to set the number of strings and buffers to the value of the MXT system initialization parameter. Doing so would avoid 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:

1. 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, you are recommended 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.

2. 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 by means of SPI commands while CICS is running, you must first close the file, which may affect BTS operation. Not all attributes can be changed by SPI commands.

---

## Defining repository files to the CSD

### 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 141.

---

## CEDA DEFINE PROCESSTYPE

Using the CICS business transaction services (BTS) API, you can 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 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. It 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 may 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 55 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.

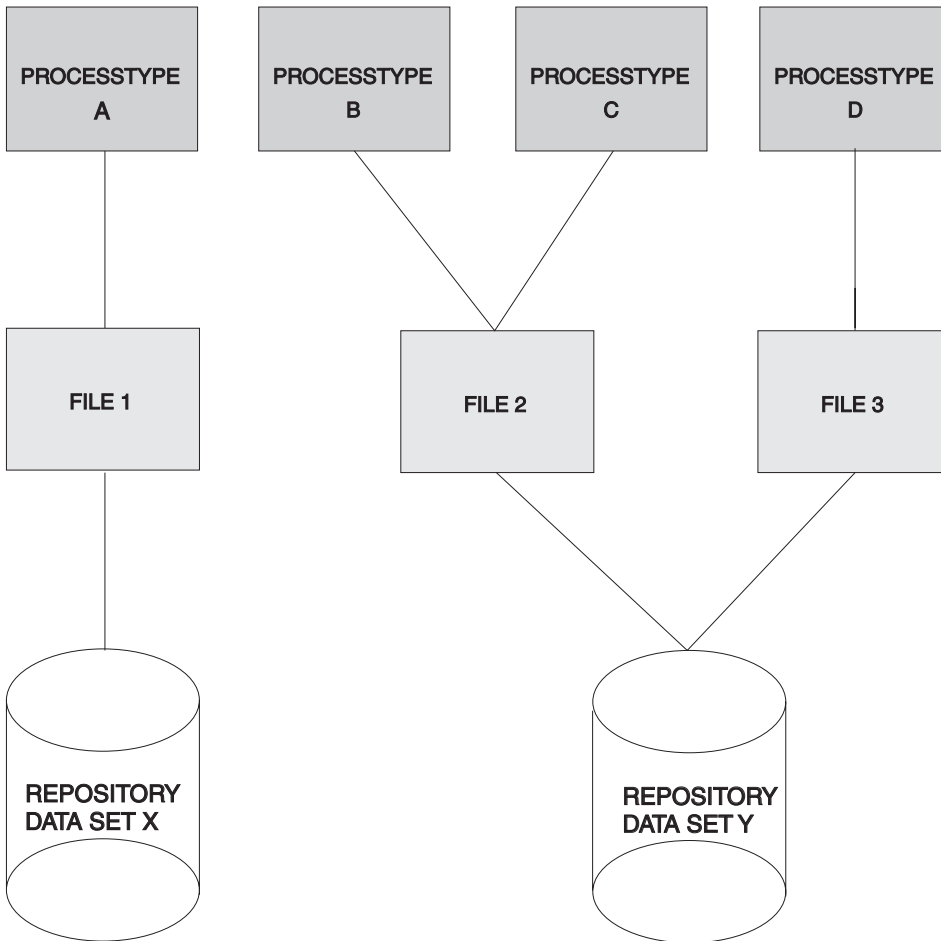


Figure 55. PROCESSTYPE definitions, FILE definitions, and repository data sets

## 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 56. 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 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 162.

**AUDITLOG**(*name*)

specifies the name of a CICS journal to which audit trail records will be 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 will be 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 may 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 two apostrophes.

**FILE**(*name*)

specifies the name of the CICS file definition that will be 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

The security considerations for CICS business transaction services are the authority to access BTS resources, the userid under which a process (business transaction) runs, the userid under which the process's constituent activities run, the authority to attach the process and its constituent activities and the authority to use BTS system programming commands.

**Important:** CICS security is described in the *CICS RACF Security Guide*. This chapter is intended to be read in conjunction with that manual.

Users of external security managers (ESMs) other than the Resource Access Control Facility (RACF<sup>®</sup>) should read this chapter in conjunction with the documentation for their own ESM.

---

### Resource security in BTS

BTS resources (processes, activities, and containers) are protected 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 syncpoint 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 userids

To activate a process or activity, you can use either the RUN or LINK command. Which you use affects the userid under which the process or activity runs.

#### Userids for activities activated by RUN commands

When a process or activity is activated by a RUN command, it may run under a different userid from that of 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 USERID option of the DEFINE PROCESS or DEFINE ACTIVITY command. If the USERID option is omitted, its value defaults to the userid of the transaction that issues the DEFINE command.

The userid obtained from the DEFINE command is referred to as the **defined process userid** or the **defined activity userid**. (In the remainder of this section, we use the term “defined userid” to mean either a defined process userid or a defined activity userid.)

If the USERID option of DEFINE PROCESS or ACTIVITY is specified, CICS performs (at define time) a surrogate security check to verify that the userid of the transaction that issued the DEFINE command is authorized to use the defined userid. 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(defined_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(defined_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 *within* a process or activity is based on the userid under whose authority the process or activity is run—that is, the defined userid or the userid of the transaction that issues the LINK command. This userid must have UPDATE access to the CICS file that corresponds to the process-type.

---

## **Attach-time security for processes and activities**

Attach-time security means the checking of a transaction's authority to attach (activate) a process or activity. It 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 the following BTS system programming commands:

- EXEC CICS CREATE PROCESSTYPE
- EXEC CICS DISCARD PROCESSTYPE
- EXEC CICS INQUIRE PROCESSTYPE
- EXEC CICS SET PROCESSTYPE

---

## Chapter 13. BTS operator commands

This chapter describes the operator commands that can be used to inquire on and control CICS business transaction services resources. It contains:

- “CBAM—BTS browser”
- “CEMT INQUIRE PROCESSTYPE” on page 133
- “CEMT INQUIRE TASK” on page 136
- “CEMT SET PROCESSTYPE” on page 138.

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

Use CBAM to browse the CICS business transaction services objects (process-types, processes, activities, containers, events and timers) known to this 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 by means of installed PROCESSTYPE definitions.
2. All the processes of a selected process-type. These are the processes of the selected type that currently exist on the repository data set pointed to by the installed PROCESSTYPE definition.

Note that, if you are operating BTS in a sysplex and the repository is shared with one or more other regions, some of the processes may have been defined on other regions.

3. The constituent activities of a selected process.
4. The details (program, transid, userid) of a selected activity.
5. One of the following:
  - The containers associated with a selected activity or process, *or*
  - The events in a selected activity's event pool, *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 overtyping them.

### Running the transaction

#### About this task

Start the 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, as shown in Figure 57 on page 126.

## Process-types screen

CBAM			
Processtype	File	Status	Auditlevel
CBTSAUDA	DFHBARF	Enabled	Activity
CBTSSHR	DFHBSSHR	Enabled	Off
CBTSSHRF	DFHBSSHR	Disabled	Activity
CBTSSHR2	DFHBSSHR2	Disabled	Off
MORTLOANS	DFHMORT	Enabled	Process

Use cursor and Enter for Processes

PF3=Return 7=Back 8=Forward

Figure 57. 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 162.

### **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 58.

## Processes screen

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 58. 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 61 on page 130.

### 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 59.

**Activities screen**

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 59. 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 60.

### Activity details screen



Figure 60. CBAM transaction: activity details screen

The *Activity details* screen tells you the name of the program that implements the selected activity, plus the transaction identifier and userid under which the activity runs.

There are also three selectable fields:

#### Containers

Pressing ENTER on this field gives you a list of the containers associated with the selected activity—see Figure 62 on page 130.

#### Events

Pressing ENTER on this field gives you a list of the events in the event pool of the selected activity—see Figure 63 on page 131.

## Timers

Pressing ENTER on this field gives you a list of the timers defined to the selected activity—see Figure 64 on page 132.

## Containers screen

The *Containers* screen lists each container associated with a specified process or activity and tells you the length, in bytes, of the data contained in it.

CBAM	Process MORT000000017	Processtype MORTLOANS
Container	Data length	
ACCOUNT-NO	36	
BORROWER-INFO	1000	

PF3=Return 7=Back 8=Forward

Figure 61. CBAM transaction: containers screen for a process

CBAM	Process MORT000000017	Processtype MORTLOANS
Activity DFHROOT		
Container	Data length	
STATUS	500	

PF3=Return 7=Back 8=Forward

Figure 62. CBAM transaction: containers screen for an activity

## Events screen



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 63. CBAM transaction: events screen

The *Events* screen lists the events in a specified activity's event pool. (Note that the events listed are those that are *currently* in the event pool. Events that have been deleted do not appear in the list.)

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 sub-event, the name of the composite event of which it forms part.

**Event**

The name of an event.

**Fired**

The fire status of the event.

Note that 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) will be shown as 'No', because the act of retrieving the event will have 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

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 64. CBAM transaction: timers screen

The *Timers* screen lists the timers currently defined to a specified activity.

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 by means of 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

INQUIRE PROCESSTYPE 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, which you can use to manage resources by capturing details of when the resource is defined, installed, and last changed. For more information, see in the *CICS Resource Definition Guide*. The resource signature fields are CHANGEAGENT, CHANGEAGREL, CHANGETIME, CHANGEUSRID, DEFINESOURCE, DEFINETIME, INSTALLAGENT, INSTALLTIME, and INSTALLUSRID. See the *CICS Resource Definition Guide* for detailed information about the content of the resource signature fields.

## Input

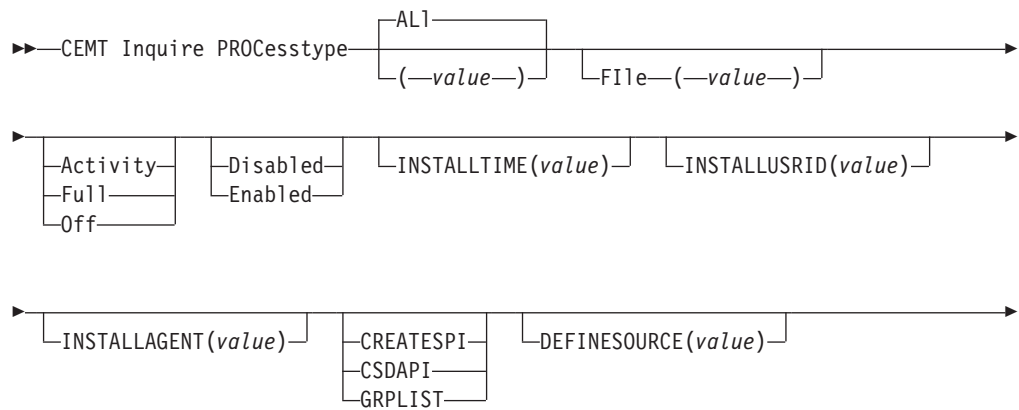
Press the Clear key to clear the screen. There are two ways of commencing this transaction:

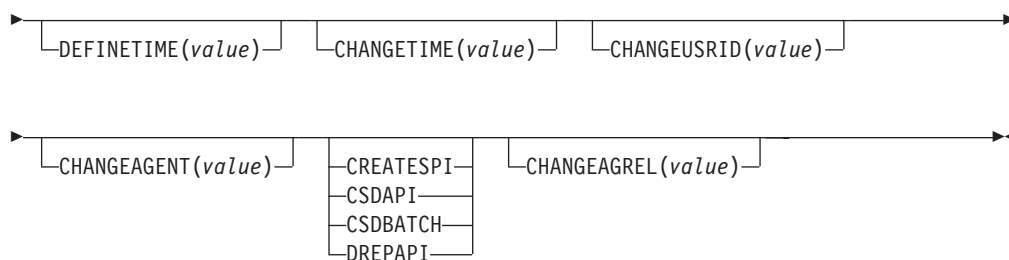
- Type CEMT INQUIRE PROCESSTYPE (the minimum abbreviation is CEMT I PROC). You get a screen that lists the current status.
- Type CEMT INQUIRE PROCESSTYPE (CEMT I PROC) followed by as many of the other attributes as are necessary to limit the range of information that you require. So, for example, if you enter `cemt i proc en`, the resulting display will show you the details of only those process-types that are enabled.

