COVER Book Cover

CICS

Application Programming Primer

Document Number SC33-0674-01

Program Number
5685-083

ABSTRACT Abstract

This book is intended for application programmers who are new to CICS, but it is also useful for new system programmers. Before reading this primer you should have some knowledge of programming and in a batch environment. This book tells you enough to be able to design, code, test and run your first CICS application programs. It describes a subset of the full CICS product and illustrates CICS facilities and useful techniques by a realistic example coded in VS COBOL II.

EDITION Edition Notice
First Edition (June 1990)

This edition applies to Version 3 Release 1 Modification 1 of the IBM licensed program Customer Information Control System/Enterprise Systems Architecture (CICS/ESA), program number 5685-083, Version 2 Release 1 and Version 2 Release 1 Modification 1 of the Customer Information Control System/Multiple Virtual Storage (CICS/MVS), program number 5665-403; Version 2 Release 1 of the Customer Information Control System/Virtual Storage Extended (CICS/VSE), program number 5686-026, and to all subsequent versions, releases, and modifications until otherwise indicated in new editions.

This is the softcopy version of the printed version of the *Application Programming Primer (VS COBOL II)*. Minor changes to formatting have been made to make the information more suitable for viewing online.

Changes made since this book was last published are indicated by the hash (#) symbol to the left of the changes.

Consult the latest edition of the applicable IBM system bibliography for current information on this product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the addresses given below.

Reader's comments on this publication should be addressed to:

International Business Machines Corporation, Attn: Dept ACV-H
1001 Wt Harris Blvd, Charlotte, NC 28257-0001, USA

or to:

IBM United Kingdom Laboratories Limited, Information Development,
Mail Point 095, Hursley Park, Winchester, Hampshire, England, SO21 2JN.

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

CONTENTS Table of Contents

FIGURES Figures

TABLES Tables

FRONT_1 Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

In this publication are illustrations in which names are used. These names are used solely for illustrative purposes and not for the identification of any person or company.

This book contains sample programs. Permission is hereby granted to copy and store the sample programs into a data processing machine and to use the stored copies for study and instruction only. No permission is granted to use the sample program for any other purpose.

The following terms, denoted by an asterisk (*), used in this publication, are trademarks or service marks of IBM Corporation in the United States or other countries:

CICS/ESA, CICS/MVS, IBM, IMS/ESA, RACF

The following terms, denoted by a double asterisk (**), used in this publication, are trademarks of other companies as follows:

Teletype	Teletype Corporation

PREFACE Preface

This book is intended to help you to write CICS application programs, using the command-level CICS interface and the COBOL programming language.

We assume you're an application programmer, bringing to CICS your existing knowledge of COBOL gained in a batch programming environment. However, experience of other (non-CICS) online systems, and of other high-level programming languages (such as PL/I or C), will be helpful.

It contains guidance about designing, coding, testing and running your first CICS application program. We want to point you to the various books in the CICS library that will fill in the gaps because, in a book this size, we won't be able to tell you **all** about CICS. The information in the book is not part of the definition of any programming interface for customers. It must not be used for programming purposes.

We'll be talking about, and basing our examples on, a subset of the full CICS facilities. This makes things easier for you because it means we won't have to keep referring you to other books in the CICS library while you're learning. These other books are shown in the library diagram for your particular release of CICS.

The subset of CICS commands we've chosen is as complete and self-sufficient as we can make it. It will give you a sound framework for your first application programs, and offer a logical starting point for more advanced work.

Changes since the first edition

Since its first edition, when CICS/VS Version 1 Release 6 was current, there's been a major innovation of direct concern to all CICS application programmers. This is the **RESP** option, which you can add to any **EXEC CICS** command. **RESP** deals with the exceptional conditions that CICS raises when things go wrong with CICS commands. By adding **RESP** to a command, you can immediately test for any condition that concerns you and decide, then and there, what to do next. This makes it much easier to write clearly-structured code.

What we've done to the **ACCT** programs that form the example application mainly reflects this new ability. We've moved almost completely away from the use of **HANDLE CONDITION** commands, adopting the more structured approach that **RESP** encourages. (We've also converted to VS COBOL II code. We've made some minor structural changes by moving sections of related processing to the end of the code and using **PERFORM** statements. This helps clarify the underlying logic of the application.)

There are two editions of this *Application Programming Primer*. The original edition (SC33-0139) contains COBOL source code that will run under CICS/DOS/VS, CICS/OS/VS, and CICS/MVS (*). This edition (SC33-0674) contains VS COBOL II source code that will run under CICS/ESA (*).

How to use this book

Read through it at your own pace until you reach "The COBOL code of our example application" in topic 4.0. At that point, you meet the VS COBOL II source code of our example application. (It's supplied in machine readable form on the CICS distribution tape in the **CICS330.SAMPLIB** library.)

Study the code. Run the application. Think how you would improve it (this might not be as difficult as you imagine!). Make your changes and try them out. Remember: "I read, and I remember; I do, and I understand."

A note on installing your CICS system

Before you can test your application, you need a CICS system on which to run it. We tell you where to find out about installing a CICS system in Appendix A, "Getting the application into your CICS system" in topic A.0.

By referring to this appendix, and with the help of a friendly system programmer, you should end up with a working CICS system on which you can install and run your first application program.

Product names

Throughout the book we've used the simple, commonly used, abbreviations for the names of IBM program products. If you want to know exactly what these abbreviations mean, look in the glossary at the back.

Computer systems

CICS runs on a wide range of IBM computer systems. Since its first edition, when CICS/VS application programmer, you can assume for the time being that the information here applies to all these systems. However, where we need to make an assumption about the computer system that you are using, we assume a relatively small system, running under MVS.

Terminals

We'll be assuming 3270 Information Display System terminals are used for the example application in this book.

(*) IBM Trademark. For a complete list of trademarks See "Notices" in topic FRONT_1.

Subtopics
PREFACE.1 Book structure

PREFACE.1 Book structure

"Setting the scene" in topic 1.0
    introduces CICS, and tries to answer the question "What's different about CICS?" (compared with, say, a batch system).

"Application design" in topic 2.0
    deals with application design from various angles: the user interface, the design of the data, the splitting of the processing steps into sensible transactions, the exercise of control and communication between transactions, and so on.

"Application programming" in topic 3.0
    tells you how to write the COBOL programs that will implement the CICS example file inquiry and update application. These programs form a realistic (non-trivial) working system.

"The COBOL code of our example application" in topic 4.0
    contains the source code in full, and detailed step-by-step notes of how it works.

"Testing and diagnosis" in topic 5.0
    covers running, testing, and debugging application programs. It shows you a complete debugging session using the powerful facilities of the Execution Diagnostic Facility (EDF). (The bug is one we deliberately added, in case you're wondering...)

    It also shows you how to work through a transaction dump of the same problem, arriving at the same conclusion.

Appendix A, "Getting the application into your CICS system" in topic A.0
    tells you where to find out how to install and bring up a CICS system with the example application.

Appendix B, "Other CICS facilities" in topic B.0
    tells you about the various features of CICS that we've not been able to cover in a book this size. It also introduces you to the three other application programming books you'll need when you start writing your own programs: the CICS/ESA Application Programming Guide , the CICS/ESA Problem Determination Guide which contain guidance information, and the CICS/ESA Application Programming Reference , which contains definitive application programming interface information.

"Glossary" in topic GLOSSARY
    defines special CICS terms used in the library and words used with other than their everyday meaning.

1.0 Setting the scene

This part of the Primer:

- Describes the ideas behind CICS
- Explains some of the CICS terminology
- Describes a typical online application program.

Subtopics

1.1 Introduction to CICS

1.1 Introduction to CICS

Subtopics
1.1.1 What is CICS?
1.1.2 Why have CICS?
1.1.3 What does CICS do?
1.1.4 How does a CICS-based application differ from a batch application?
1.1.5 How does CICS help you set up an online system?
1.1.6 How do you use CICS?

1.1.1 What is CICS?

CICS (Customer Information Control System) is a general-purpose data communication system that can support a network of many hundreds of terminals. You may find it helpful to think of CICS as an operating system within your own operating system (although this definition might offend purists). In these terms, CICS is a specialized operating system whose job is to provide an environment for the execution of your online application programs, including interfaces to files and database products. See Figure 1.

PICTURE 1

Figure 1. The CICS online environment

The total system is known as a **database/data-communication system**, but this is such a mouthful that we usually shorten it to **DB/DC system**.

Your host operating system, of course, is still the final interface with the computer; CICS is "merely" another interface, this time with the operating system itself.

Operating systems are designed to make the best use of the computer's various resources. CICS helps out by separating a particular kind of application program (namely, online applications) from others in the system, and handling these programs itself.

Subtopics
1.1.1.1 Why you may need an online system

1.1.1.1 Why you may need an online system

If you're the sort of person we've imagined as a typical reader, until now you've written programs that (typically) read a file, process individual data records, update a carried-forward version of the file, and produce some type of printed output. These files usually go offline when your program has finished with them, and the file data thus becomes inaccessible for inquiry purposes. Furthermore, the records in the files are only as up-to-date as the most recent program run, and don't reflect any intervening activity.

Nowadays, this often isn't good enough. Your users want immediate responses to their information processing needs. The overnight turnaround associated with traditional systems is no longer adequate: accurate, up-to-date information is needed within seconds. To achieve this you need an online information processing system, using terminals that can give direct access to data held in either data sets or databases. In other words, you need a DB/DC system.

Developing a DB/DC system can be a major undertaking, particularly if you choose to write all your own control programs for handling terminals and files, and provide your own job-scheduling mechanisms. However, CICS can make it very much easier by supplying all the basic components needed to handle your data communications. This allows you to concentrate on developing application programs to meet your organization's business needs. You don't need to concern yourself with the details of data transmission, buffer handling, or the properties of individual terminal devices.

PICTURE 2

Figure 2. A DB/DC system

1.1.2 Why have CICS?

The online end users within a network can make all sorts of demands on many different sets of data. The things they want to do individually are usually short. Often they are interrelated and share the same programs and data. Furthermore, the response times they get should be as short as possible. For all these reasons, the users' transactions are done more efficiently within a single operating system job, rather than as separate jobs.

If all the transactions are to be handled within the same job, a controller is needed to look after them, in much the same way that an operating system is needed within a computer to control the jobs. CICS carries out this controlling function within a DB/DC job.

CICS provides the communications control and service functions necessary for users to create their own, customized DB/DC system. This cuts down the total amount of programming needed. You can customize CICS to the needs of practically any online application, and it can support networks consisting of a wide variety of terminals and subsystems.

For most of the time, the users will be unaware of CICS and, indeed, unaware of the existence of other applications. They will spend their time using the online application programs that you've designed for their particular transactions.

Because CICS **is** a general-purpose product, the view your users get of it will depend far more on the configuration of your system and the application programs you provide, than on any features of CICS.

1.1.3 What does CICS do?

CICS controls online DB/DC application programs. But what does this mean? In fact, it means that CICS is a program that does a lot of work on your behalf. CICS handles interactions between terminal users and your application programs. An interaction may consist of one or more requests from, and responses to, a terminal user in the course of a single job by that user.

CICS provides:

- The functions required by application programs for communication wit remote and local terminals and subsystems
- Control of concurrently running programs serving many online user
- Facilities for accessing databases and files, in conjunction with th various IBM database products and data access methods that are available
- The ability to communicate with other CICS systems and databas systems, both in the same computer and in connected computer systems.

We've left things as open as possible to allow our customers to produce the system **they** need. It's up to your systems and applications designers (which could mean you, of course) to choose what they want from the various CICS facilities, and to build whatever kind of user interface that best suits the end users. So, although you still have to provide the application programs that the end users actually run, CICS makes it much easier. Your programs gain access to the CICS facilities they need by straightforward, high-level, commands.

Subtopics
1.1.3.1 CICS application programs
1.1.3.2 Couldn't I do all this myself?
1.1.3.3 Can CICS serve large systems and small systems properly?

1.1.3.1 CICS application programs

Online application programs have certain features and needs in common. Typically, they:

Serve many online users, apparently simultaneousl

Require common access to the same data sets and database

Try to give each end user a timely response to each interactio

Involve telecommunications access to remote terminals

The host operating system is in overall charge of the computer and manages resources in whatever way you set up. But the very versatility of a general-purpose operating system means that it often cannot give online programs the sort of priority treatment they need. Instead, CICS may be given "privileged" treatment on behalf of all the online programs that run under it.

To make the best use of the time and system resources that the operating system gives to CICS, CICS takes on itself some of the aspects of an operating system. For example, CICS allows more than one of its programs (tasks) to be in an active state at the same time. But CICS doesn't duplicate all of the services provided by the operating system. Whenever appropriate, CICS goes straight to the operating system to provide what its tasks ask for.

1.1.3.2 Couldn't I do all this myself?

Yes, of course, but why reinvent the wheel? CICS is a large, mature piece of software that has evolved in parallel with the growth of online terminal networks and the movement toward distributed processing. It supports a wide range of hardware and software. Many thousands of data-processing installations around the world have based their data communication systems on CICS.

1.1.3.3 Can CICS serve large systems and small systems properly?

Yes. CICS is designed in a modular fashion, and we supply it as a set of programs that you can combine rather like building blocks. If you don't need certain CICS functions, you simply leave out those parts of CICS when installing your system. Or perhaps, more typically, you might install everything, but only **use** what you need.

To start with, though, you'll be putting together your first application on the subset CICS system that we've chosen for this Primer.

1.1.4 How does a CICS-based application differ from a batch application?

As we hinted in the preface, we expect you to have a batch programming background. That being so, you don't need us to tell you what batch programming is all about. However, we **do** want to tell you how a CICS-based application differs from a batch application.

Subtopics
1.1.4.1 Basic differences
1.1.4.2 Starting a transaction
1.1.4.3 Inside CICS

1.1.4.1 Basic differences

Not **everything** is different, of course. But here are some points to think about:

In a batch program, you often define all the required input/output an work areas within the program. In CICS, these areas are allocated by CICS, as needed, by CICS itself from a dynamic storage area within the CICS region. This lets CICS economize on main storage, and use the same copy of a program to do work for several users at once.

A batch program reads its own input data, whereas CICS reads the dat on behalf of the CICS application programs. A particular CICS application program need not even be loaded into the computer before its first input message arrives.

A batch program issues its input/output instructions directly to th operating system. CICS application programs always issue such instructions to CICS, and CICS handles the interface to the operating system.

Recovering when things go wrong is more interesting (as we'll see)

Subtopics
1.1.4.1.1 Recovering when things go wrong
1.1.4.1.2 Two vital terms

1.1.4.1.1 Recovering when things go wrong

The final major difference between a batch system and an online system comes up when things go wrong.

Obviously, all data processing systems need to be able to survive faults and errors such as the loss of power supply, processor failures, program errors, data set failures, and (in online systems) communication errors. Procedures are required to recover from such faults or to restart the system if a fault has stopped it.

Recovery and restart design is inevitably more complex for an online system than for a batch system:

For **batch** processing, input data is prepared before processing begins. The data is then supplied to the batch process in one orderly sequence, which is controlled and predictable.

For **online** processing, input data isn't prepared beforehand, but is entered as needed while the application is running. Furthermore, the input data can come from many different users working concurrently. In other words, input data does not arrive in a predictable sequence.

If a failure occurs:

With a batch program, you can repeat the processing, or continue i from the point of failure. This is because the processing sequence is **predictable** (it is based entirely on the predefined input data), and because the input data is still available.

With an online application, you cannot simply rerun the application o continue from the point of failure because the state of the process is unknown. And even if it were known, you couldn't expect the terminal users to reenter a day's work.

So, online application programs need a system that provides special mechanisms for recovery and restart. In broad terms, these mechanisms ensure that each resource associated with an interrupted online application is returned to a known state so that processing can be restarted safely. As you work through this book, you'll see how CICS can help you get over your recovery and restart problems.

Perhaps the most striking difference is how a small, simple application program can be loaded into the computer and promptly be used, by hundreds of people throughout a terminal network. Not only that, but the same application program could be in use by all these people at the same time. And yet these online application programs aren't necessarily more difficult to write and get working than the programs you've been used to up to now.

1.1.4.1.2 Two vital terms

Next, we want to introduce two important words in the CICS vocabulary: "transaction" and "task." You'll constantly see these so it's good to know what they mean right from the start.

A **transaction** is a piece of processing initiated by a single request, usually from an end user at a terminal. A single transaction will consist of one or more application programs that, when run, will carry out the processing needed.

In other words, "transaction" means in CICS what it does in everyday English: a single event or item of business between two parties. In batch processing, transactions of one type are grouped together and processed in a batch (all the updates to the personnel file in one job, a list of all the overdue accounts in another, and so on). In an online system, by contrast, transactions aren't sorted by type, but instead are done individually as they arrive (an update to a personnel record here, a customer order entered there, a billing inquiry next, and so on).

Having given you this straightforward definition, we'll immediately complicate things a bit by admitting that the word "transaction" is used to mean both a single event (as we just described) and a class of similar events. Thus, we speak of adding Mary Smith to the Payroll File with a (single) "add" transaction, but we also speak of the "add" transaction, meaning all additions to that particular file.

Things are further complicated by the fact that one sometimes describes what the user sees as a single transaction (the addition to a file, perhaps) as several transactions to CICS. We get to this nicety in "Pseudoconversational or not?" in topic 2.7. Until we get there, you should use the definition of transaction we've given above; you'll be able to tell from context whether we mean a transaction type or a single bit of processing.

Now, what about a **task?**

Users tell CICS what **type** of transaction they want to do next by using a transaction identifier. By convention, this is the first "word" in the input for a new transaction, and is from one to four characters long, although this source of the identifier is sometimes overridden by programming.

CICS looks up the transaction identifier to find out which program to invoke first to do the work requested. It creates a **task** to do the work, and transfers control to the indicated program. So a task is a single execution of some type of transaction, and means the same thing as "transaction" when that word is used in its single event sense.

A task can read from and write to the terminal that started it, read and write files, start other tasks, and do many other things. All these services are controlled by and requested through CICS commands in your application programs. CICS manages many tasks concurrently. Only one task can actually be executing at any one instant. However, when the task requests a service which involves a wait, such as file input/output, CICS uses the wait time of the first task to execute a second; so, to the users, it looks as if many tasks are being executed at the same time.

1.1.4.2 Starting a transaction

Normally, end users wishing to begin an online session will first identify themselves to CICS by signing on. Signing on to CICS gives users the authority to invoke certain transactions. Once signed-on, they invoke the particular application (transaction) they intend to use. They can do so by typing the **transaction identification code** at the start of their initial request. But, if your designers decide otherwise, it's just as easy to set up a particular program function (PF) key to invoke a transaction with a single keystroke or, indeed, for a given terminal always to invoke a particular transaction.

Application programs are stored in a library on a direct access storage device (DASD) attached to the processor. They can be loaded when the system is started, or simply loaded as required. If a program is in storage and isn't being used, CICS can release the space for other purposes. When the program is next needed, CICS loads a fresh copy of it from the library.

1.1.4.3 Inside CICS

In the time it takes to process one transaction, the system may receive messages from several terminals. For each message, CICS loads the application program (if it isn't already loaded), and starts a task to execute it. Thus multiple CICS tasks can be running concurrently.

CICS maintains a separate thread of control for each task. When, for example, one task is waiting to read a disk file, or to get a response from a terminal, CICS is able to give control to another task. Tasks are managed by the CICS **task control** program; the management of multiple tasks is called **multitasking**.

CICS manages both multitasking and requests from the tasks themselves for services (of the operating system or of CICS itself). This allows CICS processing to continue while a task is waiting for the operating system to complete a request on its behalf. Each transaction that is being managed by CICS is given control of the processor when that transaction has the highest priority of those that are ready to run.

While it runs, your application program requests various CICS facilities to handle message transmissions between it and the terminal, and to handle any necessary file accesses. When the application is complete, CICS returns the terminal to a standby state. Figure 3 should help you understand what goes on.

PICTURE 3

The flow of control during a transaction (code **ACCT**) is shown by the sequence of numbers 1 to 8 on the panels. Don't take this transaction too seriously; we're only using it to show some of the stages that can be involved. The meanings of these eight stages are as follows:

1. **Terminal control** accepts characters **ACCT**, typed at the terminal, and puts them in working storage.

2. **System services** interpret the transaction code **ACCT** as a call for an application program called **ACCT00.** If the terminal operator has authority to invoke this program, it is either found already in storage or loaded from....

3. **The program library** into working storage, where....

PICTURE 4

4. **A task is created**. Program **ACCT00** is given control on its behalf. This particular program invokes....

5. **Basic mapping support (BMS)** and terminal control to send a menu to the terminal, allowing the user to specify precisely what information is needed.

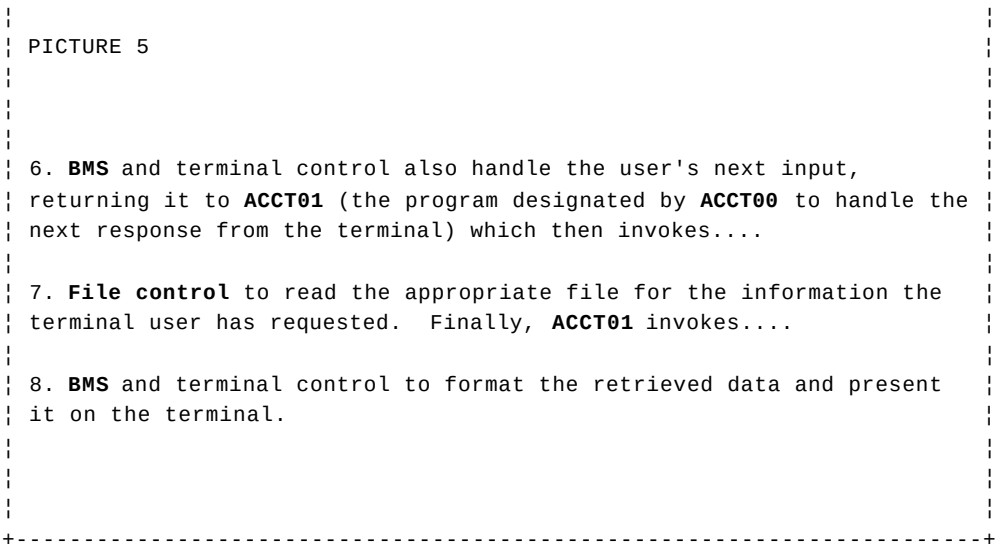


Figure 3. The flow of control during a transaction

The transaction continues to run until it reaches a place in the program at which it's waiting for some activity (such as a disk access) to end. At this point, CICS allocates the processor to the next task that can run. Only when there's no work to do on behalf of any CICS task does CICS pass control back to the operating system to allow batch work to run. This allows CICS to maintain the priority of online working over batch work in other address spaces.

In this way, CICS controls the overall flow of your online system.

Besides doing all the transaction processing, CICS also supports the bookkeeping side of the system, by accumulating performance statistics and monitoring the resources used. This gives you the information that enables user departments in an organization to be charged accordingly. It also allows you to find out which parts of CICS are being heavily or lightly used. This will help your systems people change the CICS set-up when you wish to tune your system to improve its performance.

1.1.5 How does CICS help you set up an online system?

After your system has been designed, the programming effort to turn the specification into a working application is normally divided between two groups: the people who install and maintain the system, and those who write the application programs it will use. (We don't want to rule out the possibility of all this work being done by one heroic person.) CICS offers a variety of helpful features for both groups. Concentrating on the application programming side, CICS aids include:

A **choice of programming language**. You can write your application programs in assembler, COBOL, PL/I, or C language.

A **command-level programming interface** with CICS. You need know little about how CICS works. You request data or communication with terminals by issuing CICS commands that resemble those of the programming language you are using. A **command language translator** preprocesses the application source code, translating CICS commands into CALL statements in the language of the application program. It also provides useful diagnostics.

An **execution diagnostic facility (EDF)**, for testing command-level application programs interactively.

1.1.6 How do you use CICS?

Now that you have some idea of what CICS is and how it fits into your computer system, we can explain how you use it. We're going to do so by showing you the stages in designing and implementing a reasonably typical (1) and useful application: a file inquiry and update system. This example starts in the next topic.

To get the best out of your CICS system (or, for that matter, any system) you should design the system around its applications. However, for our purposes, we'll assume that you've been through this process for other applications, and simply wish to extend your present system by adding this online file inquiry and update application.

In reality, if your proposed new application programs were very different from your existing ones, your systems programmers might have to tailor your CICS system to provide the necessary functions, typically by picking different sets of system parameters for different occasions. This could mean **initializing** the system again, to include IBM-supplied programs to help you do what you want. If your needs are very unusual, they might have to **customize** some parts of your CICS system, adding code of their own, before initializing the system. The programs that we develop and describe in this book are all supported by a simple CICS system, so you can forget about initialization or customization for the time being.

(1) Asking "What's a typical application?" is a bit like asking "How long is a piece of string?". Nevertheless, many diverse users share common information processing needs. So we shall see how CICS can meet the needs of a "typical application"--the online requirements of the *Flibinite boutique*--part of our example application, described in "Designing the user interface" in topic 2.3.

2.0 Application design

This part of the Primer:

- Explains how to design your first CICS application programs
- Defines the problem
- Describes 3270 Information System data streams
- Deals with designing the data
- Talks about establishing the user interface
- Examines special features of the CICS environment
- Defines the example application programs involved, and their interactions.

The CICS example application--a department store

The current situation

A department store (the *Flibinite boutique*) with credit customers keeps a master file of its customers' accounts. Each customer record holds the customer's name, address, telephone number, charge limit, current balance, account activity, payment history, and so on.

At the moment, a set of batch processing programs updates this master file (and some related ones) twice a week with the necessary charge and payment information. The records are also printed periodically, bound into bulky folders, and distributed to each section to help in answering questions both from customers and from within the Accounting and Customer Service sections.

However, the listing is too large to be printed often, so it's usually out-of-date.

Online access to information

The store wants online access to a customer's record, to have absolutely current information. So we need an inquiry function. Furthermore, the people in Accounts want to be able to update these customer records online, for convenience and currency. So we also need a facility to add new records, delete records and change addresses and other information unrelated to billing.

Each customer has a unique account number, which is the key to the existing master file. The users in Accounts will presumably access records by this number, because it's always available when they are processing work or answering questions.

Access by name

However, people in the Customer Service Department say they must be able to access the file by customer name if possible. Their experience suggests that customers don't usually know their account numbers, but can always remember their names!

If a customer wants to charge items but has forgotten to bring along the right charge card, a clerk calls Customer Service, verifies the existence and payment status of the account, and gets the account number for the charge slip.

Logging and printing changes

Finally, the people in Accounts have asked us to make quite sure that all changes to the file are logged, and all errors, with a hard-copy report in both cases. They seem to be rather nervous about subjecting their master file to online updating, but assure us that they will feel more confident having a printed record of all changes made.

They are also concerned about the security aspects of this first venture into online file updating, and want to be able to trace changes to specific records. Later, they will probably agree to direct this log to tape, printing it only when necessary, but for the moment they need it in hard-copy form.

Subtopics

2.1 The CICS example application--a department store

This topic explains, with the help of an example, one way of designing a CICS application. The text you've just been reading (in the boxes opposite) describes what the application is to do.

The outline specification for our example is a simple one. It shows design issues and programming requirements that arise in nearly every application. The CICS services required by this application are a subset of the full range available; however, this subset consists of those functions that most straightforward applications need to use. Let's relate the department store's needs to some general points about CICS application programs. A CICS application usually consists of three main parts consisting of the data to be processed, the transactions to be performed, and the interface with the user.

You can see these parts in the specifications just described for the example. The customer information in the account file is the data to be processed; the online operations (display a record, add a record, and so on) are the transactions to be performed on that data; and the terminals, formatted screens, and operating procedures are the interface with the user. Let's see how each of these parts could be designed.

It is important to note before starting, and it will certainly be clear in what follows, that each of these three parts bears on the others. You cannot design one without reference to the other two.