To change various attributes, you can:

- Overtyping your changes on the INQUIRE screen after tabbing to the appropriate field.
- Use the CEMT SET PROCESSTYPE command.

## CEMT INQUIRE PROCESSTYPE





**ALI**

is the default. Information about all process-types is displayed, unless you specify a selection of process-types to be queried.

(value)

is 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 65. 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 66.

```

I PROCESSTYPE(*)
RESULT - OVERTYPE TO MODIFY
  Processtype(SALESTYPE4 )
  File(SALESF4 )
  Auditlog(PLOG51 )
  Auditlevel(Activity )
  Status( Enabled )

```

Figure 66. 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:

**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 162.

**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 identifying 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(value)**

Displays the 4-digit number of the CICS release that was running when the resource definition was last changed.

**CHANGETIME(value)**

Displays an ABSTIME value that represents the time stamp when the resource definition was last changed.

**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 in the *CICS Resource Definition Guide*.

**DEFINETIME(value)**

Displays an ABSTIME value that represents the time stamp when the resource definition was created.

**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 identifying 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**(*value*)

| Displays an ABSTIME value that represents the time stamp when the resource  
| was installed.

| **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:

| **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

INQUIRE TASK 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

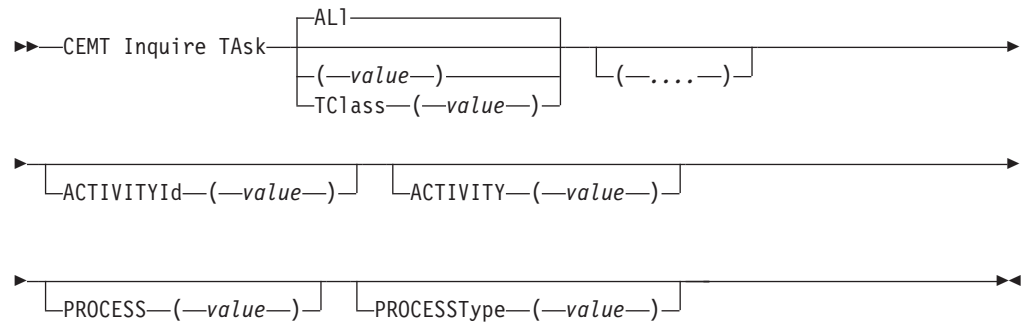
Press the Clear key to clear the screen. There are two ways of commencing this transaction:

- Type CEMT INQUIRE TASK (the minimum abbreviation is CEMT I TA). You get a display that lists the current status.
- Type CEMT INQUIRE TASK (CEMT I TA) followed by as many of the other attributes as are necessary to limit the range of information that you require. So, for example, if you enter `cemt i ta i`, the resulting display will show you the details of only those tasks for which the data is not shared with other tasks (isolated).

To change various attributes, you can:

- Overtyping your changes on the INQUIRE screen after tabbing to the appropriate field.
- Using the CEMT SET TASK command.

## CEMT INQUIRE TASK



*(value)*

is the CICS-generated task number, in the range 1–99999.

**AL1**

is the default. The maximum number of tasks that can be displayed is 32000.

**TClass***(value)*

is 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, nor can you use the symbols \* and + 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(T0) Use(BELL ) Uow(AB9001D5F56CC800) Hty(ZCIOWAIT) Hva(DFHZARQ)
Tas(0000037) Tra(CEMT) Fac(S21D) Run Ter Pri( 255 )
  Sta(T0) Use(BELL ) Uow(AB9002E745F93A00)
  
```

Figure 67. 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 68 on page 138.