Moreover, design is an iterative process. Decisions about the user interface affect transaction definition, which, in turn, causes a slight change in specifications, and the whole cycle begins again. These adjustments are normal and should be expected in any design process. However, unless you freeze the design at some point you will never complete the job.

Subtopics

2.1.1 Defining the problem

The first step in the design process is to specify broadly what the application will do. In our case, the need for the application came from two user departments, and the first functions they requested are:

Display of customer account record, given an account numbe

Addition of new account record

Modification of existing account records (by account number

Deletion of account records (by account number

Hard-copy listing of changes to the account fil

Ability to access records by name

Subtopics
2.1.1.1 The account file records
2.1.1.2 Requirements imposed by the environment
2.1.1.3 Refining and developing the program specifications
2.1.1.4 Estimating the number of transactions

2.1.1.1 The account file records

The detailed design of our programs is going to be influenced by the established form of the existing customer data, of course. The account file is very much at the center of this application. Its records are shown in Figure 4.

Field	**Length**	**Occurs**	**Total**	**Type**
Account Number (Key)	5	1	5	Online
Surname	18	1	18	Online
First Name	12	1	12	Online
Middle initial	1	1	1	Online
Title (Jr, Sr, and so on)	4	1	4	Online
Telephone number	10	1	10	Online
Address line	24	3	72	Online
Other charge name	32	4	128	Online
Cards issued	1	1	1	Online
Date issued	6	1	6	Online
Reason issued	1	1	1	Online
Card code	1	1	1	Online
Approver (initials)	3	1	3	Online
Special codes	1	3	3	Online
Account status	2	1	2	Batch
Charge limit	8	1	8	Batch
Payment history:	(36)	3	108	Batch
-Balance	8			
-Bill date	6			
-Bill amount	8			
-Date paid	6			
-Amount paid	8			

Figure 4. Account file record format

The fields marked as Type "Online" are the ones that are to be maintained by our online program. Those marked "Batch" are already updated by the existing batch billing and payment cycle and need only be displayed by our online system.

2.1.1.2 Requirements imposed by the environment

Besides the users' requirements, we're going to assume that certain others are imposed by the environment in which this application will run. These are:

The terminals available are IBM 3270 system displays and printers The screens display 24 lines, each of 80 characters (the IBM 3278 Display Station model 2, for example), with corresponding printers.

Some of the people who will use the application will do s infrequently. Consequently, the application should be as self-documenting as possible, and users should not need to memorize very much to use it comfortably. On the other hand, help to casual users should not result in slow or annoying interactions for frequent users. Some hard-copy documentation on how to use the system will be provided, but we hope users will only rarely need to look at it. The goal is to keep everything nice and simple for all users.

The integrity of the account file must be maintained. This means tha it must be protected from inconsistent or lost data, whether resulting from a failure in the application or CICS or the operating system. It also must be protected from total loss, such as a disk head crash or other catastrophe.

The existing account file is a VSAM key-sequenced data set containin about 10 000 records of 383 characters each, including the 5-digit account number key.

2.1.1.3 Refining and developing the program specifications

The next step in defining the problem is to verify the first program specifications with whoever made the original requests. You should be especially alert for information or functions that no-one requested but that nevertheless may actually be required when real work is attempted. Otherwise the users will make the same discoveries right after you complete your programming effort, and you'll be faced with making changes when it may prove difficult, rather than now when it is easy.

It is always useful to talk to the actual users of an application, to find out how they do their work and how they view the functions you intend to provide. Supervisors can provide other insights. It is very important to repeat this verification step as the design process moves along from a broad outline toward more and more detailed specifications.

2.1.1.4 Estimating the number of transactions

Now is also the time to find out how often the system will be expected to cope with the transactions of each type, what sort of response times will be expected, what times of the day the application will have to be available, and so on. This will allow you to design programs that are efficient for the bulk of the work, and it will help you in determining system and operational requirements.

For the example application, let's assume that our inquiries produced the following information:

> There will be about 10 additions, 50 modifications, 5 deletions, an 200 inquiries (by account number) per day in the Accounting Department.
>
> The people in Accounting are unable to estimate the number o inquiries that they would make by name, but they sound intrigued with the possibility, and therefore may be expected to make some use of this facility.
>
> Accounting would find it very useful to be able to get a printed cop of a customer account record, besides being able to display it on the screen. (This is a new requirement, not in the original specification. We should consider providing it.)
>
> Customer Service makes nearly 1000 inquiries per day against accoun records, ninety percent of them by name. For most of these, the only items used from the complete account record are the name and address (to verify that it is the *right* record), and the credit status and limit.

Note: In assessing estimates of transaction frequency, we need to account for a fact of life. That is, if we make it much easier to do something, such as an inquiry, users will almost certainly do it more often than they used to do. Indeed, the eventual transaction rates experienced with online systems are almost always higher than can be predicted from the current workload -- often a reliable indication of their success.

2.1.2 Summary

We've now identified some of the first steps when starting to design an application. You should:

Broadly set down the application functions based on user need

Identify the individual data elements involved in the processin

Consider any external environmental factors and restriction

Verify your initial specifications with the user

Estimate the expected load on the system from the various ne functions that your application will provide.

When you've done this, you can then go on to design the transactions and processing programs that you'll need. So, let's continue now with some application design considerations.

2.1.3 Designing the transactions: preliminaries

Earlier in this topic, we described the functions needed in our example. Let's now see how we might define transactions to perform these functions. One obvious approach is to make each function a separate transaction. The transaction to display an account record, then, would work something like this:

> Find out from the terminal user which record is to be displayed
>
> Read that record from the file
>
> Display the information from that record at the terminal

That seems straightforward. How about the add transaction?

> Get the data for the new record as keyed in by the user at th
> terminal.
>
> Write this data to the file

Even simpler. However, there are a few things we've not taken into account.

First of all, we're not dealing with the familiar batch devices of card reader and line printer here. The 3270-system terminals are radically different in their characteristics from such batch devices. They are different, too, from line-oriented or record-oriented devices such as Teletypes (**) and IBM 2741s.

Second, there are human beings operating the terminals, and their happiness and efficiency must be a major design goal in *any* application.

Third, we have to deal with the implications of processing in an online environment, where our goals and constraints may be quite different from those that govern a batch program.

Finally, we've not provided for any exceptional conditions. For example, what if the record to be displayed isn't in the file? Or if the one to be added *is* in the file? You probably know that in a batch program about 80 percent of the effort and the code is devoted to handling errors, even though this code is executed rarely. In online programs, all these same problems have to be thought about and resolved, and there are also some new potential problems.

(**) Trademark. For a complete list of trademarks, see "Notices" in topic FRONT_1.

2.1.4 What next?

Before we continue trying to design our transactions, let's learn a little more about the 3270 systems that our users will be using to communicate with the transactions. After all, one of the first things to be considered is the user interface: how will the terminal operators communicate with this application, and how will it give them the information they need?

We can then go on to find out more about a much wider range of issues: what makes users happy ("human factors"), the design of data, programming for a CICS environment, and so on.

But first, 3270s. If you are already familiar with 3270 terminals and the 3270 data stream, you can skip ahead to "Designing the user interface" in topic 2.3.

2.2 3270 terminals

Remember, you're free to skip this topic if you know about IBM 3270 terminals already.

The 3270 Information Display System is a family of display and printer terminals. Different 3270 device types and models differ in screen sizes, printer speeds, features (like color and special symbol sets) and manner of attachment to the processor, but they all use essentially the same data format.

You need to know a little about this format to make the best use of 3270-system devices, and to understand the **Basic Mapping Support** (BMS) services that CICS provides for communicating with these devices. That's the purpose of this topic.

Let's talk about the IBM 3278 Display Station Model 2, which has a display screen and a keyboard. This device is used for both input and output, and in both cases the screen (or rather a buffer that represents it) is the crucial medium of exchange between the terminal and the processor. The purpose of the keyboard is to modify the screen, in preparation for input, and to signal when that input is ready to be sent to the processor.

When your application program writes to a 3278, the processor sends a stream of data in the special format used by 3270 devices. Most of the data in the stream is the text that is to be displayed on the screen; the rest of it is control information that defines where the text should go on the screen, whether it can be overtyped from the keyboard later, and so on.

The printers that correspond to the 3278 can use this same data stream, so a stream built for a display device can be used equally well for a printer.

Subtopics

2.2.1 3270 field structure

The screen of the 3278 Model 2 can display up to 1920 characters, in 24 rows and 80 columns. That is, the face of the screen is logically divided into an array of positions, 24 deep and 80 wide, each capable of displaying one character, with enough space around it to separate it from the next character.

Each of these 1920 character positions is individually addressable. This means that your COBOL application program can send data to any position on the screen, without having to space it out with space characters to get it into the right location. Your program does not, however, give an address for each character you want displayed. Instead, within your program, you divide your display output into **fields**. A field on the 3278 screen is a consecutive set of character positions, all having the same display characteristics (high intensity, normal intensity, protected, not protected, and so on). Normally, you use a 3270 field in exactly the same way as a field in a file record or an output report: to contain one item of data.

To show you how this works, Figure 5 shows the screen that the CICS system uses for the standard sign-on transaction:

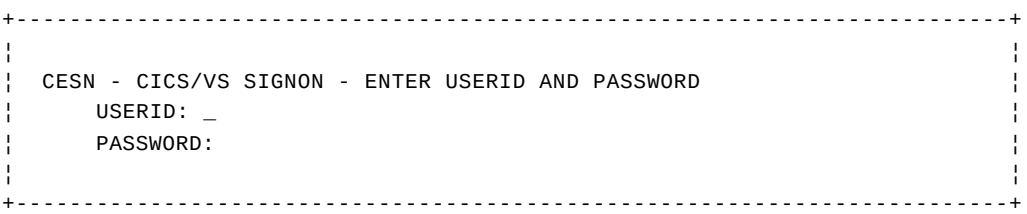


Figure 5. The CICS sign-on screen

There are a number of fields on this screen although, as shown, only three of the fields are displaying character data. The first one is at row 1, column 1 (position 1,1), and it contains the data **CICS/VS SIGNON - ENTER USERID AND PASSWORD**. The field is specified as both **protected** (meaning that the terminal operator cannot type over that area of the screen) and **bright** (high intensity, in this case just for emphasis). The second field is at position (4,5) and contains the data **USERID:**. This is also protected and bright. (The underscore after **USERID:** is the cursor and marks the position into which the next character entered from the keyboard will go.) Both of these fields have been used for output only, to convey something to the user. For the second field, it was to show what should be typed into the third field. The second field is followed by an attribute byte at position (4,11), and then the third field starts at position (4,12).

This third field is different because we intend the user to key something into it which will become input the next time the terminal transmits. So it isn't protected. It is set for normal intensity, and, even though you cannot see this by looking at the screen, it is 20 positions long. This is the permitted length of the name field in the CICS **Sign-On Table**, with which the contents of this field will later be compared.

At the end of this field is another field, known as a **stopper field**. (You can't see this one, either.) Its only function is to stop the user from keying more than 20 characters into the name field. The reason for this is that the beginning, but not the end, of each field is flagged in the buffer that represents the screen. The end of a field is one position before the start of the next field. There's no data in this "stopper" field; the important thing is that it is protected. Whenever you try to key into a protected field on the screen, you are prevented from doing so, and the keyboard locks. Users who try to key more than 20 characters into

the name field, therefore, run into this protected field, and are made aware of the error by the locking of the keyboard.

The next three fields are two lines down, at positions (6,5), (6,15) and (6,24). They are rather like the three fields on the earlier line. The first of them contains the data **PASSWORD:** and is protected. The second is the field into which the user is supposed to enter the password. It is unprotected, and has another attribute that may at first seem curious. It is **dark** or **nondisplay**. This means that the data in the field does not show on the screen (whether the user puts it there or the program does), even though it is very much there. Nondisplay is used for this field because passwords are supposed to be secret, and this way no one passing by while the user is signing-on will see the password. The third field is again a stopper field to stop the user from keying in more than eight characters of password information.

2.2.2 3270 output data stream

Now let's see how this information is formatted for transmission from the processor to the 3278. Figure 6 shows the data stream.

Control information affecting the whole transmission, such as whether to unlock the keyboard or not, where to place the cursor, and so on.	
First field:	**Encoded screen address** showing where the next field goes on the screen (row 1, column 1)
	Control information to show that a field is about to begin
	Control information to describe display attributes of field: high intensity, protected
	Data to be displayed: "CESN - CICS/VS SIGNON - ENTER PERSONAL DETAILS"
Second field:	**Encoded screen address** showing where the next field goes on the screen (row 4, column 5)
	Control information to show that a field is about to begin
	Control information to describe display attributes of field: high intensity, protected
	Data to be displayed "USERID:"
Third field:	**Control information** to show that a field is about to begin
	Control information to describe display attributes of field: normal intensity, unprotected
	Control information cursor position
Fourth field:	**Encoded screen address** showing where the next field goes on the screen (row 4, column 32)
	Control information to show that a field is about to begin
	Control information to describe display attributes of field: protected (stopper)
Fifth field:	**Encoded screen address** showing where the next field goes on the screen (row 6, column 5)
	Control information to show that a field is about to begin
	Control information to describe display attributes of field: high intensity, protected

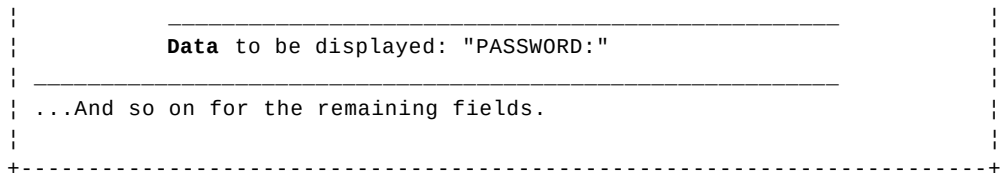


Figure 6. A 3270 output data stream

There are several things to note about this data stream:

For the first and second fields, a screen address appears in the dat stream, whereas for the next field it does not. This is because no new address needs to be provided when one field immediately follows another. Addresses for these fields could be included, but they would increase the length of the transmission. It is important to keep transmissions as short as possible when dealing with terminals that may be connected by telephone lines.