```

IN TASK
RESULT - OVERTYPE TO MODIFY
Task(0000033)
Tranid(CEMT)
Facility(944D)
Runstatus(Suspended)
Ftype(Term)
Priority( 255 )
Purgetype(          )
Startcode(T0)
Userid(BELL)
Uow(AB9001D5F56CC800)
Htype()
Hvalue()
Htime()
Indoubt(Backout)
Indoubtwait(Wait)
Bridge()
Activityid()
Activity()
Process()
Processtype()

```

Figure 68. 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

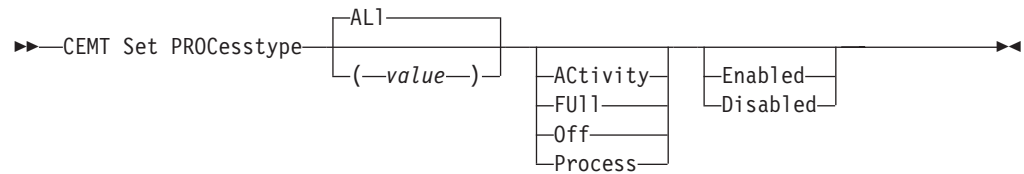
SET PROCESSTYPE enables you to change the current state of audit logging and the enablement status of BTS PROCESSTYPE definitions installed on this CICS region.

**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

### A**CT**ivity | F**U**ll | O**ff** | P**RO**cess

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:

#### A**CT**ivity

Activity-level auditing. Audit records will be written from:

1. The process audit points
2. The activity primary audit points.

**FU**ll 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.

#### P**RO**cess

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 162.

### A**LI**

specifies that any changes you specify are made to all process-types that you are authorized to access.

### E**n**abled | D**is**abled

specifies whether new processes of this type can be created. The values are:

#### D**is**abled

The installed definition of the process-type is disabled. New processes of this type cannot be defined.

#### E**n**abled

The installed definition of the process-type is enabled. New processes of this type can be defined.

### P**RO**Cesstype(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. Within a sysplex you can, for example:

- Use workload separation to ensure that processes of the same process-type are handled by a particular set of regions.
- Use workload balancing to route activity requests across a set of regions. This means that, within a single process:
  - The activities that constitute the process may execute on several regions.
  - Different activations of the same activity may execute on different regions.

---

### The scope of a BTS-set

By **BTS-set** we mean the set of CICS regions across which related BTS processes and activities may execute. All the regions in a BTS-set:

- Must be interconnected (to support routing of activities between the regions). This means, not that the regions must be in the same CICSplex, but that each region in the BTS-set must be connected to every other region—see “Understanding distributed routing” on page 143.
- 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—see Figure 71 on page 147. 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 may execute on several regions. Therefore, in order to collate the audit data for each process, your audit logs should use MVS shared logstreams.

If all the CICS regions in your BTS-set are in the same MVS image, you can define the logstreams to use either coupling facility structures or DASD-only logging. However, if the CICS regions are on *different* MVS images, the logstreams should

use coupling facility structures rather than DASD-only logging. This is because CICS regions on different MVS images cannot access the same DASD-only logstream at the same time.

If your BTS-set spans multiple MVS images and you use DASD-only logstreams for your BTS logs, you will not be able to use shared logstreams. In this case, the audit records for a particular process could be split between several logstreams; you have to collate the data yourself.

For further information about audit logs, see Chapter 16, “Creating a BTS audit trail,” on page 161.

---

## Dynamic routing of BTS activities

Within a BTS-set, your BTS processes and activities can be routed dynamically across the participating regions.

Note that BTS routing is at the activation level; for example, within the same process, different activations of the same activity may execute on different regions. When an event is signalled, the relevant activity is activated in the most appropriate region in the BTS-set, based on one or more of the following:

- 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—by means of a RUN ASYNCHRONOUS command—can be routed either dynamically or statically.

Processes and activities that are activated synchronously with the requestor—by means of a RUN SYNCHRONOUS or LINK command—are always run locally. They cannot be routed, *neither dynamically nor 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 its parent's TRANSID—there is no **context-switch**.)

Thus, to be eligible for dynamic routing:

- A process or activity must be run asynchronously with the requestor, by means of 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. That is, once a BTS process or activity has been routed to a target region it cannot be re-routed 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**. Likewise, there are two user-replaceable sample routing programs—the **dynamic routing program**, DFHDYP, which implements the “hub” model, and the **distributed routing program**, DFHDSRP, which implements the distributed 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 this differs from the traditional “hub” model.

### The hub model

The “hub” is the model that has traditionally been used with CICS dynamic transaction routing. 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 no dynamic routing. Figure 69 shows a “hub” routing model.

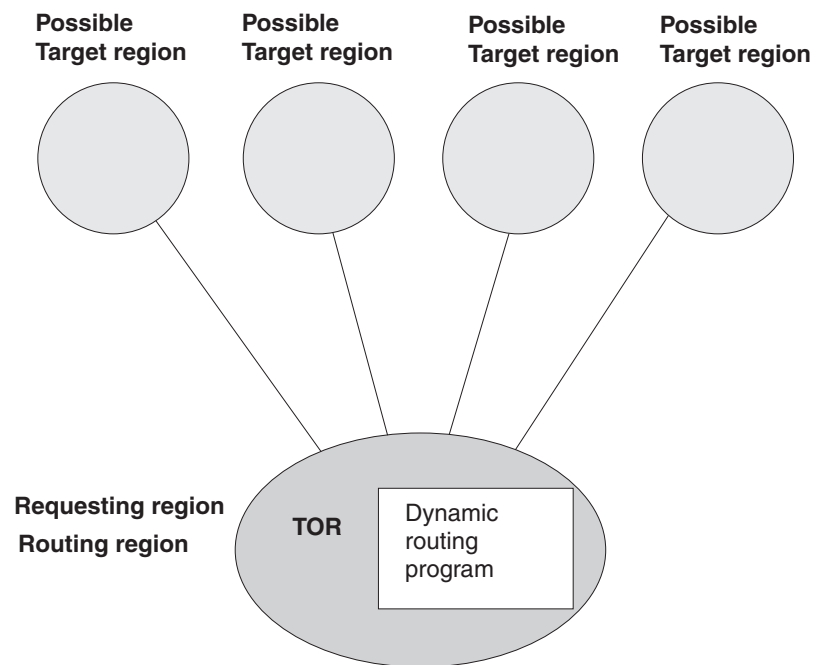


Figure 69. Dynamic routing using a hub routing model. One routing region (the TOR) selects between several target regions.

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 may be both a routing region and a target region. A distributed routing program runs in each region. Figure 70 on page 145 shows a distributed routing model.

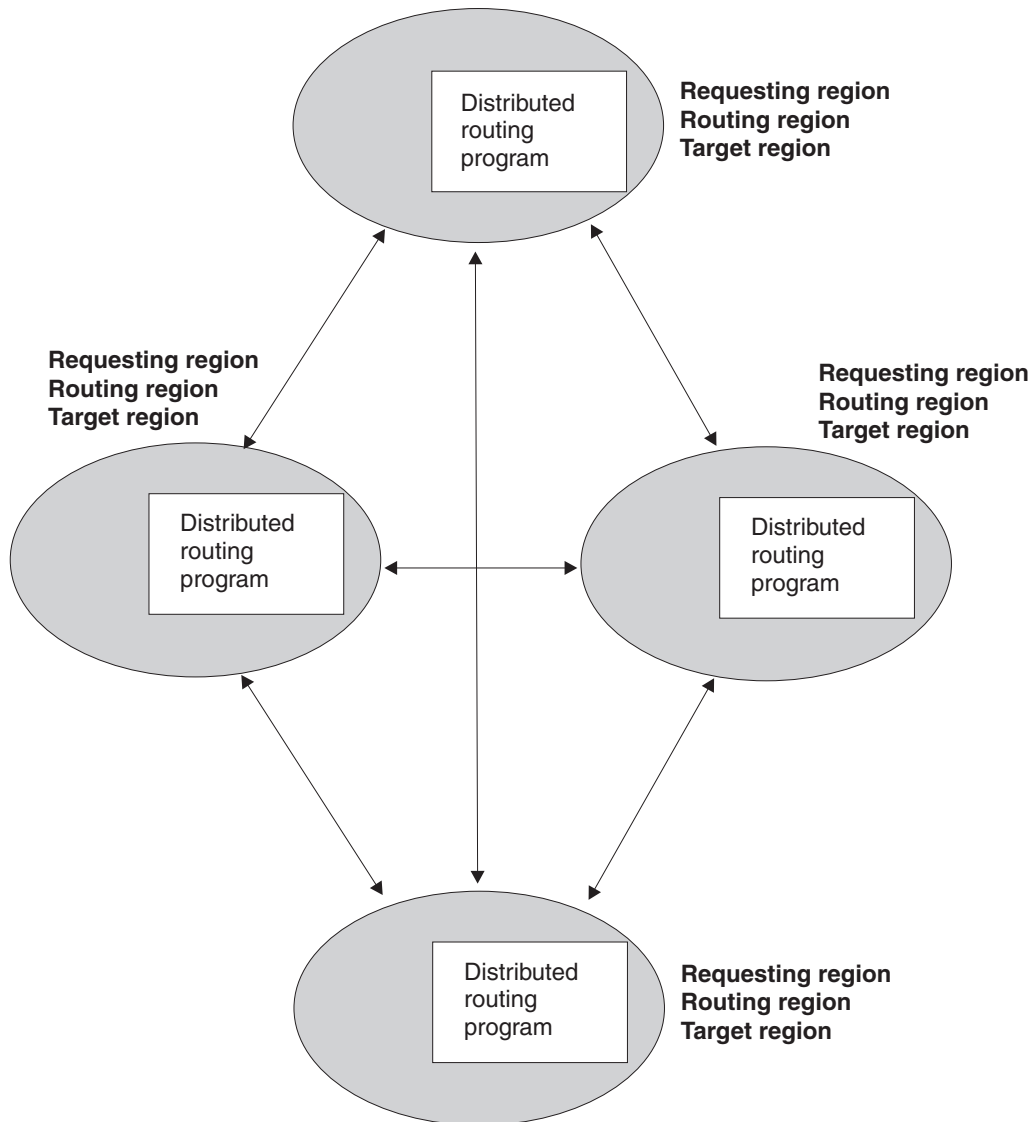


Figure 70. Dynamic routing using a distributed routing model. Each region may be both a routing region and a target region.

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 dataspace.)

## Controlling BTS dynamic routing

### 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 148.
2. Using the CICSplex System Manager (CICSplex SM) product to:
  - Specify workload separation for your BTS processes
  - Manage affinities
  - Control workload balancing of the transactions associated with BTS activities.See “Using CICSplex SM with BTS” on page 151.

---

## Creating a BTS-set

### About this task

The recommended method of using BTS in a sysplex is to have several sets of BTS regions (BTS-sets). Each set is designed to deal with one or more process-types (types of business transaction). The easiest way to create the sets is by cloning individual regions.

Figure 71 on page 147 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.



## MVS sysplex

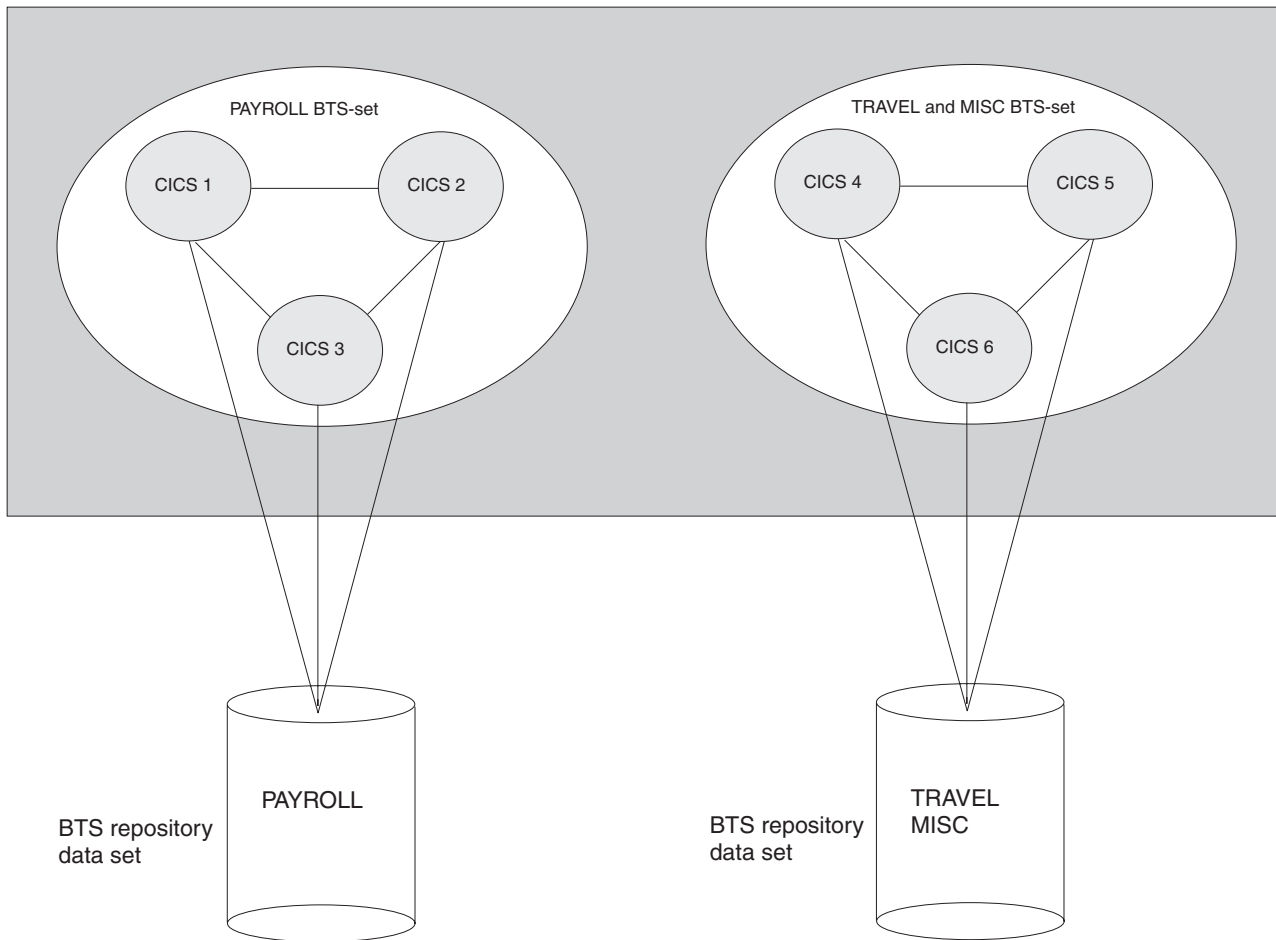


Figure 71. A sysplex containing two BTS-sets. BTS-set 1 handles all processes of type PAYROLL. BTS-set 2 handles all processes of types TRAVEL and MISC.

The number of regions in a BTS-set will be 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 Defining transactions for transaction routing the *CICS Intercommunication Guide*.
4. Enable the distributed routing program. See the next section.

## Naming the routing program

You specify which distributed routing program CICS is to use on 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 EXEC CICS or CEMT 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 an EXEC CICS or CEMT SET SYSTEM command. The DSRTPROGRAM option specifies the program name.

---

## Using a CICS distributed routing program

This section describes some of the things you need to consider if you decide to write your own distributed routing program.

### How the distributed routing program relates to the dynamic routing program

The two CICS-supplied user-replaceable programs for dynamic routing complement each other.

#### 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, on the other hand, 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, on the other hand, 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. Figure 72 on page 150 shows the points at which the distributed routing program is invoked, and the region on which each invocation occurs.

## Writing a distributed routing program

### About this task

You can use the CICS-supplied default distributed routing program, DFHDSRP, as a model when writing your own version. 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 invoked

For processes and activities started by RUN ASYNCHRONOUS commands, your distributed routing program is invoked at the following points:

#### On the requesting region:

1. Either of the following:
  - For routing the activity. This occurs when the transaction associated with the activity is defined as DYNAMIC(YES).
  - For notification of a statically-routed activity. This 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.

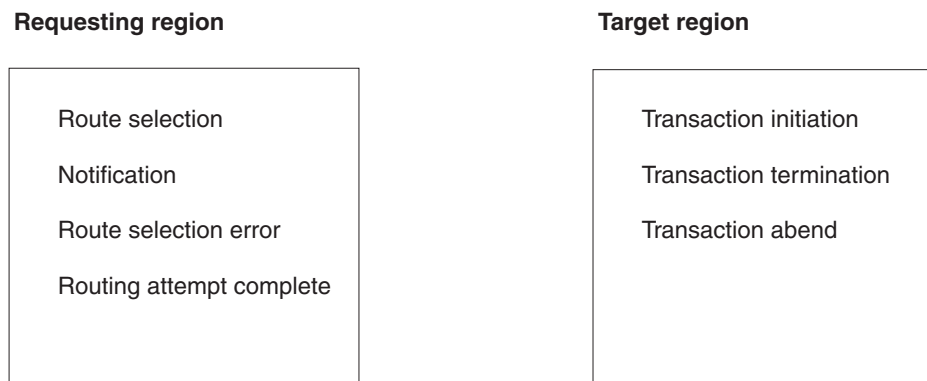
This invocation signals that (unless the requesting region and the target region are one and the same) the requesting region's responsibility for the transaction has been discharged. The routing program might, for example, use this invocation to release any resources that it has acquired on behalf of the transaction.

**On the target region:**

*These invocations occur only if the routing program on the requesting region has specified that it should be reinvoked 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 72 shows the points at which the distributed routing program is invoked, and the region on which each invocation occurs. Note that 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 72. When and where the distributed routing program is invoked*

When it is invoked 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 invoked 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 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**

Because the distributed routing program executes outside a unit of work environment, your program must not:

- Alter any recoverable resources
- Issue file control or temporary storage requests.

---

## Using CICSplex SM with BTS

Rather than writing your own distributed routing program, you can use CICSplex SM services to route BTS activities around your BTS-sets.

### Overview of CICSplex SM Workload Management

CICSplex SM's workload management (WLM) function provides a dynamic routing program that can route eligible transactions (those defined as dynamic) from a requesting region to a suitable target region selected at the time the transaction is initiated.

#### WLM functions

CICSplex SM's dynamic routing program supports:

- *Workload separation*, which is the routing of particular transactions to a particular 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 CICSplex SM's 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 balancing*, which is the routing of transactions among a group of target regions according to the availability and activity levels of those regions. Workload balancing can be used in addition to, or in place of, workload separation. For example, CICSplex SM can balance 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.
- *Inter-transaction affinity*, which is the requirement for related transactions to be processed by the same target region. The IBM CICS Interdependency Analyzer for z/OS can be used to identify affinities between transactions. For information about the IBM CICS Interdependency Analyzer for z/OS, see the *CICS Interdependency Analyzer for z/OS User's Guide and Reference*.

For further introductory information about CICSplex SM and workload management, see the CICSplex SM Concepts and Planning guide.

### Using CICSplex SM to route BTS activities

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 may introduce affinities. You must define such affinities to CICSplex SM WLM, so that it is able to make sensible routing decisions. It is particularly important to specify each affinity's lifetime; failure to do so may restrict WLM's routing options unnecessarily.

Note that:

- A single CICSplex SM may 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

This part of the manual contains advice on how to:

- Solve BTS-related problems
- Measure and tune BTS performance.

*Table 12. Problems and performance road map*

<b>If you want to...</b>	<b>Refer to...</b>
Find solutions to some common problems	Chapter 15, "Having problems?," on page 155
Track the progress of BTS applications across a BTS-set	Chapter 16, "Creating a BTS audit trail," on page 161
Examine records on a repository data set	Chapter 17, "Examining BTS repository records," on page 181
Know where to find the meaning of a BTS message, or how to control BTS trace or dump	Chapter 18, "BTS messages, trace, and dump," on page 193
Know where to find BTS monitoring data	Chapter 19, "Tuning BTS performance," on page 197





---

## Chapter 15. Having problems?

This chapter suggests solutions for some of the more common problems that you might meet when running a BTS system.

---

### Dealing with stuck processes

A process is said to be “stuck” when it cannot proceed because it is waiting for an event that cannot, or does not, occur. 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 158.
- A CICS region fails—see “Dealing with CICS failures” on page 160.

### Application design errors

A stuck process may be caused by a program logic error. For example, consider the following scenarios:

#### 1. Outstanding user events:

- a. One of the process's activities 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 activity's reattachment queue, 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 a case like this, the activity becomes dormant, and there is no way for it to be 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 “Acquiring an activity” on page 70.) However, the expected user input doesn't happen. Perhaps the clerk is sick, or the data he or she requires is not available. The process is stuck.

The recommended way to recover from this scenario is to set a timer which, if the expected external interaction does not occur within a specified period, will cause the activity (or its parent) to be reactivated anyway.

#### 3. Timer error:

A programming error results in a timer being set to expire in five days rather than five minutes. The process is stuck. See “Restarting stuck processes” on page 156.

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

## Restarting stuck processes

### About this task

For advice on restarting processes that are stuck because of unserviceable requests, see “Dealing with unserviceable requests” on page 158.

For advice on restarting processes that are stuck because of a CICS failure, see “Dealing with CICS failures” on page 160.

### Using activity timers

#### About this task

The best way to restart processes that are stuck for other reasons—including application errors—is to use timers. For example, a parent may set a timer which will cause it to be reactivated after a specified period, if a particular child activity does not complete. (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 may have varying “natural” lifespans; and these lifespans may 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 will probably 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. On the other hand, 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

#### About this task

As well as, or instead of, setting timers for individual child activities, you could set a timer for the process itself. That is, the root activity could set a timer with an expiry time some time after the whole process could reasonably be expected to have completed.

If the process is short-lived, you may 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. This 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 208.)
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 stuck activity's completion event 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

### About this task

To make it easier for a root activity to identify which of its descendant activities are stuck, you could use **status containers**. 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 just 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 will always be in one of a limited number of processing states. (The “processing states” we refer to here are application-dependent and quite 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

### About this task

We have said that it is difficult from *outside* a process to tell whether the process is stuck or merely dormant. To help you decide, you can use a utility program.

### CICS-supplied utility programs:

CICS supplies two utility programs for diagnostic purposes:

#### The audit trail utility, DFHATUP

You can use DFHATUP to print selected audit records from a logstream. 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 161.

#### 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 181.

## User-written utility programs:

### About this task

You could write a utility program that could check for and restart stuck processes, particularly if 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 latter's completion event fires, causing its parent 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.

---

## Dealing with activity abends

### About this task

If a program that implements an activity abends, the activity's parent 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:

- Retry the failed activity—see “Retrying failed activities” 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. It may be, for example, that an activity is not available, or that the region on which the request is to execute is not accessible.

### Unserviceable routing requests

If you operate BTS in a sysplex, it is possible to route processes and activities across a set of CICS regions called a **BTS-set**. (For detailed information about routing processes and activities, see Chapter 14, “Using BTS in a sysplex,” on page 141. )

When a process or activity is started by a RUN ASYNCHRONOUS command, it may be routed either statically or dynamically. Mostly, you will probably choose dynamic rather than static routing.

#### 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 reinvoked 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 re-route the request to an alternative region, assuming that alternatives are available?

Sometimes, perhaps due to a transaction affinity, it may be essential that an activation should run on a specific region, and no other. If this is the case, 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**

When a request is “unserviceable”, CICS:

1. Issues message DFHSH0105, which identifies the request and indicates that it cannot be serviced.
2. Retries the request 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 will purge 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**

### **About this task**

In many cases, CICS will resolve 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.

Your operators should watch for occurrences of DFHSH0105 and DFHSH0106 messages. They should investigate why the request is unserviceable, and take any necessary corrective action. It may 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, not only are all BTS processes on the failing region halted, but processes on other regions may also become stuck, because expected events are not generated.

If a CICS region fails, you should perform an emergency restart.

Only in very rare circumstances—for example, if the CICS global catalog or system log is corrupted—should 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.

At an initial or cold start:

- BTS repository data sets are unchanged.

**Note:** Repository data sets are never re-initialized at CICS startup, because they may be shared.

- The local request queue data set is unchanged. All information about BTS timers, pending and unserviceable requests is preserved. However, it's likely that some of this information will now be irrelevant or invalid, because it refers to processes that no longer exist.

### Emergency starts

At 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 re-run. 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

At an initial or cold start:

- BTS repository data sets are unchanged.

**Note:** Repository data sets are never re-initialized at CICS startup, because they may be shared.

- The local request queue data set is unchanged. All information about BTS timers, pending and unserviceable requests is preserved. However, it's likely that some of this information will now be irrelevant or invalid, because it refers to processes that no longer exist.

---

## Chapter 16. Creating a BTS audit trail

### About this task

This chapter tells you how to create an audit trail for BTS processes and activities. It contains:

- “Introduction to BTS audit trails”
- “Specifying the level of audit logging” on page 162
- “Audit trail constraints—using DASD-only logstreams” on page 164
- “Audit trail examples” on page 165
- “Using the audit trail utility program, DFHATUP” on page 168.

---

### Introduction to BTS audit trails

You may want to create an audit trail for the BTS processes and activities that run in your CICS systems. Doing so allows you to:

- Track the progress of complex business transactions
- 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 logstream.
2. Trace entries record the progress of tasks over a relatively short period of time, 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 may 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 logstreams 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.

How to specify the levels, and what they mean, is described in “Specifying the level of audit logging” on page 162.



Audit log records are written to an MVS logstream by the CICS Log Manager. You can read the records off-line 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 logstream to a backup file and to delete the logstream. 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 logstream only when the buffer is full or a syncpoint occurs. This means that, when multiple CICS regions share the same logstream, audit records may not be in exact date and time order.

---

## Specifying the level of audit logging

You 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:

- When a process is first defined, BTS obtains the process's audit level and audit log information 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 has no effect on 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 should not specify the SMF data set.
- Several process-types can share the same audit log.
- In a sysplex, different parts of a process may 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—using DASD-only logstreams” on page 164.

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 will be written from:

1. The process audit points
2. The activity primary audit points.

That is, an audit record will be 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 will be written from:

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

That is, an audit record will be 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 will be written. This is the default value.

#### PROCESS

Specifies process-level auditing. Audit records will be written from the process audit points only. That is, an audit record will be 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 will take to run. The fewer records written, the less information there will be 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. Note that changes to an installed PROCESSTYPE definition have no effect on *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 this is done successfully, auditing is resumed and a CICS message is issued to this effect. Some audit records will have been lost.

---

## Audit trail constraints—using DASD-only logstreams

### About this task

If you are running BTS in a sysplex, the activities that make up a process may run on different CICS regions. If you want to use audit logging, you should ensure that audit records can be written to a single logstream from any region that any of the activities run on.

If the CICS regions are in the same MVS image, you can define the logstream to use either a coupling facility structure or DASD-only logging. However, if the CICS regions are on *different* MVS images, the logstream should 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 logstream at the same time.

If the regions are in different MVS images and you use DASD-only logging, you will not be able to use shared logstreams for your BTS logs. This means that audit records for a single process may be split across several logstreams; you will have to collate them yourself.

---

## Audit trail examples

Figure 73 on page 166 shows the sequence of activations of a BTS process, SALES1234567890. The activities that make up the process run on two CICS regions. For the sake of clarity, the example does not show the activations of any other processes that might also be running in these regions.



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 child's completion status. 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 by means of 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 the sake of brevity, 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 73 on page 166):

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 DFHROOT's 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

A setting of FULL on the AUDITLEVEL attribute of a PROCESSTYPE definition specifies full auditing for processes of the defined type. 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

The audit trail utility program, DFHATUP, allows you to read BTS audit records from a logstream and to print them. It allows you to filter selected records. It formats the records to make them easier to interpret.

### Using DFHATUP to read audit logs

#### About this task

You should run DFHATUP as a batch job against a logstream that is not in use by any CICS regions. (If you run it against a logstream that is connected to CICS, DFHATUP will not find any records that CICS has in its buffers.)

DFHATUP reads the records in the order that they were written to the MVS logstream. 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

Figure 74 on page 169 shows an example job stream to run the DFHATUP program. The job stream should 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 74 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 170.

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 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.JRNLO01,  
// 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 74. Sample job to run the DFHATUP utility program

## EXEC parameters

You can use the PARM keyword on the EXEC statement to pass one or more of the following parameters to the DFHATUP utility. The form of the EXEC statement is:

```
EXEC PGM=DFHATUP,PARM='parm1,...,parmn'
```

### NATLANG({EN|CSvKA})

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 is used to pass information to DFHATUP. 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:**

```
SYSIN DD *  
[AUDITLOG(name)]  
[PTYPE(name) <PROCESS(name)>]  
[PROCESS(name)]  
[ACTIVITY(name)]
```

An AUDITLOG statement cannot contain additional arguments. Other statements may 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 logstream.

**ACTIVITY**(*name*)

The 1-16 name of an activity. Records for this activity will be 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 and/or a PROCESS argument 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 and/or an ACTIVITY argument 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 and/or an ACTIVITY argument on adjoining lines, using a continuation character in column 72.

## **Example output from the DFHATUP utility**

CICS writes records to an audit log in chronological order. Particularly on busy systems within a sysplex, records from different processes or from different activities within 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 may want to select particular sets of records.

“Audit trail examples” on page 165 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 75 would 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 75. Example control statements, to format all the records for the SALES1234567890 process*

### **Example audit trail—process-level auditing**

Extending our previous example, Figure 76 on page 172 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 was set to 'PROCESS'
- The control statements in the SYSIN data set specified that all records for the SALES1234567890 process should be formatted.

Exec Parm Options: Natlang (EN)  
 Translate (mixedcase)  
 Pagesize (60)

```

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 )
                                (CCDCCDF11CCCDCEC4CECECFB3235800CCDDDE44444444)
                                (21469351A172924981B98363339AA51E6014689663000000000)
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 )
                                (CCDCCDF11CCCDCEC4CECECFB3235800CCDDDE44444444)
                                (21469351A172924981B98363339AA51E6014689663000000000)
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 )
                                (CCDCCDF11CCCDCEC4CECECFB3235800CCDDDE44444444)
                                (21469351A172924981B98363339AA51E6014689663000000000)
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 )
                                (CCDCCDF11CCCDCEC4CECECFB3235800CCDDDE44444444)
                                (21469351A172924981B98363339AA51E6014689663000000000)
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 )
                                (CCDCCDF11CCCDCEC4CECECFB3235800CCDDDE44444444)
                                (21469351A172924981B98363339AA51E6014689663000000000)
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 )
                                (CCDCCDF11CCCDCEC4CECECFB3235800CCDDDE44444444)
                                (21469351A172924981B98363339AA51E6014689663000000000)
Current: Transid(R ) Userid(CICSUSER) Date(1999.029) Time(15:59:26.849330)
    
```

Number of Audit records read : 6  
 Number of records selected : 6  
 Processing Complete

Figure 76. Example audit trail, showing the types of record written for process-level auditing

### Example audit trail—activity-level auditing

Figure 77 on page 173 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 was set to 'ACTIVITY'
- The control statements in the SYSIN data set specified that all records for the SALES1234567890 process should be formatted.

Exec Parm Options: Natlang (EN)  
 Translate (mixedcase)  
 Pagesize (60)

```
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 )
(CCDCDCFC11CCCCDCEC4CECECFB3349C00CCDDDE444444444)
(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 )
(CCDCDCFC11CCCCDCEC4CECECFB3349C00CCDDDE444444444)
(21469351A172924981B98363339A28606014689663000000000)

Current: Transid(P ) Program(P ) Userid(CICSUSER) Date(1999.029) Time(15:23:53.324025)
```

Figure 77. Example audit trail, showing the types of record written for activity-level auditing

### Example audit trail: full auditing

Figure 78 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 was set to 'FULL'
- The control statements in the SYSIN data set specified that all records for the SALES1234567890 process should be formatted.

Exec Parm Options: Natlang (EN)  
 Translate (mixedcase)  
 Pagesize (60)

Figure 78. Example audit trail, showing the types of record written for full auditing (Part 1)

```

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 )
(CCDCDCDF11CCCDCEC4CECECFB2902A00CCDDDE4444444444)
(21469351A172924981B98363339A709F5014689663000000000)

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 )
(CCDCDCDF11CCCDCEC4CECECFB2902A00CCDDDE4444444444)
(21469351A172924981B98363339A709F5014689663000000000)

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 )
(CCDCDCDF11CCCDCEC4CECECFB2902A00CCDDDE4444444444)
(21469351A172924981B98363339A709F5014689663000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29B0300C4444444444444444)
(21469351A172924981B98229672A7111C0120000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29B0300C4444444444444444)
(21469351A172924981B98229672A7111C0120000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29B0300C4444444444444444)
(21469351A172924981B98229672A7111C0120000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29B0300C4444444444444444)
(21469351A172924981B98229672A7111C0120000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29DF500C4444444444444444)
(21469351A172924981B98229672A71C690130000000000000000)

Current: Transid(R ) Program(R ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.472960) Activity(DFHROOT )
    
```

Figure 79. Example audit trail, showing the types of record written for full auditing (Part 2)

```

Ptype(SALES ) Function(Run Activity ) Process(SALES1234567890 ) System(SYS1) Auditlog(BAMAUDIT)
Taskno(0000034) Activity(C) ) Asynchronous
                                ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j.6...C )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (21469351A172924981B98229672A71C69013000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (21469351A172924981B98229672A71EE90140000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (21469351A172924981B98229672A71EE90140000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (21469351A172924981B98229672A71C69013000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (21469351A172924981B98229672A71EE90140000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB29DF500C444444444444444)
                                (21469351A172924981B98229672A72BA90160000000000000000)
Current: Transid(C ) Program(C) ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.668485) Activity(C)
    
```

Figure 80. Example audit trail, showing the types of record written for full auditing (Part 3)

```

Ptype(SALES ) Function(Run Activity ) Process(SALES1234567890 ) System(SYS1) Auditlog(BAMAUDIT)
Taskno(0000036) Activity(F ) Asynchronous
                                ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..I..F )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB290AC00C444444444444444)
                                (21469351A172924981B98229672A72BA9016000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB2906900C4444444444444444)
                                (21469351A172924981B98229672A729460150000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB290AC00C4444444444444444)
                                (21469351A172924981B98229672A72BA90160000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB2925000C4444444444444444)
                                (21469351A172924981B98229672A72F750170000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB2925000C4444444444444444)
                                (21469351A172924981B98229672A72F750170000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB2925000C4444444444444444)
                                (21469351A172924981B98229672A72F750170000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB290AC00C4444444444444444)
                                (21469351A172924981B98229672A72BA90160000000000000000)
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 )
                                (CCDCCDCF11CCCDCEC4CEDFECEFB290DF500C4444444444444444)
                                (21469351A172924981B98229672A71C690130000000000000000)
Current: Transid(C ) Program(C ) Userid(CICSUSER) Date(1999.029) Time(14:36:14.979781)
    
```

Figure 81. Example audit trail, showing the types of record written for full auditing (Part 4)

```

Ptype(SALES ) Function(Completion ) Process(SALES1234567890 ) System(SYS1) Auditlog(BAMAUDIT)
Taskno(0000038) Activity(E ) ) Compstatus(Normal )
ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..k..o..E )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2906900C44444444444444)
(21469351A172924981B98229672A71C69013000000000000000)
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 )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2902A00CCDDDE4444444444)
(21469351A172924981B98229672A71C69013000000000000000)
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(0000043) Activity(C ) ) Compstatus(Normal )
ActivityId(BAMFILE1..GBIBMIYA.IYK2ZFX2..j..6..C )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2902A00CCDDDE444444444444)
(21469351A172924981B98229672A71C69013000000000000000)
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 )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2902A00CCDDDE444444444444)
(21469351A172924981B98229672A71C69013000000000000000)
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 )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2925000C4444444444444444)
(21469351A172924981B98229672A72F75017000000000000000)
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 )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2925000C4444444444444444)
(21469351A172924981B98229672A72F75017000000000000000)
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 )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2925000C4444444444444444)
(21469351A172924981B98229672A72F75017000000000000000)
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 )
(CCDCDCDF11CCCCDCEC4CEDFECEFB2925000C4444444444444444)
(21469351A172924981B98229672A71EE90140000000000000000)
Current: Transid(D ) Program(D ) Userid(CICSUSER) Date(1999.029) Time(14:36:21.948512)
    
```

Figure 82. 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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29FED00C444444444444444)
(21469351A172924981B98229672A78F7F018000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29FED00C444444444444444)
(21469351A172924981B98229672A78F7F018000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29FED00C444444444444444)
(21469351A172924981B98229672A78F7F018000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29FED00C444444444444444)
(21469351A172924981B98229672A78F7F018000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29DBF00C444444444444444)
(21469351A172924981B98229672A71EE9014000000000000000)

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 )
(CCDCDCDF11CCCDCEC4CEDFECEFB29DBF00C444444444444444)
(21469351A172924981B98229672A71EE9014000000000000000)

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.IYCWT39....v..DFHROOT )
(CCDCDCDF11CCCDCEC4CECECFB2902A00CCDDDE4444444444)
(21469351A172924981B98363339A709F5014689663000000000)

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.IYCWT39....v..DFHROOT )
(CCDCDCDF11CCCDCEC4CECECFB2902A00CCDDDE4444444444)
(21469351A172924981B98363339A709F5014689663000000000)

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 83. 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. It 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 doesn't know, at the time the activation is started, whether or not there are any.
  - An activation terminates but there are several more fired events that it needs to service. BTS reactivates the activity immediately, but doesn't 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

### About this task

This chapter tells you how to examine records on a BTS repository data set. It contains:

- “Introduction to the repository utility program, DFHBARUP”
- “Using DFHBARUP.”

---

### Introduction to the repository utility program, DFHBARUP

There may be times when you need to examine records on a repository. This may be necessary for diagnostic purposes, for example.

You can use the repository utility program, DFHBARUP, to print selected records from a specified repository data set.

### The repository utility program, DFHBARUP

By default, DFHBARUP prints all the records currently on the specified repository. Thus, you could use it to take a “snapshot” of your BTS system at the time the utility is run (if you have more than one repository, it will be 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 may be added constantly; conversely, as processes complete and events are deleted their associated records will disappear from the repository.

DFHBARUP allows you to filter selected records—you could, for example, 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 activities' event pools.

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 activity's event pool.

DFHBARUP formats the records it extracts, to make them easier to interpret.

---

### Using DFHBARUP

#### About this task

Run DFHBARUP as a batch job.

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

Figure 84 shows an example job stream to run the DFHBARUP program. The job stream should include DD statements for the following data sets:

### The repository

The repository data set to be examined to produce the output data. (Figure 84 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 183.

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 84. 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 one or more of the following parameters to the DFHBARUP utility. The form of the EXEC statement is:

```
EXEC PGM=DFHBARUP,PARM='parm1,...,parmn'
```

**NATLANG({EN|CSvKA})**

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|nm})**

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 is used to pass information to DFHBARUP. You can include statements to select specific sets of records to be formatted. Comments are identified by an asterisk in the first position. The SYSIN data set must be defined.

**Format of the SYSIN control statements:**

```
SYSIN DD *
[REPOSITORY(name)]
[PTYPE(name)]
[PROCESS(name)]
[ACTIVITY(name)]
```

The REPOSITORY statement cannot contain additional arguments. Other statements may 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 logstream.

**ACTIVITY(name)**

The 1-16 character name of an activity. Only records for activities of this name will be formatted. To limit the scope of the search, specify a PROCESS or PTYPE argument in conjunction 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

The example control statements in Figure 85 would 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 85. Example control statements, to format all records on the SALEREP repository for the SALES1234567890 process

Figure 86 shows the output that might be produced by the control statements in Figure 85.