Similarly, data is included for the first two fields but not for th next two. Again, if there is no data, it isn't necessary to include anything in the data stream. This also reduces the length of the transmission.

We've shown the various fields for the screen being transmitted in th order they appear on the screen. This is customary and natural, but it isn't required by the device, which will accept fields in any order, but it's much faster to display them in the correct order.

The most striking feature of the data stream is its variable lengt and format, which depend on the presence or absence of data, adjacency or nonadjacency of fields, and so on. This would be very cumbersome to produce in a COBOL program, to say the least. Moreover, every time you moved something about on the screen, you would have to change the program that produced the data stream.

Don't panic! "Saved by BMS" in topic 2.2.6 shows us the light at the end of this particular tunnel.

2.2.3 3270 attribute bytes

One more point about this output data stream. If you followed the screen positions used in the example carefully, you may have noticed that each field seems to be one position too long. If the 20-position name field begins at (4,11), why doesn't the stopper field start at (4,31) instead of (4,32)? This is because the display attributes to which we've referred (protected, bright, and so on) actually occupy one screen position for each field. That is, if we start a 20-character field at position (4,11), the **attribute byte** (as it is called) for the field is located at (4,11) and the actual data goes from (4,12) through (4,31). The attribute byte looks like a space on the screen, and is itself protected (whether or not the field to which it applies is protected), so that the user cannot key into it and change the field identity.

As noted earlier, the attribute byte controls how data is shown on the screen. The choices are:

    High intensit
    Normal intensit
    Dark (nondisplay)

The attribute byte also governs what can be done to the field from the keyboard. Here the choices are:

    Unprotected: The user may key anything into the field

    Numeric: The user may key only digits, decimal points, and minus sign into the field.

    Protected: The user may not key into the field

    Autoskip: The user may not key into the field and, furthermore, th cursor will automatically skip over the field if the previous field is filled.

Autoskip is usually used for stopper fields if the information in the previous field is of fixed length and always fills the field. That way, the user can key continuously, and doesn't have to use the cursor advance key after filling a field to get to the next one.

After variable-length data (such as the name field in the sign-on screen) however, it is customary to make the stopper a protected field, instead. If you specify autoskip, and the user keys too much, the excess goes into the next unprotected field, and the user may not be aware of this. Where there are fields for both fixed-length and variable-length data, some programmers like to use only protected stoppers, so that the user consistently has to use the cursor advance key to get to the next field, whether or not the current field is full. Others prefer to use both kinds on the same screen.

The attribute byte also carries one more piece of information. This is the **modified data tag**. It has to do with input, however, and so we'll explain it later. (If you can't wait, you'll find more details on 2.2.4 and in "The BMS macros" in topic 3.2.2.)

Note: Not all combinations of attributes are permitted, but all the useful ones are. We should also point out now that displays with additional features, like color and special symbols, have more complex attribute combinations to express the additional possibilities. However, the logic for formatting the data stream with these extended attributes is essentially the same.

2.2.4 3270 input data stream

Now that we've described what output to a 3278 looks like, what does the input look like? There are several different possible formats, and the one used depends both on the type of read command used and on certain other circumstances. Figure 7 shows our sign-on screen after John Jones has been busy at it.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦  CESN - CICS/VS SIGNON - ENTER USERID AND PASSWORD                     ¦
¦      USERID: JONESJO                                                   ¦
¦      PASSWORD: OPNSESME_                                               ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 7. The sign-on screen in use

We're showing you the password here but, remember, you wouldn't normally see it because it's held in a nondisplay field.

What is of interest to us is what CICS gets when it reads a screen like this one. Figure 8 shows us what comes back after the user presses the **ENTER** key.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ _____________________________________________________________          ¦
¦ Control information affecting the whole transmission, such             ¦
¦ as which key caused the input to be sent (ENTER, PFx),                 ¦
¦ where the cursor is, and so on                                         ¦
¦ _____________________________________________________________          ¦
¦ First     Encoded screen address showing where                         ¦
¦ field:    the field was on the screen (here Row 4,                     ¦
¦           Column 11).                                                  ¦
¦           _____________________________________________________        ¦
¦           Contents of the field: "JONESJO"                             ¦
¦           (7 characters, not the full 20 allowed).                     ¦
¦ _____________________________________________________________          ¦
¦ Second    Encoded screen address showing where                         ¦
¦ field:    the field was on the screen (here Row 6,                     ¦
¦           Column 15).                                                  ¦
¦           _____________________________________________________        ¦
¦           Contents of the field: "OPNSESME"                            ¦
¦ _____________________________________________________________          ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 8. A 3270 input data stream

Points to note about this transmission are:

Practically nothing came back. All the fields used for titles an labels have been omitted from the transmission, and even the "new password" field, which the user did not fill in, is missing. This is because only *changed* fields are transmitted back on the kind of read used here by CICS. The reason the hardware works this way is, again, to minimize the length of the transmission.

How does the 3278 know what to send? When a user keys into a field, a bit in the attribute byte is turned on. This is the modified data tag, or "MDT." You can also turn this bit on when you write to the screen, so that the field is returned whether or not the user keys into it. This provides a handy method for storing information on the

screen between transactions, but we'll explain that later, in "Communication between transactions" in topic 2.9.1.

The second thing to note is that only the significant portion of changed field is sent; the unused portion on the right-hand side of the field is not. This is because the 3270 does not send empty positions on the screen. Empty positions are called **nulls**, and have a character encoding of hexadecimal (hex) 00 (**LOW-VALUE** in COBOL). If you ask for the screen to be erased (as you'll often want to) before your data stream is written to it, the screen is set to nulls. Nulls aren't the same as spaces, even though they look the same on the screen. Spaces have a hexadecimal representation of 40 and are transmitted; thus the space between **JOHN** and **JONES** comes in, but the unused part of the field after **JONES** does not. This is, once again, to minimize the length of the data transmission.

The result of all these length-reduction measures is another data stream of extremely variable format. This time the position of the data coming back depends not only on the content of what was sent but also on what the operator did, presenting a considerable challenge to decode.

We mentioned earlier that there were several different formats used for transmission to the processor, depending on the type of read used and other circumstances.

One of the other circumstances is the type of key the operator used to send the input. A number of keys cause the 3278 to send input to the processor at the earliest opportunity (these keys include **CLEAR** and **ENTER,** the program access (PA) keys, and the program function (PF) keys). Of these, the **CLEAR** key and the PA keys send only the identity of the key itself, without sending any of the data on the screen. If the operator uses one of these so-called "short-read" keys, the data stream shown in Figure 8 ends right after the initial control information. This causes a special situation which you'll have to deal with in any program that tries to read a formatted screen.

2.2.5 Unformatted 3270 data

As well as transmitting a short data stream to the processor, the **CLEAR** key also erases the screen. The entire screen is set to null values, and there are no fields. You may prefer to think of the screen as just one big field, but it is a field without attributes. The user can key into this field and send it to the processor. In fact, if you think about it, almost every new transaction is going to start this way. The user presses **CLEAR** to erase the leftovers from the previous operation, and then keys in something to identify the next request and transmits it with the **ENTER** key. What does this look like coming in to the processor?

Data that comes in from a screen that was not formatted into fields by a previous write is called, very logically, **unformatted** data. The data stream looks like the one in Figure 8 in topic 2.2.4 except that no address is provided (the data is assumed to start at the first position on the screen), and there is only one field. The field consists of every character that isn't a null--that is, every character that the user keyed--regardless of where it is on the screen, and in the order it appears on the screen).

Unformatted data is handled in CICS with a slightly different set of commands from formatted data. Unformatted data is actually simpler than formatted data (and you can write it as well as receive it), but it isn't nearly as useful. So we'll only cover formatted data in this Primer, and point you to where you can find out how to use unformatted screens if you should want to.

2.2.6 Saved by BMS

We said earlier that you do not have to deal directly with this data format in your CICS program. The feature of CICS that spares you this complexity is called Basic Mapping Support (BMS). BMS:

Allows you to deal with data in a fixed format, providing a dat structure for you to copy into your program in which the input fields (the name, password, and new password in the example we showed) are always in the same place and of the same (maximum) length.

Allows you to deal with data by name. In this instance we might hav called the fields where we expected input **NAME,** and **PSWD.** (We would do this when we first defined the screen.) Then we could refer to these variables by name in our program, without any concern for where they are on the screen.

Allows you to define all the constant data for the screen (titles field labels, and so on) separately from your program, so that you don't have to clutter your code with a great many statements like

```
MOVE 'ENTER PERSONNEL NUMBER' TO ....
```

Saves you from having to know about the details of the 3270 dat stream.

With these facilities, you can change the arrangement of the screen, the words in the titles, and so on without any changes to your program--a very important advantage.

"What BMS does" in topic 3.2.1 tells you more about BMS and explains how to use it.

Now, let's go on and look at what we'll have to consider when designing the user interface.

2.3 Designing the user interface

We know broadly what we want our application to do:

- Display customer account records, given their account number
- Add new account record
- Modify existing account record
- Delete account record
- Print a list of the changes made to the account fil
- Print a single copy of a customer account recor
- Access records by name

We also now know something about how the 3270 data stream works and how CICS starts transactions. So we can start thinking about how our application might look to the user.

Subtopics

2.3.1 A first approach
2.3.2 A user-friendly approach
2.3.3 Some interface design principles

2.3.1 A first approach

One approach is to review the transactions which the user wants to do, and think about what the user should see while performing each one.

Subtopics
2.3.1.1 The display transaction
2.3.1.2 The print transaction
2.3.1.3 The add transaction
2.3.1.4 The modify transaction
2.3.1.5 The delete transaction

2.3.1.1 The display transaction

If we take the simplest one as a starting point, displaying a record in the file, then we need to decide:

1. How the user enters a request.

2. How we show the user the requested record.

3. What to do if the user makes a mistake.

The user need enter only a very little information to request the display of a record: just the transaction type (display, in this case) and something to identify the record to be displayed. The output, on the other hand, is quite extensive, consisting of all the fields in the account record.

We can therefore imagine that a user wanting to display a record might switch on the terminal, sign-on to the system, clear the screen, and enter something like:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ DISP12345                                                              ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

DISP here is the transaction identifier that CICS needs to decide which transaction the user wants to perform, and **12345** is the number of the account to be displayed.

If the requested record can be found in the Account File, the application program should respond with a screen showing the data in the record.

To make the screen as easy as possible to understand, we should label each field to show what it means. Figure 9 shows a possible screen format.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦  ACCOUNT FILE: RECORD DISPLAY                                         ¦
¦  ACCOUNT NO: 12345       SURNAME:   MOUNCE                            ¦
¦                          FIRST:     DAVID           MI: C  TITLE:     ¦
¦  TELEPHONE: 7512483960   ADDRESS:   79 WISTFUL VISTA                   ¦
¦                                     PLEASANTVILLE, NY 10549           ¦
¦  OTHERS WHO MAY CHARGE:                                               ¦
¦  CHRISTA MOUNCE (WIFE)              PETER MOUNCE (SON)                ¦
¦  NO. CARDS ISSUED: 2     DATE ISSUED: 04 01 89    REASON: L           ¦
¦  CARD CODE: C            APPROVED BY: CES         SPECIAL CODES: A J  ¦
¦  ACCOUNT STATUS: N       CHARGE LIMIT:  2000.00                       ¦
¦  HISTORY:   BALANCE      BILLED        AMOUNT     PAID        AMOUNT   ¦
¦               0.00       04/25/89      101.37     05/05/89    101.37   ¦
¦               0.00       05/25/89       42.50     06/08/89     42.50   ¦
¦            3210.97       06/25/89      321.97                         ¦
¦  PRESS "CLEAR" OR "ENTER" WHEN FINISHED                               ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 9. An example of a display screen format

If the request wasn't correct, we have to write back some sort of message explaining exactly what's wrong. Very little can go wrong here with the display transaction (unlike the add transaction, where all sorts of things can happen!). The user can make a format error in specifying the record,

or name a non-existent record and thus try to display something that isn't there.

Note that CICS has to deal with errors in the transaction type. If the user gets the **DISP** part wrong, CICS won't know what transaction to start up, and will so inform the user. So, if the user enters something other than **DISP**, but something that happens to match a valid transaction identifier, CICS will happily start up the "wrong" transaction. Beware! (The "cure" the user generally tries in such a situation is usually to press the **CLEAR** key and try again.)

Other, "higher level" error possibilities include:

The user may not be authorized for acces

The account file may not be onlin

There may be a physical error while accessing a record from the file

However, in the absence of these "high level" problems, as we said, very little can go wrong here.

2.3.1.2 The print transaction

We can make the print transaction very similar to the display transaction. The only functional difference will be that the output will go to a printer instead of the screen. If we intend to use more than one printer, we'll probably want to let the user tell us which one, which means another item of input (and, we must admit, more opportunity for error).

2.3.1.3 The add transaction

When it comes to adding a new record to the file--an add transaction--we must still think about the same three things as for the display transaction. Unlike the display situation, however, the input required is very extensive. We could let users enter the request and the particulars for an add at the same time, but this would make things rather difficult for them, besides being a poor use of the 3270. With that many fields to enter, we definitely want users to enter the input into formatted screens, with labels to show where and how to enter the data.

So users will have to make two entries to do an add. The first one will display the formatted screen, and the second will contain the input for the addition. The output screen for the first stage of the add will be the skeleton into which the user is to enter the data. No output is actually *required* from the second stage of the add, but good human factors suggest that we consider telling the user that the transaction was successful.

Also, unlike the display transaction, there are plenty of opportunities for errors on an add. The record to be added might already exist on the file, or some of the fields entered might be missing or incorrect or inconsistent with each other. We don't want to make our users start all over again if they get one or two items wrong, so we'll have to think of a way for them to fix any bad fields without rekeying the good ones.

Maybe an add transaction could go like this. The user would enter something like **ADD 12345** and the transaction would do one of two things. Either it would respond with an error message that the record to be added already existed (far better to tell the user now, instead of after all the data for the record has been keyed in). Or it would display a skeleton screen for the user to fill in.

Now, users entering records are probably reading from a form of some sort while they do the data entry. It's very helpful to them if you make the screen look as much like their original data form as possible. Figure 10 shows the sort of skeleton screen that we'd want. (The underscores simply show where the input fields are; they wouldn't appear on the screen.)

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦  ACCOUNT FILE: NEW RECORD                                               ¦
¦  ACCOUNT NO: ______      SURNAME: ____________________                   ¦
¦                          FIRST:   _____________   MI: _  TITLE:_____    ¦
¦  TELEPHONE: ___________  ADDRESS: ___________________________           ¦
¦                                   ___________________________           ¦
¦  OTHERS WHO MAY CHARGE:                                                 ¦
¦  ___________________________________   ___________________________________  ¦
¦  ___________________________________   ___________________________________  ¦
¦  NO. CARDS ISSUED: _     DATE ISSUED: __/__/__    REASON: _             ¦
¦  CARD CODE: _            APPROVED BY: ___         SPECIAL CODES: _ _ _  ¦
¦  ____________________ (message area) ____________________________________  ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 10. A corresponding skeleton screen

Notice there are some bits and pieces on the form that we haven't transferred to the data entry screen. For example, the addresses of the other account users, the meanings of the four "reason" codes, the format of the date, and the customer's signature.

While it's generally true that a well-designed form will translate

painlessly into a data entry screen, never miss the chance to re-think aspects of the data entry task from the terminal operator's point of view. Also remember that if the operator's receiving information during a telephone conversation, the original form may be largely irrelevant to that particular situation.

After the user had filled in this screen, the transaction would check the input fields for reasonable and consistent values. If one or more of them were unacceptable, it could redisplay the user's input with the fields in error highlighted, and with a message added that the highlighted fields were either wrong or inconsistent with each other. The user could then fix the errors, and this input-edit-redisplay cycle could be repeated until the input was right. Then the transaction would send a message to the terminal saying that the record had been added to the file.

Strictly speaking, the transaction needn't *confirm* that the addition was successful. However, many users don't entirely trust computers, and a wary user might develop the habit of doing a display transaction after each add, just to make sure the add worked. This would waste a lot of user and computer time, and can easily be avoided by having a confirmation message.

2.3.1.4 The modify transaction

A modification could be almost like an add, except that instead of a skeleton screen being displayed, the information in the record would be displayed instead. The user would show the changes by typing over the old information on the screen.

2.3.1.5 The delete transaction

The deletion could be a very simple matter. We could let the user enter **DELE12345**, and then simply delete account number **12345,** and send back a message that we had done so. It turns out that this isn't a good idea, however. Users could easily make a mistake in keying the account number, and would be very distressed when they realized that they had removed the wrong record and had to put it back again. Worse than that, they might not notice at all!

Generally, when you're about to perform something as potentially irrevocable as a deletion in an online system, it's a good idea to confirm that the user really wants to go ahead with it.

Therefore, we probably want a deletion to be handled like a special case of a modification. Users will enter the account number to be deleted; we'll show them the record they are about to delete; and instead of keying in changes as they would for a modification, they will enter something to confirm that the record on the screen is really the one they want to delete. Only then will we delete it and say that we've done so.

Of course, we must give the user some way to say "no, I didn't mean it," cancel the transaction, and escape the deletion. Come to think of it, we'll have to do that in all these update transactions. If a user starts to add a record and then can't complete the entry for some reason (perhaps some required information is missing), then the user must be able to cancel the request without corrupting the files with a half-completed addition, modification, or whatever.

2.3.2 A user-friendly approach

Subtopics
2.3.2.1 Using a menu screen
2.3.2.2 Printing the logs
2.3.2.3 Name inquiry

2.3.2.1 Using a menu screen

Before going on to the other transactions, let's look at an alternative approach to this growing list of transaction identifiers. It's called the **menu** technique, and it's very popular as a user interface.

It works like this. For any application, users need to remember just one transaction identifier. When they want to do any transaction in that application (in our case, add, display, print, and so on) they enter just the one transaction identifier. In response, the screen displays a menu of things that the users can do in this application. The menu has formatted fields for the data items that are required on input. It also shows instructions in case users don't remember exactly what to do.

The chief advantage of this technique is that the user has to remember almost nothing, a big help to the "infrequent" users of our example application.

There are some other benefits as well: you can diagnose errors in the request input in the same convenient way that we described for the "add" screen, so that the user gets a good explanation of the problem and has to do a minimum of rekeying to correct the errors. Also, when you complete a transaction such as an add, you can combine your confirmation message with this menu screen. This way the user knows that the previous entry was successful, and is all ready to enter the next request.

The menu for this application might look like the one here (Figure 11). Again, the input fields are underscored in the figure to show their position, but the underscores wouldn't appear on the actual screen:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦  ACCOUNT FILE: MENU                                                     ¦
¦     TO SEARCH BY NAME, ENTER:                        ONLY SURNAME       ¦
¦                                                      REQUIRED. EITHER   ¦
¦        SURNAME: __________  FIRST NAME: __________   MAY BE PARTIAL.    ¦
¦                                 ____________________                    ¦
¦     FOR INDIVIDUAL RECORDS, ENTER:                                      ¦
¦  ___________________________________   ___________________________________ ¦
¦  ___________________________________   ___________________________________ ¦
¦  NO. CARDS ISSUED: _     DATE ISSUED: __/__/__     REASON: _            ¦
¦  CARD CODE: _            APPROVED BY: ___          SPECIAL CODES: _ _ _  ¦
¦  ____________________ (message area) ____________________________________ ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 11. An example of a menu screen

Almost the only disadvantage to this menu technique is that a user has to go through one extra screen for the *first* transaction of a session, and one extra step (clearing the screen in this case) to escape. The only time this is a serious matter is when users need to mix transactions from different *applications* constantly. This isn't the case in our example, and we do have infrequent users to think about, so we'll use the menu approach.

So here's how, say, a modify transaction will work:

1. The user keys in the four-character transaction identifier to get started.

2. The menu screen is displayed in response.

3. The user enters **M** for the request type, keys in an account number, and presses **ENTER.**

If there's a problem, the user will see the same screen with the fields in error highlighted and a message at the bottom saying what's wrong.

Otherwise, the response will be a display of the record to be modified, ready for the user to change. The user will change the fields to be modified, and then press **ENTER** to send the screen back. If there are errors in the changes, the transaction will send back the input with the errors highlighted and a message if necessary. If (when) the user gets it right, the transaction will update the file, and send back the menu screen, with a message at the bottom saying that the modification just requested was completed successfully. The user will then enter the next request, or clear the screen to quit our application.

2.3.2.2 Printing the logs

We've not yet dealt with the printing of the two logs: the log of changes to the account file, and the log of errors. The logs will be printed only occasionally, perhaps once a day, and this will be done by a supervisor in the Accounting Department. We probably don't want to include these options in our menu, because it will only confuse the other users, who may not even know what a log is. So we'll have separate transaction identifiers for these two functions.

The main output in either case, of course, will be a printed log. We should also send a confirmation to the input terminal, however, in case the printer isn't in the immediate area or is busy with another task at the time of the request.

2.3.2.3 Name inquiry

Finally, we must think a little more about the name inquiry transaction.

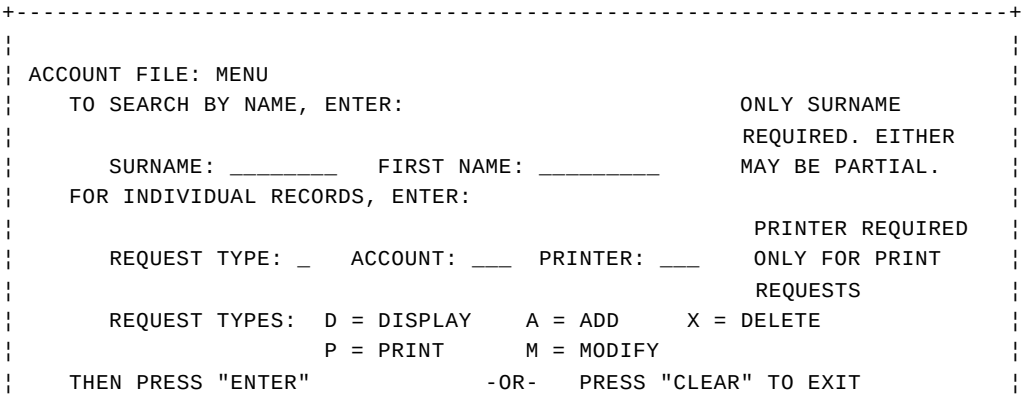
In view of the structure of the rest of the application, it would be very convenient if we could just fetch a single record from the file on the basis of a name instead of an account number. Unfortunately, this won't usually be possible, because names are a notorious problem. They cannot be depended on to be unique, they vary enormously in format and length, and spelling is a great challenge. That, in fact, is exactly why we assign an account number to each customer and use it as the file key, instead of using the one identifier that is most natural (and that the customer is least likely to forget).

It isn't usually possible to guarantee a unique response to a request that specifies a name, because we can't depend on that name being unique (and the user may even have misspelled it). What we want to do, then, is to give the users who need this facility some way to get to the right account number by entering a name. Suppose that our response to such a request is a list of customer names, in alphabetical order, starting with the first one that matches the requested name, up to the capacity of the screen.

In fact, since the user may be uncertain of the spelling, we'll treat the name entered as a generic or partial name, and show all the names that start in the way specified. So, if the user enters "Adams," the response will begin with the Adamses and continue with the Adamsons. But if the name were one that had several common spellings, such as "Reid" (also often "Reade"), then the user could enter just "Re" and get both forms. We can treat the first name similarly. The user could enter the first name (or initial) if known, to limit the number of responses, but we won't make this mandatory.

In our example, remember, we learned from our user survey that the Customer Service people are going to be the heaviest users. Most of their transactions will be inquiries by name. Moreover, most of these inquiries involve just three items besides the name: DFHP1CDU inquires by name, it makes sense to display these items along with the name and account number. That way these users will usually see all the data they want on the first response, without having to go on to ask for the detailed display of one particular record.

Sometimes, of course, they will want to see the whole record, and the Accounting Department will want this facility as well. So we must provide some easy way to get from the summary display to the other transactions that the users might want to do, once they have the account number. Suppose we use the remaining lines on the menu screen to display the results of a name search when one is requested. After a search, the users can then enter the request directly, without changing screens, on the menu to which they are accustomed. Figure 12 shows how the expanded menu screen might look:



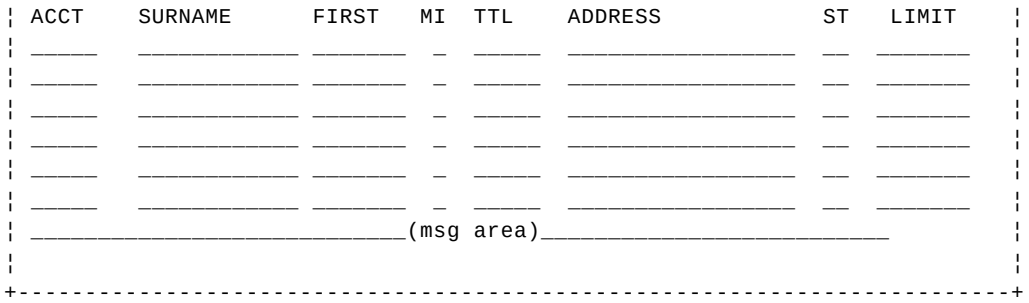


Figure 12. An expanded menu screen

2.3.3 Some interface design principles

In reaching our current idea of how our user interface will look, we've based most of our decisions on what is easiest for the user. Indeed, that should be the cardinal rule. Human time has become so much more valuable than computer time that it is worth a lot of effort and coding to make the user as productive as possible.

It isn't always obvious how to do this to best advantage, and what is best for one user may not be best for another. This applies especially to occasional users of an application. In fact, the style of conversation between users and computers has changed significantly as people have learned more about the "human factors" aspect of online systems.

The advent of sophisticated terminals, like those in the 3270 system, has also had an enormous effect in this area, as it became practical to deal with users in ways not possible with earlier devices. The whole idea of using a menu, for example, came much later than the original release of CICS, and depends explicitly on the characteristics of the 3270 for success.

Though there are no hard-and-fast rules, and though there can be many good designs for the user interface, there are five guidelines that we can safely propose:

1. Make screens easy to understand

Keep to the rules used in forms design: try to give the screen layout an uncluttered appearance and, to the extent possible, a columnar structure, so that the reader's eye moves easily from one item to the next and doesn't have to jump long distances.

Put a title on the screen, so that users know where they are in the current transaction.

Be consistent from screen to screen. If you put the title on the top center of one screen, put it there on all the screens. If you put the messages at the bottom of one screen, put them there on all the screens.

If the user will be reading from a form for input to a screen, make the screen look as much as possible like the form. Put the fields in the same order, and use the same placement as far as possible.

Likewise, if a screen is used to display information that the user is accustomed to seeing printed on a form, make the screen resemble the form as nearly as possible.

2. Cut down what the user must remember

If there are more than a few fields to be filled in, use a formatted screen with labels and instructions.

Where possible, put instructions on the screen to show what the user can do next.

Use consistent procedures, both within and across application programs. For example, if the **CLEAR** key is used to cancel in one transaction, use it that way in all transactions.

3. Protect users from themselves

If a user is about to do something that's hard to undo, such as a file deletion, get the user to confirm that it's the right deletion.

4. Save the user's time and patience

Minimize the number of characters that have to be keyed.

Make the user change screens as little as possible.

Make it as easy as possible to correct errors. There are many ways to do this. In our application, for example, we stick to the following:

- We redisplay the user's input in the same screen as the one in which it was entered.
- We diagnose all the errors at once (to the extent possible).
- We highlight fields that have errors.
- If the user misses any required fields, we fill them with asterisks and highlight them.
- We place the cursor under the start of the first field in error.
- We display an explanatory message if the error may not be obvious.

Place the cursor where the user will probably want to key first.

Minimize the number of times that the users have to skip over fields.

5. Reassure users

Give a positive confirmation that a requested action has been done successfully.