```
CICS Business Transaction Services - Parameter Validation  
Exec Parm Options: Natlang (EN)  
                   Translate (mixedcase)  
                   Pagesize (60)
```

```
Date : 29/01/1999 Time : 14:39:04 Page 0001
```

```
REPOSITORY(SALEREP)
```

Figure 86. 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  
 11CCCCDCEC4CECECF44042F00CCDDDE4444444444  
 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 *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 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
11CCCCDCEC4CEDFECECF440F3A00CCE6F44444444444
A172924981B98229672B8CF9F01133D30000000000000
Child Name : ACT_ONE Id : ..GBIBMIYA.IYK2ZFX2.....ACT_ONE Generation : 0000001
11CCCCDCEC4CEDFECECF44068A00CCE6DDC44444444444
A172924981B98229672BBB35A01133D6550000000000
```

Eventpool

```

Event : (Reattach)
Type : Activity
Fired : No
Reattach : Yes
Retrieve : No
Subevent : No
```

Figure 87. Example output from the DFHBARUP utility (Part 2)

```
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 88. 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  
 11CCCDCEC4CEDFECEF44068A00CE6DDC4444444444  
 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 *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.IYCWT37...<....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 4003C2C1 D4C1E4C4 C9E30000 * .....BAMAUDIT..*
000180 00000000 0000FFFF FFFFFFFF *.....*
  
```

Related BTS Objects

Proce Parent Name : DFHROOT Id : ..GBIBMIYA.IYCWT37.....DFHROOT Generation : 0000001  
 11CCCDCEC4CECECF44042F00CCDDDE4444444444  
 A172924981B98363337BB0C1B0146896630000000000

No Children

Eventpool

Event : (Reattach)  
 Type : Activity  
 Fired : No  
 Reattach : Yes  
 Retrieve : No  
 Subevent : No

Figure 89. Example output from the DFHBARUP utility (Part 4)

```

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 10101010 10          *.....*
    
```

Figure 90. 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 *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.IYCWT37...<...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..ABU*
000140 F8F1E940 00000000 00000000 D9E4D7E9 C3C9C3E2 E4E2C5D9 C1C3E36D F3404040 *81Z .....RUPZCICSUSERACT_3 *
000160 40404040 40404040 01404040 40404040 4003C2C1 D4C1E4C4 C9E30000 * .....BAMAUDIT..*
000180 00000000 00000000 00000000 *.....*
```

Related BTS Objects

Process Type : SALES Name : SALES1234567890  
 Parent Name : DFHROOT Id : ..GBIBMIYA.IYCWT37.....DFHROOT Generation : 0000001  
 11CCCCDCEC4CECECF44042F00CCDDDE444444444444  
 A172924981B98363337BB0C1B01468966300000000000

No Children

Eventpool

Event : (Reattach)  
 Type : Activity  
 Fired : No  
 Reattach : Yes  
 Retrieve : No  
 Subevent : No

Figure 91. Example output from the DFHBARUP utility (Part 6)

```
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 92. 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
11CCCCDCEC4CECEECFF44042F00CCDDDE4444444444
A172924981B98363337BB0C1B0146896630000000000
Audit Level : Full (Off, Pro, Act, Full)
Audit Log   : BMAUDIT
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 *
0000A0 00000000 00000000 00000000 00000000 00000000 108F90AC 108F90AC 00000000 *.....?B.....dz.....*
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 *.....*

```

Figure 93. Example output from the DFHBARUP utility (Part 8)

```
000000  22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 *.....*  
      .  
      lines omitted  
      .  
0003E0  22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 22222222 *.....*  
Container Name : Container_three Container Length : x'00000019'
```

```
000000  10101010 10101010 10101010 10101010 10101010 10101010 10101010 10          *.....*
```

Figure 94. 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, trace, and dump

BTS messages are identified by the following prefixes:

- DFHBA
- DFHEM
- DFHSH

All CICS messages, including BTS messages, are listed in the *CICS Messages and Codes* manual. To discover the meaning of a particular message, refer to that book.

---

### BTS messages

BTS messages are identified by the following prefixes:

- DFHBA
- DFHEM
- DFHSH

All CICS messages, including BTS messages, are listed in the *CICS Messages and Codes* manual. To discover the meaning of a particular message, refer to that book.

---

### Setting trace levels for BTS

#### About this task

BTS consists of three CICS domains:

Domain name	CICS Component code
Business application manager	BA
Event manager	EM
Scheduler services	SH

You can use the CICS component codes to specify the level of standard and special tracing for BTS. 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 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*.

## Defining tracing levels when CICS is running

You can use the CETR transaction to define, dynamically on the running CICS system, the level of tracing required for BTS.

Figure 95 shows you what the CETR Component Trace Options screen looks like. To make changes, you overtype the settings shown on the screen, and then press ENTER.

CETR		Component Trace Options	
Overtyping where required and press ENTER.			
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	

PAGE 1 OF 2

PF: 1=Help 3=Quit 7=Back 8=Forward 9=Messages ENTER=Change

Figure 95. 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*.

## The BTS trace points

BTS trace points are listed in the *CICS Trace Entries* manual.



---

## Extracting 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*.



---

## Chapter 19. Tuning BTS performance

You can use the monitoring data produced by CICS to 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*.



---

## Part 5. BTS Application Programming Reference

This part of the manual contains programming information about the CICS business transaction services application programming interface (API).

*Table 13. API reference road map*

<b>If you want to...</b>	<b>Refer to...</b>
Understand how the BTS API commands fall into functional groups	<i>Overview of BTS API commands</i>
Check the syntax of an API command	<i>BTS application programming commands</i>



---

## Chapter 20. Overview of BTS API commands

This chapter introduces the CICS business transaction services application programming interface (API) commands. It contains:

- Process- and activity-related commands
- Container commands
- Event-related commands
- Browsing and inquiry commands
- System events

This chapter groups the API commands by function, giving a brief overview of what each can be used for. For an alphabetical listing of the commands, or for detailed programming information, see *BTS application programming commands*.

---

### Process- and activity-related commands

This section describes the CICS business transaction services commands that relate to processes and activities.

#### Creating, activating, and terminating 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.

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 retrying the process.

**RESET ACTIVITY**

Resets an activity to its initial state—used before retrying an activity.

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

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 205.

## Relating UOWs and activities

Use this 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

The CICS business transaction services commands that act 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 already exist, it is created. If the named container already 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

This section describes the CICS business transaction services event-related commands.

### Terminology

This section is a brief reminder of some of the terms used to describe 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 *sub-event*. Sub-events that fire are placed on the composite event's *sub-event queue*. Each composite event has a sub-event queue associated with it. The sub-event queue:

- May be empty
- Contains only those sub-events that have fired and not been retrieved.

## Reattachment events

An event that fires, and thereby causes an activity to be reattached, is known as a *reattachment event*. All user-defined events except sub-events 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 sub-events) are potentially reattachment events. All system events are reattachment events.

At times, reattachment may occur because of the firing of more than one event. Reattachment events are placed on the activity's *reattachment queue*, 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

All the event-related commands operate on the *current* activity's event pool.

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 timer's associated event 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 sub-event to a composite event.

**REMOVE SUBEVENT**

Removes a sub-event from a composite event.

**DELETE EVENT**

Removes an input or composite event from the current activity's event pool.

**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 sub-event in a composite event's sub-event queue, and resets the retrieved sub-event's fire status 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 should not need to inquire on or browse the objects it creates, and therefore will not use these commands.

CICS business transaction services provide a set of commands that enable programs to search for and examine BTS objects. These commands can be summarized as:

**Browsing commands**

Used to locate BTS objects, and to retrieve some or all of their attributes.

**Inquiry commands**

Used to retrieve the attributes of specific BTS objects (which may have been located by browsing).

The object-identifiers and names retrieved from the browsing or inquiry commands can be specified on a subset of the other BTS API commands. This allows actions to be started against specified activities, processes, and data-containers.

## Browsing commands

The browsing commands allow BTS objects to be located and their relationships to each other examined. The objects that can be browsed are:

- 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 23, “BTS system programming commands,” on page 219.

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 to be 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 of the child activity's containers, for instance.

### ENDBROWSE

Ends the browse.

## Inquiry commands

A program can get details of a specific BTS object by issuing an INQUIRE command. You can inquire on:

- Activities
- Data-containers
- Events
- Processes
- Process-types
- Timers.

The object inquired upon may 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 may 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 may be modified by the same program. The INQUIRE command always goes to the repository for the record it needs, and may 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. The same token returned on a STARTBROWSE command must be supplied on the corresponding GETNEXT and ENDBROWSE commands. CICS discards it after the ENDBROWSE.

The lifetime of a browse token is from a STARTBROWSE to an ENDBROWSE or syncpoint, 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 syncpoint is encountered before the browse has completed

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. Once 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 which take identifiers returned by browse operations.” The lifetime of an activity identifier is the same as that of the activity it refers to. Thus, it can be used after an ENDBROWSE and after a syncpoint.

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 which take identifiers returned by browse operations

It is essential that the systems programmer should be able to modify a business transaction after it has started. This is particularly important if the transaction gets into a state where 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

This section contains some examples of how the browsing and inquiry commands can be used.

### Example 1

An application, which has not issued any requests to BTS, wants to see if a particular container belongs to a child of the root activity of a particular process, whose name and type 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 96. Browsing example 1*

## Example 2

An application, which has not issued any requests to BTS, wants to know whether a particular data-container is one of the global containers associated with a particular process. If it is not, the program wants 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 97. Browsing example 2*

### **Example 3**

A program running as an activation of an activity wants 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's event pools. If the event exists, the program wants 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 this to browse the child's events.

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 98. Browsing example 3

---

## System events

BTS produces system events as a result of its own processing. BTS system events are identified by the prefix 'DFH'. Currently, there is only one type of system event:

### **DFHINITIAL**

The activity is being attached for the first time in this process, *or* it is being retried after being reset with a `RESET ACTIVITY` command. An activity must be coded to cope with this event, which tells it that it should perform any initial housekeeping.



---

## Chapter 21. BTS application programming commands

The following CICS API commands support BTS.

**Restriction:** None of these commands is threadsafe.



---

## Part 6. BTS System Programming Reference

This part of the manual contains detailed programming information about the CICS business transaction services system programming commands.

*Table 14. System programming reference road map*

If you want to...	Refer to...
Understand how the BTS system programming commands fall into functional groups	Chapter 22, "Overview of BTS system programming commands," on page 217
Check the syntax of a system programming command	Chapter 23, "BTS system programming commands," on page 219



---

## Chapter 22. Overview of BTS system programming commands

This chapter introduces the CICS business transaction services system programming commands. It contains:

- “Control commands”
- “Inquiry 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 alphabetical listing of the commands, or for detailed programming information, see Chapter 23, “BTS system programming commands,” on page 219.

---

### Control commands

CICS business transaction services provide a set of commands that enable 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.

---

### Inquiry 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 205.







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

Remove a PROCESSTYPE definition.

### DISCARD PROCESSTYPE

►►—DISCARD—PROCESSTYPE—(*data-value*)—◄◄

**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.

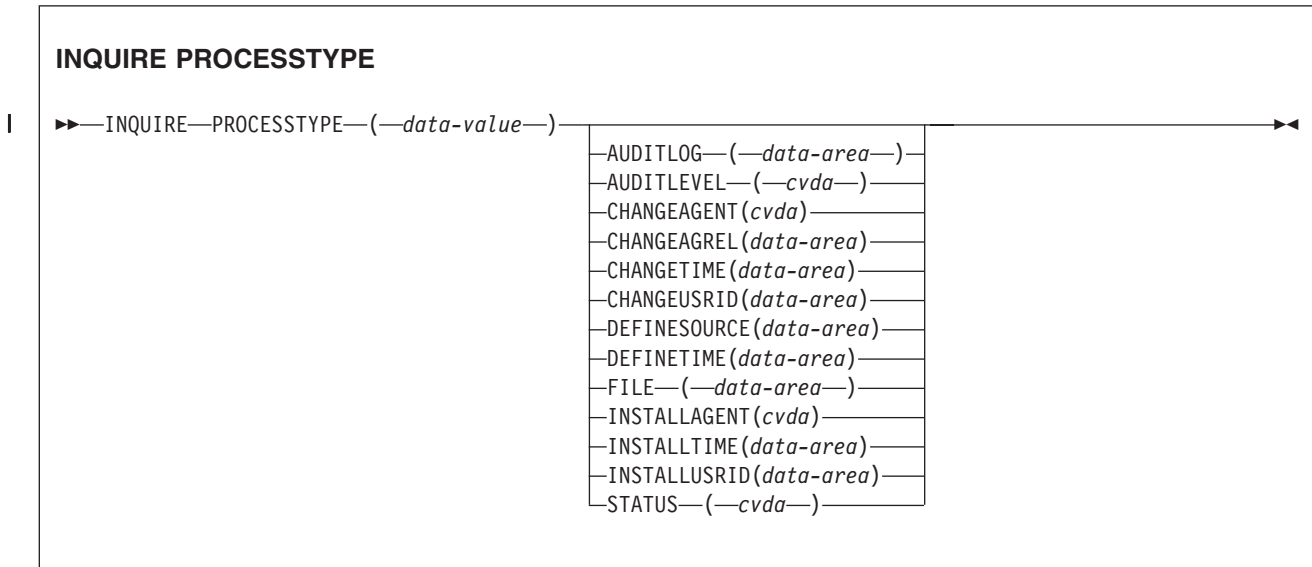
##### PROCESSERR

RESP2 values:

- 1 The process-type named in the PROCESSTYPE option is not defined in the process-type table (PTT).

# INQUIRE PROCESSTYPE

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 display the resource signature fields, which you can use to manage resources by capturing details of when the resource is defined, installed, and last changed. For more information, see in the *CICS Resource Definition Guide*. The resource signature fields are CHANGEAGENT, CHANGEAGREL, CHANGETIME, CHANGEUSRID, DEFINESOURCE, DEFINETIME, INSTALLAGENT, INSTALLTIME, and INSTALLUSRID. See the *CICS 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:

#### 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 162.

**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 identifying 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.

**CHANGEUSRID(*data-area*)**

Returns the 8-character user ID that ran the change agent.

**DEFINESOURCE(*data-area*)**

Returns the source of the resource definition. The DEFINESOURCE value depends on the CHANGEAGENT. For details, see the *CICS 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 identifying the agent that installed the resource. The possible values are as follows:

**CREATESPI**

The resource was installed by an EXEC CICS CREATE command.

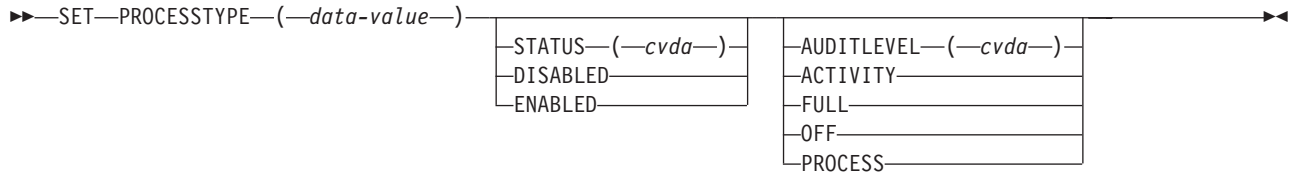


---

## SET PROCESSTYPE

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 162.

**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



---

## 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. 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 keep track of 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 are supplied, in source code, in the SDFHSAMP library, together with copybooks and BMS maps and mapsets. 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 could use to create a repository, see Figure 52 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 15 shows the transactions and programs that make up the sample application.

*Table 15. Transactions and programs in the BTS sample application*

Transaction	Program	Function
SALM	DFH0SAL0	Order entry. This 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: <ul style="list-style-type: none"><li>• Customer credit-check, run synchronously with the root activity</li><li>• Stock check, run synchronously</li><li>• Delivery note, run asynchronously</li><li>• Invoice, run asynchronously</li><li>• Payment-not-received reminder, run asynchronously (if payment is not received before the timer expires).</li></ul>
RED1	DFH0RED1	Implements the customer credit-check activity.
STOC	DFH0STOC	Implements the stock-level check activity.
DEL1	DFH0DEL1	Implements the delivery note activity.
INV1	DFH0INV1	Implements the invoice activity.
REM1	DFH0REM1	Implements the payment-not-received reminder activity.

Table 15. Transactions and programs in the BTS sample application (continued)

Transaction	Program	Function
PAYM	DFH0PAY0	Payment input. This 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 16 shows the copybooks, maps and mapsets supplied with the sample application.

Table 16. Copybooks, maps and mapsets supplied with the BTS sample application

Type	Module	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 mapset is DFH0SAS. The name of the map is ORDER.
Map/Mapset	DFH0PAYM	Source of the BMS map for the PAYM transaction. The mapset 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 need to 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 will need to 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 source code of the DFH0SAL0 program contains detailed instructions on how to 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.

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](http://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](http://www.ibm.com/legal/copytrade.shtml).

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

---

## Bibliography

---

### CICS books for CICS Transaction Server for z/OS

#### General

*CICS Transaction Server for z/OS Program Directory*, GI13-0536  