When you know a particular response time is likely to be longer than usual (because of the operation being performed) consider sending an intermediate display.

2.4 Coming to grips with the data

Having decided what you want to do, you can now determine what data will be required to do it and how to organize that data.

Subtopics

2.4.1 The account file

2.4.2 Recovery requirements

2.4.1 The account file

In this application, we know that we need access to all the fields that make up records in the existing account file, because this is the data that we intend to maintain and display. We need direct access to these records by account number for several of the required operations (display, add, and so on). Happily, this file exists in a form directly usable by CICS (a VSAM key-sequenced data set (KSDS), with the exact key that we need). This isn't pure luck or coincidence. The account number is the natural key for this file, and a VSAM key-sequenced data set is a good choice for a mixture of sequential and direct processing, such as probably occurs now in the batch programs that already use this file. Figure 13 shows the record format for this file.

Field	**Length**	**Occurs**	**Total**
Account Number (Key)	5	1	5
Surname	18	1	18
First Name	12	1	12
Middle initial	1	1	1
Title (Jr, Sr, and so on)	4	1	4
Telephone number	10	1	10
Address line	24	3	72
Other charge name	32	4	128
Cards issued	1	1	1
Date issued	6	1	6
Reason issued	1	1	1
Card code	1	1	1
Approver (initials)	3	1	3
Special codes	1	3	3
Account status	2	1	2
Charge limit	8	1	8
Payment history:	(36)	3	108
-Balance	8		
-Bill date	6		
-Bill amount	8		
-Date paid	6		
-Amount paid	8		

Figure 13. Account file record format

Subtopics
2.4.1.1 Access by name

2.4.1.1 Access by name

As well as accessing the account file records by account number, we need to access them by a second key--the customer name. There are many ways of achieving an alternative path into a file. For example, VSAM provides a facility called an **alternate index**, which can be used in CICS. CICS supports the **DATABASE 2** relational product, and IMS/DB DL/I. These systems provide powerful cross-indexing facilities, and they have many other features that reduce the coding required in user applications. They support complex data structures, provide increased function, and simplify the maintenance of file integrity. If you have data that you need to access by more than just a few different key fields, or if you have data that does not arrange itself into neat units like the account records in this application, you should evaluate seriously the use of a database system.

However, all these database products are beyond the scope of this Primer. For our application we'll use a simple technique, frequently used and quite appropriate to an application of this size. We'll build a small separate file, in name sequence order, to use as an index into the account file.

This is probably going to offer us better performance for sequential browsing of customer names than, say, an alternate VSAM index.

Subtopics
2.4.1.1.1 Choosing the file organization
2.4.1.1.2 Name index records
2.4.1.1.3 Choosing a control interval (CI) size

2.4.1.1.1 Choosing the file organization

For the initial read, we'll need direct access to the index file when we process an inquiry by name. After that, we'll read sequentially until we have enough names to fill one screen. So VSAM key-sequenced organization (2) is appropriate to this file as well as to the account file. ("Other file services" in topic 3.4.4 lists the other file access methods supported in CICS. VSAM KSDS is widely applicable, however, and is the only one covered in this book.)

(2) File organization, of course, isn't generally chosen by an application programmer, but by the application designer.

2.4.1.1.2 Name index records

What do we need in our name index records? We need the surname, clearly, and the first name. We need the account number, for access to the main file and to ensure a unique key. This is all we really need. However, since we're maintaining our own index file, we've the option of putting more than pointers into it. Let's see what else we can usefully put into the name index file.

In our application, we could produce the display shown in Figure 12 in topic 2.3.2.3 in two different ways:

> Read the name from the name index record and, for each name, use th account number in the index to access the account file. This can get us the address, the account status, and the charge limit.
>
> Repeat the address, the account status, and the charge limit field within the name index file. We'd then only need to access the name index file (and not the account file) to get these items.

In the second case, the index records would be a little larger as a result, and we'd have two copies of some fields (a potential source of trouble in large file-based systems). On the other hand, we could avoid one read for every name in the response to a name inquiry.

This latter point turns out to be important. In VSAM, one read brings a whole **control interval** (CI) of data into virtual storage. CICS passes to your program only the particular logical record that your program asked for, but on your next program read, CICS can return your record directly, without another VSAM read, if the record is in the same control interval. When you are reading in key sequence, the probability of the record being in the same control interval is very high. In our example, we'll be going through the name index records in name order, and the records are small, so we can expect there to be only one physical read for several logical reads.

However, if we needed to access the account file once for each of these reads, there would probably be a physical read to that file for every logical read to the index file, as we wouldn't be reading the account file in sequential (customer number) order.

In deciding which method to choose, we must weigh the cost of the many additional reads against file space and against the possible complications of keeping the two files synchronized. Changes that will have to be made to the batch billing and payment system need to be evaluated as well. If searching by name were an infrequent request, or if any of these other factors had a large cost associated, we might choose the first method. However, for our example we'll assume that this isn't so and, since inquiry by name will be by far the most frequent transaction, we'll include these fields in the index.

-- Fig 'INXFMT' unknown -- shows a reasonable layout for the name index record:

Field	**Length** (in bytes)	
Surname	12	These two fields form
Account Number	5	the key.
First name	7	
Middle Initial	1	
Title	4	
Street Address	24	
Account Status	2	
Charge Limit	8	

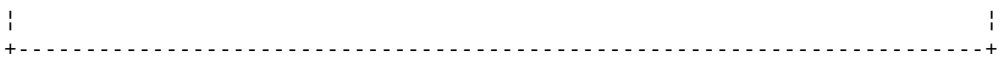


Figure 14. The name index record format

The first two fields together form the key. It will be unique because account numbers are unique, and it will allow us to search by surname, using a partial key of variable length. Notice that we chose field lengths for the surname and first name that were shorter than the corresponding fields in the account file. We also included only one line of the address. This keeps our index records reasonably small and lets us display a name index record on a single line of the screen. We can afford to do this because our purpose is to help the user in recognizing the right name, not to account for all the possibilities that can occur in names and addresses.

2.4.1.1.3 Choosing a control interval (CI) size

One of the issues in designing VSAM files is choosing control interval sizes for the data and the index. The choice depends partly on the fit of records into the CI, but it also depends on whether the data will be accessed directly or sequentially. In our example, the account file will always be accessed directly. That is, there is little or no chance of reading account records in account number order. So a large data control interval will hurt rather than help us. It will mean larger buffers (more demand for virtual storage), and more data will be transferred than can be used (the larger the interval, the more records transferred in one read). Therefore a small data CI is appropriate for this file.

In contrast, the name index file will be read sequentially more often than directly. The first read in a name inquiry will of course, be random, but after that we'll tend to read several records in sequence. Therefore it will be helpful to get many logical records in a single physical read, and so we'll choose a large data CI size for the name index.

All these physical reads are done by CICS using VSAM. Your program is concerned only with logical reads, which are completely unaffected by CI size. So you don't **have** to think about these factors. However, a good application designer will try to take all such factors into consideration. While learning, you can certainly put off the choice of the "best" CI size until your program is working. After all, you can change the CI sizes of your files without changing your application code or your CICS tables, and you may wish to do this later if trying to tune your system.

2.4.2 Recovery requirements

One of the first requirements for the example application was to maintain the integrity of the account file. We'll see in "Pseudoconversational or not?" in topic 2.7 how CICS prevents the loss of integrity associated with partially completed transactions, and we'll use this feature to keep the two files (the name index file and the account file) properly synchronized. However, we must also protect the account file from disasters such as a head crash.

In a batch environment, you can keep an extra copy of an important file, or keep enough information to recreate it (by keeping back versions, for instance, with the inputs to the update runs). In an online environment, this isn't so easily done. You cannot copy the file after every update. Nor can you afford to lose all the updates since the last time you copied the file. These updates were entered at terminals by many different users, who may not remember what stage they had reached when you last secured the file, who may not have ready access to the input documents any longer, and who will certainly be very cross if they have to rekey a large number of transactions.

CICS solves this problem by using a variation on the batch technique. If you have a file that must be protected, you ask CICS to **journal** the updates. CICS then keeps a copy of every change made to the file on a tape or disk. It logs these changes on the system log, which is journal number one. If you lose a file, you go back to the most recent copy of it and recreate it from that. Then you run a program that applies the changes recorded on all the journals created since that copy was made.

In our example application, the account file is clearly a file that must be protected in this way. In contrast, the index file does not require these precautions. We do have to protect its integrity from partially completed transactions, just as we do the account file. However, we can always recreate the index file from the account file with a very simple batch program (the CICS tape includes the source code of a program called **ACCTINDX** to do this--see Appendix A, "Getting the application into your CICS system" in topic A.0) so it isn't necessary to journal the changes to it, nor even to make periodic backup copies.

2.5 Refining the transaction design

We've now looked at several principles that we need to bear in mind when working on application programs for online transactions. Next, let's have a closer look at what we have to do to accomplish the functions that make up our example. Some people just write out, in English, the transaction flow. Others prefer flowcharts. You'll find both in this topic.

Now that we've decided to give the user a "menu" screen, we'll start by displaying this menu and analyzing the request entered on it. After that we'll describe the requirements according to the type of request (add, display, and so on).

Subtopics

2.5.1 Request analysis

1. Display the menu screen, (as shown in Figure 11 in topic 2.3.2.1)

2. Wait for the user to enter a request

3. Analyze the request, which may be:

    a. To leave the application entirely

    b. To add, modify, delete, display, or print a record

    c. To search on a name

    d. None of the above.

4. Process according to the type of request.

        In case a above, simply return control to CICS.

        In cases b and c, process as described later.

        If the request cannot be deciphered (case d), send an error message to the user. Then go back to step 2 to wait for the user to correct the input. (When it arrives, repeat the processing from step 3 above.)

Figure 15. Request analysis

2.5.2 Add processing

1. Check the customer account number that was entered along with the request. It must be present, and:

   a. Numeric

   b. In the proper range (we'll assume the Accounting Department restricts numbers to the range from 10 000 to 79 999)

   c. Not already used (that is, not already in the file).

   If any of these conditions isn't met, send a message to the user saying what is wrong. Then go back to step 2 of "Request analysis" in topic 2.5.1 to wait for the corrected input. When it arrives, processing will resume at step 3 of that process, so that the user has a full range of choices at this point. That is, the user can correct the add request, change to a different type of request, or quit the application entirely.

2. If the account number is acceptable, send a skeleton screen (see Figure 10 in topic 2.3.1.3) back to the terminal so that the user can fill in the fields for the new record.

3. Wait for the user to enter the data (or to signal a desire to quit by using the **CLEAR** key).

4. See whether the user wants to continue this operation. (He or she might have had trouble entering this particular record or had a change of mind.) If the user doesn't want to go on, display the menu screen again with a message like "previous request cancelled" and go to step 2 of "Request Analysis" to wait for the next request to come in.

5. Otherwise, check the fields read from the filled-in data entry screen for reasonableness and consistency. If there are errors, send a message back to the terminal saying what the errors are, and go back to step 3 to wait for the next input.

6. If no errors are detected in the input, update the files:

   a. Write an image of the new record to the change log.

   b. Build a new account record using the information from the input screen, and add this record to the file.

   c. Build the corresponding name index record and add this to the name index file.

7. Redisplay the menu screen, with a message to say what has just been done, and resume at step 2 of "Request Analysis."

Figure 16. Add processing

2.5.3 Modify processing

1. Check the account number that is entered along with the request. It must be present, and:

   a. Numeric
   b. In the proper range (10 000 to 79 999)
   c. Already on file.

   Just as in the add processing, if any of these conditions isn't met, send a message to the user saying what is wrong, and then go to step 2 of "Request analysis" in topic 2.5.1 to await corrected (new) input.

2. Build a display of the current contents of the record from the information on file, and send it to the user's screen.

3. Wait for the user to enter the changes (or to indicate, with the **CLEAR** key, a desire to abandon the transaction).

4. If the user doesn't want to continue, send a fresh menu screen with a message acknowledging the cancellation and then go to step 2 of "Request Analysis" to wait for the next request.

5. Build a new version of the record by applying the changes entered on the screen to the old version of the record.

6. Check that the old record hasn't been updated in the meantime.

7. Check all items in the new record for reasonableness and consistency with each other. If there are errors, send the input screen back to the terminal with all the errors noted. Also, if there are no differences between the new record and the old one, send a message noting this (the user may have made an error and should be notified). Treat this situation just like an error in a data item. Return to step 3 to await corrected input.

8. If there are no errors in the input, update the files:

   a. Write a record of the changes (that is, images of the old and new records, plus an indication of the changed areas) to the change log.

   b. Replace the old record in the file with the new version.

   c. If the changes affected the corresponding index record, replace that record, too, with a revised version.

9. Redisplay the menu screen, with a message to say what has just been done, and resume at step 2 of "Request Analysis."

Figure 17. Modify processing

2.5.4 Delete processing

1. Check the account number entered with the request; the requirements and the error processing are the same as for "Modify processing" in topic 2.5.3.

2. Build a display of the contents of the record from the information in the account file and send this to the terminal.

3. Wait for the user to confirm or cancel the delete request.

4. See if the user has decided to cancel the delete request. If so, proceed as in step 4 of "Add processing" in topic 2.5.2.

5. If the user has not cancelled, see whether he or she has confirmed the delete request. If not, send a message asking the user either to confirm or cancel, and go back to step 3.

6. If the delete request is confirmed, update the files:

   a. Write an image of the deleted record to the change log.

   b. Delete the record from the account file.

   c. Delete the corresponding name index record from that file.

7. Redisplay the menu screen, with a message to say what has just been done, and go back to step 2 of "Request analysis" in topic 2.5.1 to wait for the next request.

Figure 18. Delete processing

2.5.5 Display processing

1. Check the account number entered with the request; the requirements and the error processing are the same as for "Modify processing" in topic 2.5.3.

2. Build a display of the contents of the record from the information in the account file, and send it to the screen.

3. Wait for the next input from the terminal (indicating that the user has finished looking at the display), and then go back to step 1 of "Request analysis" in topic 2.5.1.

Figure 19. Display processing

2.5.6 Print processing

1. Check the account number entered with the request; the requirements are the same as for a "modify" request. Also check the name of the printer entered with the request. It must be present and must correspond to the name of a real printer known to CICS. If either input item is in error, send an appropriate message to the terminal and return to step 2 of "Request analysis" in topic 2.5.1 to await corrected input.

2. Build a display image of the contents of the record from the information in the account file, (printers understand the same data streams that displays do).

3. Send this image to the indicated printer.

4. Send a message to the terminal, saying that the print request has been processed; then go back to step 2 of "Request analysis" in topic 2.5.1 to await the next request.

Figure 20. Print processing

2.5.7 Name inquiry processing

1. Check the name search input:

   The surname must be present and alphabetic.
   The first name must be alphabetic, if present.

   If either condition isn't met, send an error message to the terminal and go back to step 2 of "Request analysis" in topic 2.5.1 to wait for corrected input or another request.

2. If the names are correct, find the first index file record that has a surname that matches the (full or partial) surname specified in the input, or which is just higher in the alphabet than the input surname.

3. Build the search output part of the display, one line at a time.

   a. Read the next record in the index file.

   b. See if this record meets the input criteria for the given name. If it does, build an output line from it.

   Repeat this step (building one line at a time, remember) until the surname read from the file is higher in the alphabet than any that would match the input surname, or the end of the file is reached, or all the output lines have been used.

4. Send the completed output to the screen.

5. Wait for the user's next request.

6. If the next input shows that the user wants to continue the search, go back to step 2, using as a starting point the last record read in producing the previous display.

7. If the user doesn't want to continue, go to step 3 of "Request analysis" in topic 2.5.1 to find out what he or she wants to do instead.

Figure 21. Name inquiry processing

2.5.8 Printing the change log

1. Read the first (next) record from the log.

2. Write the information read to the log printer.

3. Repeat steps 1 and 2 until there are no more records on the log.

4. Delete the log records once they have been printed.

You'll find more information about both this change log and the error log in "Program ACCT03: requests for printing" in topic 2.10.4.

Figure 22. Printing the change log

2.5.9 Printing the error log

1. Read the first (next) record from the log.

2. Write the information read to the log printer.

3. Repeat steps 1 and 2 until there are no more records on the log.

4. Delete the log records once they have been printed.

You'll find more information about both this error log and the change log in "Program ACCT03: requests for printing" in topic 2.10.4.

Figure 23. Printing the error log

2.5.10 Summary

We've now seen the requirements for the various functions our users can perform at (or, in the case of printing, from) their terminals.

The next thing we need to do is to consider how to break up these functions into CICS transactions, and what factors affect program design in a CICS environment.

2.6 Programming for a CICS environment

The overall design goals in an online environment are the same as those in a batch environment: to provide as much service (do as much useful work) as possible while using as little resource as possible.

Deciding what services to provide is, as we noted in "Defining the problem" in topic 2.1.1, the first step in the design. It takes a little experience and experimentation in online programming to know what additional services you can provide at reasonable cost, beyond simply replacing batch services with equivalent online services.

In our example, for instance, we decided initially to replace the function of the old printed account listing with the ability to display individual records on the screen. Originally, we had no plans to allow users to print individual records, even though it seemed an obvious feature to provide, once a user pointed out how useful it would be. This kind of interaction with potential users is invaluable in arriving at a design that is good from the user's point of view. It should be repeated often in the design cycle, as your insight into the application and the programming requirements develops.

Subtopics

2.6.1 Resources

After deciding what to do, what resources do we have to conserve while providing this function? Some of them are the traditional ones that are common to both batch programming and online programming:

Processor storag
Processor tim
Auxiliary storage space and transmission capacity to it

Others are new, and require some new considerations in design. They are:

User time and good humo
One-user-at-a-time resources, such as terminals, file records scratch-pad areas, and so on
Line transmission capacity

Let's take these individually and develop some guidelines for designing and programming CICS applications from them. Remember, there's bound to be conflict from time to time when trying to save one resource at the "cost" of another. The appropriate compromises will vary from one program to the next.

Subtopics
2.6.1.1 "Traditional" resources
2.6.1.2 Resources specific to working online

2.6.1.1 "Traditional" resources

First, the resources common to both batch and online programming.

Subtopics
2.6.1.1.1 Processor storage
2.6.1.1.2 Processor time
2.6.1.1.3 Auxiliary storage

2.6.1.1.1 Processor storage

Your applications use up processor storage in two ways. First, there are the CICS control blocks associated with any transaction being processed, and second, there is the program, or programs, being executed to accomplish the transaction. The programs, in turn, take up space both for executable code and for working storage areas. In an online system, the storage needs for these purposes constantly come and go. They exist only for at most the duration of a transaction, and so in assessing storage needs, we have to consider not only how much, but for how long. The trade-off between space and time is complex, but at a minimum we can say:

+--- **Processor storage guidelines (1)** ----------+
¦ ¦
¦ Keep programs short. ¦
¦ ¦
¦ Keep **WORKING STORAGE** short. ¦
¦ ¦
¦ Keep programs short in duration of use. ¦
¦ ¦
+----------+

How transactions use storage over time is taken up again in "Pseudoconversational or not?" in topic 2.7.

We should also note that CICS is a virtual storage system, and the good coding practices (whether COBOL or otherwise) observed in batch programming for a virtual storage environment apply equally well to CICS. These include:

+--- **Processor storage guidelines (2)** ----------+
¦ ¦
¦ Keep **GOTOs** to a minimum. ¦
¦ ¦
¦ Place subroutines near the code that **PERFORMs** or otherwise calls them. ¦
¦ ¦
¦ Avoid long searches for data. ¦
¦ ¦
+----------+

Some remarks about PERFORM: Having mentioned subroutines, let's stay with them for a few moments. COBOL programmers learning CICS often ask about the pros and cons of using **PERFORMs** in CICS.

First of all, using **PERFORM** to execute a COBOL subroutine is very much more efficient than the CICS overheads associated with linking to, or transferring control to, another program. However, repeating the subroutine in each of your COBOL application programs is going to cost you more storage. That is, if you're using **PERFORM** for repeated code, you're trading space against (possible) paging.

Like earlier COBOL compilers, the VS COBOL II compiler allows a COBOL program to use **CALLs** to external routines, but now the called routines can issue CICS commands. This avoids the CICS overheads of transferring control between programs, but it does mean link-editing the routines with every calling program.

We've some more to say in the next part of the Primer (in "The COBOL CALL statement" in topic 3.6.2.4).

Secondly, the matter also arises in COBOL loop situations. You see, COBOL doesn't let you put the **PERFORM** which controls the loop physically adjacent to the actual code of the loop, unless you cheat and use a **GOTO** rather unnaturally. **PERFORMs** are OK for loops, but always keep the code you **PERFORM** as near as you can to the controlling **PERFORM** statement, to

minimize the risk of the two things being in separate pages of storage.

Finally, the question of a **PERFORM** also crops up with regard to code that isn't a true "subroutine" in the old-fashioned sense, and code which the programmer never really considered breaking off as a separate (sub)routine.

This kind of **PERFORM** comes from some of the structured programming rules, where you **PERFORM** blocks of code (often physically distant in the program, with attendant paging implications) for reasons of neatness, readability, maintainability, and so on. The response time impact of flipping through a lot of pages is of course much more critical in a real-time environment than in batch, because you have to compete with all those other terminal users instead of just a few other jobs.

Our "PERFORM" guidelines

- Use **PERFORMs** to help structure your code (but watch out for increased paging).
- Keep **PERFORMed** code as close as possible to the **PERFORM** statement.
- Use **PERFORM** for long code, or code used in a great many places.

2.6.1.1.2 Processor time

In general, we need to conserve processor time in CICS in the same way as in a batch program. The major factor is exactly the same: calls for operating system services take much longer, relatively speaking, than straight application code. This is true whether you are coding in CICS, where a call takes the form of a CICS command, or in batch COBOL, where a call is implicit in your input-output statements (**OPEN, READ, WRITE**, and so on). So, avoiding unnecessary commands in a CICS design will reduce processor time much more than fine tuning your COBOL code, just as avoiding a single input/output operation in a regular program will make up for many **MOVEs** and **GOTOs.**

It is never desirable to do long calculations (matrix inversion and such) in an online program. This is because any online program is sharing the processor with many other programs (or occurrences of the same program) servicing users who each think they have the full attention of the "computer." Fortunately, such long calculations are rarely needed in online programs.

The highest cost of CICS programs is incurred by maintenance. Since structured code is easier to maintain, it may well be worth incurring higher paging rates because of PERFORMs. For example, one section that performs input/output to a file, rather than having 20 copies of the same code is much easier to modify if the file organization changes.

+--- **Processor time guidelines** --+
¦ ¦
¦ Avoid unnecessary CICS commands. ¦
¦ ¦
¦ Avoid excessively long calculations. ¦
¦ ¦
+---+

2.6.1.1.3 Auxiliary storage

Disk space and transfer capacity are optimized in an online system in the same way as in a batch system. What differs is the following. In a batch system, the system programmer arranges data sets on disk according to what *jobs* might run concurrently. In an online system, however, the system programmer arranges data sets according to what *transactions* might execute concurrently. The same techniques are used for tuning: statistics on device and channel utilization in combination with knowledge of the applications.

2.6.1.2 Resources specific to working online

This brings us to the new considerations.

Subtopics
2.6.1.2.1 User time and good humor
2.6.1.2.2 One-user-at-a-time resources
2.6.1.2.3 Line transmission capacity

2.6.1.2.1 User time and good humor

We've already seen (in "Some interface design principles" in topic 2.3.3) how user time and aggravation can be minimized. You'll find our guidelines there.

2.6.1.2.2 One-user-at-a-time resources

The next candidates for conservation are a whole class of resources that can be used by only one user (one transaction) at a time. A file record is a perfect example of this type of resource. As we've noted several times, we do **not** want two transactions updating the same record at the same time. CICS provides the enqueue mechanisms to prevent conflicts between transactions over such resources. What you have to remember in designing a transaction is that when one user has access to such a resource, everyone else who wants it will have to wait. Therefore:

+--- **Exclusive-use resource guideline** ----------+
¦ ¦
¦ Minimize the duration of transactions that require exclusive use of ¦
¦ resources. ¦
¦ ¦
+---+

We'll say some more about these resources in later topics.

2.6.1.2.3 Line transmission capacity

The last new element on our list is line transmission capacity. In an online system with terminals located a long way from the processor, the signals between them are generally (although not invariably) carried over the public voice telephone network. Compared to most of the elements of a computing system, telephone lines are very slow indeed. Transmission time, especially over a congested line, may be a major component of the total response time. Therefore:

Line transmission guideline

Avoid sending unnecessary data to and from screens.

For the most part, CICS does this for you automatically, using the 3270 hardware features explained in "3270 terminals" in topic 2.2. Sometimes, however, you can help as well. For example, if you were writing a data entry application program in which the operator repeatedly filled in the same screen, you would not need to rewrite the constant information on the screen (the titles and field labels) after the first display. It would be well worth your while to add a little extra program logic, to distinguish between the screen for the first entry and that for subsequent entries, and thereby reduce line traffic by not resending data that is already on the screen.

2.7 Pseudoconversational or not?

Now that we've established guidelines for design, let's return to the problem of defining the transactions that make up the example application. In "Refining the transaction design" in topic 2.5, we described the processing required for the various transaction types that the user sees: add, modify, display, and so on. If we were to define our CICS transactions along these functional lines, we can foresee several problems:

There is much repetitive code, which suggests that we should at leas use common programs for some of the transactions, if not combine some transactions.

Every transaction involves a wait for the user to enter data, and th update transactions contain two such waits. This means that these transactions will be running for a relatively long time, which is a violation of the guideline to keep program duration short.

The modify and delete transactions will be holding on to one-user-at-a-time resource during one of the waits, contradicting the guideline to minimize the duration of transactions that use such resources.

Let's dodge the first problem for a moment, and look at the other two, which bring up an important issue in CICS design.

Take, for example, the modify transaction. If programmed as outlined earlier, the sequence of major events would be as shown here in Figure 24:

Operations
1. Display menu screen.
2. Wait for response.
3. Receive menu screen (which is presumed to contain a correct modify request).
4. Read the subject record from the account file.
5. Display the record in formatted form.
6. Wait for the user to enter changes.
7. Receive the changes.
8. Write changes to the printed log.
9. Update the account and index files accordingly.
10. Redisplay the menu screen.

Figure 24. The conversational sequence of the modify transaction

Subtopics

2.7.1 Conversational transactions
2.7.2 Pseudoconversational transactions
2.7.3 Maintaining file integrity

2.7.1 Conversational transactions

In CICS, this is called a **conversational transaction**, because the program(s) being executed enter into a conversation with the user. A **nonconversational transaction**, by contrast, processes one input (which was read by CICS and which was what started the task), responds, and ends (disappears). It never pauses to read a second input from the terminal, so there is no real conversation.

There are important differences between the two types: for example, duration. Because the time required for a response from a terminal user is much longer than the time required for the computer to process the input, conversational transactions last that much longer than nonconversational transactions. This means, in turn, that conversational transactions use storage and other resources much more heavily than nonconversational ones, because they hold on to their resources for so long. Whenever one of these resources is critical, you have a compelling reason for using nonconversational transactions if possible.

2.7.2 Pseudoconversational transactions

This led to a technique in CICS called **pseudoconversational** processing, in which a series of nonconversational transactions gives the appearance (to the user) of a single conversational transaction. In the case we were just looking at, the pseudoconversational structure is shown in Figure 25:

Transaction		**Operations**
First	1.	Display menu screen.
Second	3.	Receive menu screen.
	4.	Read the subject record from the account file.
	5.	Display the record in formatted form.
Third	7.	Receive the changes.
	8.	Write changes to the printed log.
	9.	Update the account and index files accordingly.
	10.	Redisplay the menu screen.

Figure 25. The pseudoconversational structure

Notice that steps 2 and 6 of the conversational version have disappeared. No transaction exists during these waits for input; CICS takes care of reading the input when the user gets around to sending it.