*CICS Transaction Server for z/OS What's New*, GC34-6994  
*CICS Transaction Server for z/OS Upgrading from CICS TS Version 2.3*, GC34-6996  
*CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.1*, GC34-6997  
*CICS Transaction Server for z/OS Upgrading from CICS TS Version 3.2*, GC34-6998  
*CICS Transaction Server for z/OS Installation Guide*, GC34-6995

#### Access to CICS

*CICS Internet Guide*, SC34-7021  
*CICS Web Services Guide*, SC34-7020

#### Administration

*CICS System Definition Guide*, SC34-6999  
*CICS Customization Guide*, SC34-7001  
*CICS Resource Definition Guide*, SC34-7000  
*CICS Operations and Utilities Guide*, SC34-7002  
*CICS RACF Security Guide*, SC34-7003  
*CICS Supplied Transactions*, SC34-7004

#### Programming

*CICS Application Programming Guide*, SC34-7022  
*CICS Application Programming Reference*, SC34-7023  
*CICS System Programming Reference*, SC34-7024  
*CICS Front End Programming Interface User's Guide*, SC34-7027  
*CICS C++ OO Class Libraries*, SC34-7026  
*CICS Distributed Transaction Programming Guide*, SC34-7028  
*CICS Business Transaction Services*, SC34-7029  
*Java Applications in CICS*, SC34-7025

#### Diagnosis

*CICS Problem Determination Guide*, GC34-7034  
*CICS Performance Guide*, SC34-7033  
*CICS Messages and Codes*, SC34-7035  
*CICS Diagnosis Reference*, GC34-7038  
*CICS Recovery and Restart Guide*, SC34-7012  
*CICS Data Areas*, GC34-7014  
*CICS Trace Entries*, SC34-7013  
*CICS Supplementary Data Areas*, GC34-7015  
*CICS Debugging Tools Interfaces Reference*, GC34-7039

#### Communication

*CICS Intercommunication Guide*, SC34-7018  
*CICS External Interfaces Guide*, SC34-7019

#### Databases

*CICS DB2 Guide*, SC34-7011  
*CICS IMS Database Control Guide*, SC34-7016

## CICSplex SM books for CICS Transaction Server for z/OS

### General

*CICSplex SM Concepts and Planning*, SC34-7044  
*CICSplex SM Web User Interface Guide*, SC34-7045

### Administration and Management

*CICSplex SM Administration*, SC34-7005  
*CICSplex SM Operations Views Reference*, SC34-7006  
*CICSplex SM Monitor Views Reference*, SC34-7007  
*CICSplex SM Managing Workloads*, SC34-7008  
*CICSplex SM Managing Resource Usage*, SC34-7009  
*CICSplex SM Managing Business Applications*, SC34-7010

### Programming

*CICSplex SM Application Programming Guide*, SC34-7030  
*CICSplex SM Application Programming Reference*, SC34-7031

### Diagnosis

*CICSplex SM Resource Tables Reference*, SC34-7032  
*CICSplex SM Messages and Codes*, GC34-7035  
*CICSplex SM Problem Determination*, GC34-7037

---

## 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 1.

*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.



---

# Index

## Numerics

- 3270 bridge support
  - conversational transactions 101
  - introduction 10, 99
  - processing considerations 105
  - pseudoconversational transactions 103
  - resource definition 100
  - running a 3270 transaction 99
  - sample programs 106

## A

- abends, of activities 158
- acquiring a process 61
- acquiring an activity 61, 70
- activities
  - abends 158
  - acquiring access to 61
  - activation of 17
  - asynchronous 17
    - checking response from 18
  - auditing of
    - introduction 161
    - specifying the logging level 162
  - browsing with CBAM 125
  - checking response from
    - asynchronous 51
    - synchronous 49
  - child 15
  - compared with terminal-related pseudoconversations 17
  - compensation 85
  - completion event 22
  - data-containers 20
  - described 15
  - destruction of 18, 44
  - identifiers 207
  - implementation by existing 3270-based transactions 99
  - input event 22
  - 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 19
  - transferring input and output data 44
  - unserviceable requests 158
- activity abends 158
- activity completion event 22
- activity identifiers
  - described 207
- activity-related commands
  - INQUIRE PROCESSTYPE 222

- activity-related commands (*continued*)
    - overview 201
  - administration
    - controlling BTS
      - operator commands 125
      - resource definition 117
      - security 123
      - sysplex considerations
        - dealing with affinities 152
        - using CPSM 151
      - system definition
        - defining local request queue data set 113
        - defining repository data sets 111
        - naming the distributed routing program 115
  - affinities, in a sysplex 152
  - ALL
    - CEMT INQUIRE TASK 137
  - API commands
    - activity-related 201
    - browse tokens 207
    - browsing commands 206
    - container 202
    - event-related 203
    - examples 208
    - inquiry commands 206
    - overview 201
    - that take activity identifiers 207
  - asynchronous activities
    - checking response from 51
    - how invoked 17
  - atomic event
    - activity completion 22
    - input 22
    - system 22
    - timer 22
  - attach-time security 124
  - audit commands
    - introduction to 161
  - audit trail
    - examples 165
    - introduction to 161
    - sharing a logstream between CICS regions 164
    - specifying the logging level 162
    - utility program, DFHATUP 168
  - audit trail utility program, DFHATUP 168
  - AUDITLEVEL attribute
    - PROCESSTYPE definition 120
  - AUDITLOG attribute
    - PROCESSTYPE definition 121
- ## B
- browse tokens 207
  - browsing commands
    - CBAM 125
    - INQUIRE PROCESSTYPE 222
  - BTS messages 193

- BTS sample application 229
- BTS-set
  - dealing with affinities 152
  - how to create 146
  - introduction to 141
  - scope of 141
- business transaction
  - described 3
  - Sale example application 33
  - sample BTS application 229

## C

- CBAM, CICS-supplied transaction 125
- CEDA DEFINE PROCESSTYPE
  - command 119
- CEMT INQUIRE PROCESSTYPE
  - ALL 134
  - AUDITLEVEL 134
  - AUDITLOG 135
  - FILE 135
  - STATUS 136
  - value 134
- CEMT INQUIRE TASK
  - ACTIVITY 138
  - ACTIVITYID 138
  - PROCESS 138
  - PROCESSTYPE 138
- CEMT SET PROCESSTYPE
  - (value) 139
  - ACTIVITY 139
  - ALL 139
  - DISABLED 139
  - ENABLED 139
  - FULL 139
  - OFF 139
  - PROCESS 139
- CEMT transaction
  - PROCESSTYPE 133, 138
  - TASK 136
- child activity
  - described 15
- CICS business transaction services
  - 3270 bridge support 99
  - administration
    - controlling BTS 125
    - defining local request queue data set 113
    - defining repository data sets 111
    - performance tuning 197
    - resource definition 117
    - security 123
    - sysplex considerations 141
    - system definition 111
  - browsing BTS objects 125
  - client/server processing 10
  - components
    - activities 15
    - data-containers 20
    - events 21
    - introduction to 7

- CICS business transaction services
  - (*continued*)
  - components (*continued*)
    - processes 15
  - external interactions
    - acquiring activities 61
    - acquiring an activity 70
    - acquiring processes 61
    - client/server processing 63
  - introduction to 7
  - monitoring 197
  - parallel activities 53
  - problem determination 155
  - recovery and restart 10
  - reusing existing code 99
  - sample application 229
  - security
    - attach-time 124
    - command-level 124
    - resource-level 123
  - sysplex support 11
  - user-related activities 71
  - Web Interface support 10
- CICS-supplied transactions
  - CBAM 125
  - CEMT INQUIRE PROCESSTYPE 133
  - CEMT INQUIRE TASK 136
  - CEMT SET PROCESSTYPE 138
- CICSPlex SM
  - use with BTS 11, 151
- client/server processing
  - example 63
  - introduction 10, 63
- cold start, of CICS 160
- 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 22
- conditions
  - CREATE PROCESSTYPE
    - command 220
  - DISCARD PROCESSTYPE
    - command 221
  - INQUIRE PROCESSTYPE
    - command 224
  - SET PROCESSTYPE command 226
- container commands
  - overview 202
- controlling BTS
  - operator commands 125
- CPSM, use with BTS 151
- CREATE PROCESSTYPE command 219
  - conditions 220

## D

- DASD-only logstreams, restrictions on
  - sharing 164

- 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
- 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 197
- DFH0CBAC, sample client activity
  - program for 3270 bridge 106
- DFH0CBAE, sample bridge exit
  - program 106
- DFHATUP, audit trail utility
  - program 168
- DFHBARUP, repository utility
  - program 181
- DFHDSRP, distributed routing
  - program 115
    - how to write 149
    - relation to dynamic routing
      - program 148
- DFHINITIAL system event 211
- DISCARD PROCESSTYPE
  - command 221
    - conditions 221
- distributed routing
  - introduction to 143
    - of BTS activities 148, 149
      - creating a BTS-set 146
      - which activities can be dynamically routed? 142
    - routing program, DFHDSRP 148, 149
  - distributed routing program, DFHDSRP 115
    - how to write 149
    - relation to dynamic routing
      - program 148
- DSRTPGM, system initialization
  - parameter 115
- dump formatting keywords, for BTS 195
- dynamic routing
  - of BTS activities
    - creating a BTS-set 146
    - naming the distributed routing
      - program 115
    - using a distributed routing
      - program 148
    - using CPSM 151
    - which activities can be dynamically routed? 142

## E

- emergency restart, of CICS 160
- errors
  - checking response from asynchronous
    - activities 51
  - checking response from synchronous
    - activities 49
- event
  - atomic
    - activity completion 22
    - input 22
    - system 22
    - timer 22
  - composite 22
  - deleting 25
  - described 21
    - reattaching an activity on firing of 26
- event-related commands
  - overview 203
- examples
  - API commands 208
  - attach-time security 124
  - audit trails 165
  - basic Sale application
    - data flow 33
    - initial request 35
    - overview 33
    - root activity 38
    - transferring data to activities 44
  - browsing 208
  - client/server processing
    - client program 63
    - server program 67
  - compensation 86
  - error handling 51
  - output from DFHATUP 171, 172
  - output from DFHBARUP 184
  - 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 101
    - introduction 10, 99
    - processing considerations 105
    - pseudoconversational
      - transactions 103
    - resource definition 100
    - running a 3270 transaction 99
    - sample programs 106

## F

- failure, of CICS 160

FILE attribute  
PROCESSTYPE definition 121

## I

initial request  
in basic Sale application 35  
initial start, of CICS 160  
input and output data, transfer of 44  
input event 22  
INQUIRE PROCESSTYPE command 222  
conditions 224  
interacting with non-BTS code  
acquiring activities 61  
acquiring an activity 70  
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 193  
modes, of an activity  
described 18  
monitoring  
performance class data 197  
using DFH\$MOLS 197  
monitoring data print program,  
DFH\$MOLS 197

## N

notation, syntax ix

## O

operator commands  
CBAM 125  
CEMT INQUIRE PROCESSTYPE 133  
CEMT INQUIRE TASK 136  
CEMT SET PROCESSTYPE 138

## P

parallel activities  
data flow 53  
example 54  
introduction 53  
parent activity  
described 15  
transferring data to child  
activities 44  
performance class monitoring data 197  
performance tuning  
introduction 197  
problem determination  
activity abends 158

problem determination (*continued*)  
BTS-related messages 193  
CICS failures  
cold starts 160  
emergency starts 160  
initial starts 160  
dump formatting keywords 195  
examining repository records  
utility program, DFHBARUP 181  
introduction to 155  
stuck processes  
due to application errors 155  
due to unserviceable requests 158  
trace levels 193  
unserviceable requests 158  
using an audit trail  
examples 165  
introduction 161  
sharing a logstream between CICS  
regions 164  
specifying the logging level 162  
utility program, DFHATUP 168

process  
acquiring access to 61  
auditing of  
introduction 161  
specifying the logging level 162  
browsing with CBAM 125  
categorizing 15  
data-containers 20  
described 15  
identifier 207  
security of 123, 124  
stuck 155  
unserviceable requests 158  
process identifiers 207  
process-type  
browsing with CBAM 125  
CBAM requests 125  
CEMT INQUIRE requests 133  
CEMT SET requests 138

PROCESSTYPE attribute  
PROCESSTYPE definition 121  
PROCESSTYPE command  
CEMT INQUIRE transaction 133  
CEMT SET transaction 138  
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 181  
utility program, DFHBARUP 181  
repository utility program,  
DFHBARUP 181  
resource definition  
defining BTS resources to CICS 117  
RDO commands  
CEDA DEFINE  
PROCESSTYPE 119  
resource-control commands  
CREATE PROCESSTYPE 219  
DISCARD PROCESSTYPE 221  
SET PROCESSTYPE 225  
resource-level security 123  
reusing existing code  
3270 bridge support  
conversational transactions 101  
introduction 10, 99  
processing considerations 105  
pseudoconversational  
transactions 103  
resource definition 100  
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 158

## 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 71  
sample BTS application 229  
sample programs  
BTS application 229  
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  
resource-level 123

- SET PROCESSTYPE command 225
  - conditions 226
- sharing an audit logstream between CICS regions 164
- SPCTR, system initialization parameter 193
- special trace, setting the level of 193
- standard trace, setting the level of 193
- STATUS attribute
  - PROCESSTYPE definition 121
- stuck processes 155
- surrogate security checking 123
- synchronous activities
  - checking response from 49
  - how invoked 17
- syncpoint
  - user, issued by an activity 19
- syntax notation ix
- sysplex considerations
  - BTS's sysplex support 11, 141
  - dealing with affinities 152
  - introduction 11
  - using CPSM 151
- system definition 111
- system events
  - described 22
  - DFHINITIAL 211
- system initialization parameters
  - DSRTPGM 115
  - SPCTR 193
  - SPCTRBA 193
  - STNTR 193
  - STNTRBA 193
- system programming commands
  - control commands 217
  - CREATE PROCESSTYPE 219
  - DISCARD PROCESSTYPE 221
  - INQUIRE PROCESSTYPE 222
  - inquiry command 217
  - overview 217
  - SET PROCESSTYPE 225

## T

- task
  - CEMT INQUIRE requests 136
- TASK command
  - CEMT INQUIRE transaction 136
- TCLASS
  - CEMT INQUIRE TASK 137
- terminal-related pseudoconversation
  - comparison with an activity that is activated multiple times 17
- timer
  - described 20
- timer event
  - described 22
- trace
  - special, setting the level of 193
  - standard, setting the level of 193
- trademarks 232
- transaction affinities, in a sysplex 152
- transferring data to child activities 44

## U

- unserviceable requests 158
- user-related activities
  - example 71
- utility programs
  - audit trail utility, DFHATUP 157, 168
  - repository utility, DFHBARUP 157, 181

## V

- value
  - CEMT INQUIRE TASK 137

## W

- Web Interface
  - introduction 10

---

## Readers' Comments — We'd Like to Hear from You

CICS Transaction Server for z/OS  
Version 4 Release 1  
Business Transaction Services

Publication No. SC34-7029-01

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 e-mail to: [idrpf@uk.ibm.com](mailto:idrpf@uk.ibm.com)

If you would like a response from IBM, please fill in the following information:

\_\_\_\_\_

Name

\_\_\_\_\_

Address

\_\_\_\_\_

Company or Organization

\_\_\_\_\_

Phone No.

\_\_\_\_\_

E-mail address



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







SC34-7029-01