A word about "transactions". If we seem to be using the word in two different ways, well ... yes we are. We defined the word earlier in the way that the user sees a transaction: a single item of business, such as an add, a display operation, and so on. This is a correct use of the word. However, what the user sees as a transaction isn't necessarily what CICS sees.

To CICS, a transaction is a task that begins (usually on request from a terminal), exists for long enough to do the required work, and then disappears. It may last milliseconds or it may last hours. As we've just explained, you can use either one or several CICS transactions to do what the user regards as a single transaction. We're still deciding what we should define to CICS as transactions to accomplish the user transactions in our example problem. At the moment, the pseudoconversational approach seems promising; it will use shorter programs, which are desirable in CICS, and although there may be more of them, the programming does not look any more complicated.

There is a second important issue in this choice of techniques, however. It brings up a characteristic of the conversational transaction that can be both a significant advantage and a serious disadvantage. This characteristic is the length of the transaction, and it affects both file integrity and the ownership of resources that other transactions may need.

2.7.3 Maintaining file integrity

We said earlier (in "Recovery requirements" in topic 2.4.2) that CICS has facilities for maintaining the integrity of files and other resources that are important enough to protect. CICS does two things:

1. It makes sure that file modifications for a transaction are either executed completely or not at all. For example, if a transaction has to update two related files and, after updating the first, finds it cannot do the second, then CICS undoes (**backs out**) the first update. We'll make use of this feature in our example application. If the application changes the account file and then discovers that someone has closed the index file by the time it goes to make the corresponding change there, CICS automatically removes the update to the account file.

2. It makes sure that protected resources (records in protected files, protected scratchpad areas, and so on) are updated by only one transaction at a time, and that any transaction updating such a resource finishes completely before a second transaction gets access to that resource.

Let's reexamine the conversational or pseudoconversational issue in view of this new information. We've been insisting that we do not want two users to update the same record at once. If we use a single conversational transaction for our modify, CICS will prevent this from happening (that's good). When we issue the read (for update) in this sequence, CICS will prevent any other task from writing this record. If a second task comes along and requests the same record, for update, CICS will suspend that task until the first one is finished.

However, the program being executed in this second transaction won't be notified that it is going to get suspended, and so the user won't know why the request is taking longer than usual (that's bad).

To be honest, it's a little more complicated than that...

Both CICS and VSAM get involved in protecting the file from concurrent updates. VSAM's mechanism is based on the control interval, and has this effect: while one transaction is updating a record, no other transaction can update any record *in the same control interval*. Furthermore, other transactions may not even be able to *read* a record in the same CI as the one being updated. (3) Moreover, the wait experienced by the second transaction may be substantial; it will last as long as it takes the first user to enter the modifications on the screen. If he or she should leave the terminal before finishing, or go through a lot of error cycles getting the input correct, the wait may be very long indeed.

(3) Whether a second transaction can read a record in the same control interval depends on whether the file is using local shared resource (LSR) or nonshared resource (NSR). For NSR only, a second task can perform a simple read (but not a read-for-update) on a record in the same control interval.

Subtopics
2.7.3.1 Double updating...
2.7.3.2 ...and how to avoid it

2.7.3.1 Double updating...

If we choose the pseudoconversational technique, this waiting problem disappears, but so does the protection. In this case, the second transaction in the pseudoconversational sequence could issue the same "read" as in the conversational form. But as soon as this transaction ends, CICS releases the record, long before the update process is complete. A second user can come along and request the same record. Then you have two users making changes on the basis of the same "old" copy of the record. Changes made by the first user will go into the file, but then changes from the second user will go into the file right over the first user's, and the first set of changes will be lost (that's very bad).

Now clearly, in our application, we can separate off the first part of our user transaction (the first transaction in the pseudoconversational sequence) because we're not yet dealing with any protected resources. Nothing is done in this step that a later failure would have to undo. But what about the rest of it? We're caught between two unfortunate alternatives. If we use a conversational approach, there will be greater use of storage and, worse, occasional unexplained waits. If we use a pseudoconversational approach, we may compromise file integrity.

There's no easy way to get around the unexplained waits of the conversational approach, but there **are** ways to get around the integrity problem, with a little extra coding.

For example, suppose that as soon as a user asked to update an account number, we made a note in a scratchpad area. (CICS provides scratchpad facilities for keeping track of things *between* transactions.) We can leave the number there until the update is entirely completed and then erase it. In our example, this means that we write a scratchpad record in the second transaction, and erase it in the third. Before we start any update request, we can check to see if the number is in use. If it is, we can tell the user this and ask him or her to resubmit the request later. Furthermore, we can let the user **display** the record even if it is in use.

This isn't quite all, however. Because CICS ensures that transactions are either done completely or not at all, we have to make sure that all our protected resources get updated in what CICS regards as a single transaction to ensure file integrity. In the conversational case, this takes care of itself, as there is only one transaction. In the pseudoconversational case, the files are all updated in the third transaction (good), but the scratchpad is updated in two different transactions (not so good). If the second transaction is completed successfully, but something happens to the third, the scratchpad record is written but not erased. Our files would be okay, which is the main thing, but we'd be unable to update the record involved until we could somehow reset the scratchpad.

2.7.3.2 ...and how to avoid it

We'll get around this by designing a slightly more sophisticated scratchpad mechanism. We can, for instance, put a limit on the time for which a transaction can "own" an account number. Then an accident in the third transaction or thoughtless behavior by a user (going to lunch in the middle of a modification) will not cause an account record to become unusable for more than a short period of time. All this involves extra coding and complications, however. Is it worth it?

In this example, it really isn't obvious whether conversational or pseudoconversational is the better choice (after the menu phase, in which being pseudoconversational is definitely better). The choice really comes down to how many of these transactions we might expect at once. If there were a great many, the storage burden of a conversational transaction alone might cause us to choose pseudoconversational. If there were only a modest number, then we would have to consider how often a user would experience the unexplained wait if we chose conversational. If nearly all the activity consisted of displaying and printing, with only an occasional update, then the conversational approach might still be the correct choice.

We'll assume here, however, that there are enough transactions with enough updates to justify choosing the pseudoconversational approach, and we'll program our own mechanism for avoiding concurrent updates.

Double updating is one of those problems you can tackle in a variety of ways. We've chosen a scratchpad (partly because it's a reasonable method, and partly because it's going to allow us to show you how to use a CICS facility called temporary storage). A drawback of our scratchpad, however, is that **all** future (and, as yet, unknown) transactions that update the account file will have to refer to this scratchpad. We'll mention an alternative solution in "The need for scratchpad and queuing facilities" in topic 3.5.1.

2.8 Arranging the processing

We've now reached the point where we can start to arrange the processing described earlier into transactions and programs. Remember, a CICS transaction uses one or several programs to do its work. When a transaction is invoked, CICS looks in its list of installed transaction definitions to find out which program should be executed *first* to accomplish that transaction. However, that program may invoke any number of other programs. Several transactions may use the same program or programs, in the same order or in a different order.

Subtopics

2.8.1 Defining the transactions

Let's first look at the transactions we'll need, and then we can assess what programs we'll require. Because we're going to use the pseudoconversational approach, we need transactions that take an input from the screen, process it, and write back either the final result or an intermediate result ready for the next transaction.

Subtopics
2.8.1.1 Displaying the menu
2.8.1.2 Analyzing the user's response
2.8.1.3 Adding a new record
2.8.1.4 Handling updates and other requests

2.8.1.1 Displaying the menu

The first thing we need is a very simple transaction that will accept a request to get started: that is, one that will put the menu up on the screen.

2.8.1.2 Analyzing the user's response

Once the menu is on the screen, we need a transaction to analyze and respond to the input request that comes in after the user has completed fields on the menu screen. Going back to "Refining the transaction design" in topic 2.5, we see that this transaction must do the following steps:

Steps 3-4 of "Request analysis" in topic 2.5.1

Steps 1-2 of "Add processing" in topic 2.5.2

Steps 1-2 of "Modify processing" in topic 2.5.3

Steps 1-2 of "Delete processing" in topic 2.5.4

Steps 1-2 of "Display processing" in topic 2.5.5

Steps 1-4 of "Name inquiry processing" in topic 2.5.7

All steps of "Print processing" in topic 2.5.6.

Remember, we don't have to do all this processing with a single program. We'll decide on the programs we need later, after we've laid out the transactions.

2.8.1.3 Adding a new record

The next transaction that we need is one to do steps 4 through 7 of "Add processing" in topic 2.5.2. We'll use this transaction if the request in the previous transaction was to add an account record.

2.8.1.4 Handling updates and other requests

Similarly, we'll need four other transactions to do, respectively, the steps shown below:

Steps 4-8 of "Modify processing" in topic 2.5.3

Steps 4-7 of "Delete processing" in topic 2.5.4

Steps 6-7 of "Name inquiry processing" in topic 2.5.7

All steps of "Printing the change log" in topic 2.5.8.

We might use a separate transaction for each of these requirements, or we might combine some of them. We won't make that decision for the time being.

2.8.2 Defining the programs

Let's look at the programs that we're going to need in support of these transactions, because that will help us to decide how many different transaction types we need.

Subtopics
2.8.2.1 Displaying the menu--ACCT00
2.8.2.2 Analyzing the user's response, ACCT01
2.8.2.3 Handling updates (including additions)--ACCT02

2.8.2.1 Displaying the menu--ACCT00

Let's go back to the first transaction, the one that puts up the menu screen, and give it a name so that we can refer to it easily. We'll need a four-character transaction identifier to define it to CICS anyway, so let's call it, say, **ACCT.** This is what the terminal user will key in to see the menu screen for this application. Now **ACCT** needs a program that will display the menu screen. This program is so simple that perhaps it should be combined with some other program, but for clarity we'll keep it separate. Let's call this program **ACCT00.**

2.8.2.2 Analyzing the user's response, ACCT01

The next transaction is the one that processes the menu input. Let's also give it a name, say, **AC01.** It's a good idea to use some sort of naming convention for both your transactions and programs. You should be able to tell which application they belong to just by their names. There's a temptation when writing your first application to use names like **MENU, ADD,** and **UPDT.** These turn out to be unfortunate choices when you get around to doing your *second* application, however, and so names that identify the application are generally better.

ACCT is the only transaction identifier the general user will have to remember, so we'll start the others with **AC** for ease of recognition, and just number them from there. Similarly, the programs will start with **ACCT** and be numbered.

But back to transactions. Let's see the processing that **AC01** has to do, to help us visualize the programs required. Looking back at the list of requirements, one approach would be to write a separate program for each item on the list. The first program (the one that did the initial request analysis) would transfer control to one of the others, depending on the type of request. However, if we look at the content of Steps 1 and 2 of "Add Processing," "Display Processing," "Modify Processing," "Delete Processing", and "Print Processing," we find that they are very similar. They start with the same data and access the same file record, so we probably want to combine these into a single program. So we can cut down our original list for this transaction to:

Steps 3-4 of "Request analysis" in topic 2.5.1

Steps 1-2 for add, modify, delete, display, and print

Steps 1-4 for "Name inquiry processing" in topic 2.5.7

Steps 3-4 for "Print processing" in topic 2.5.6.

None of these is a very long piece of code, so it will probably be most convenient to put them in the same program. For the moment we'll call this program **ACCT01.** However, it may turn out later that it's better to break out one or more of these segments of code into additional programs. Program and transaction structures often become clearer when you start to code, and you may find that you can come up with a better structure than your original one once you start. Don't worry if everything isn't obvious at first; it takes practice.

2.8.2.3 Handling updates (including additions)--ACCT02

For the transactions that follow transaction **AC01** and finish the processing for adds, modifies, and so on, we again might consider a separate program for each type of function. Once more, however, it's obvious that the processing for adds, modifies and deletes is very similar. Most of the steps, in fact, are identical. So, let's combine these into a single program, and call it **ACCT02.** Then we can use a single transaction for all three processes. We could use different transactions, all using the same program, but it would be pointless in this case. Let's assign the identifier **AC02** to the transaction that gets executed when the user has filled in an update screen (add, modify or delete).

We still need a transaction that will do the remaining steps of "Name inquiry processing" in topic 2.5.7. But this code will be almost identical to an initial name search request, so we can probably include it in program **ACCT01.**

2.8.3 Summary

To summarize, so far we've defined three transactions, and three programs in support of them, as shown in Figure 26:

Identifier	**Transaction**	**Programs Used**
ACCT	Displays menu.	ACCT00
AC01	Analyzes requests; Processes name search, display and print requests;* Does first part of update requests	ACCT01
AC02	Completes update requests	ACCT02

* Almost, as we'll see later

Figure 26. The three transactions and three programs

The only thing left is the printing of the log. Or is it? In fact, in addition to printing, we haven't yet thought much about the business of telling the user at the terminal about any error conditions that may arise. So before we consider the log, we shall digress to discuss three considerations that will bear on our definition of transactions and programs.

These are:

Communication between transaction

Error handlin

The relationship between transactions and terminals

These bear directly on how we'll handle the last two application functions.

2.9 Three remaining considerations

Subtopics

2.9.1 Communication between transactions

You may have noticed when we were explaining pseudoconversational processing that there seemed to be some gaps in control and communication.

When one transaction of a pseudoconversational sequence has been completed, doesn't this task disappear when control goes back to CICS? And if so, how can we make sure that the transaction we intend to follow this one is actually the one that gets executed? And how will the next transaction know what this one was doing? When transaction **AC02** is supposed to follow **AC01,** for example, doesn't **AC02** need to know what kind of an update has been requested and *which* record was being updated?

Yes, CICS does indeed effectively erase all the storage associated with a transaction when it ends, and it often erases the program as well. However, before it passes out of existence, the departing transaction is allowed to pass data forward to be used by the next transaction initiated *from the same terminal*, whenever that transaction arrives. It is also allowed to specify *what* that next transaction should be. You can see that this is a very useful--indeed, vital--facility for pseudoconversational programming. It's what allows us to ensure that transaction **AC01** always follows **ACCT,** that **AC02** follows **AC01** when we're updating, and so on. It's called, not surprisingly, the "next transaction identifier" feature. We'll shorten this to "next transid."

The main way one transaction passes data forward to the next is by using the **COMMAREA** (for **communication area**). The same facility is available to pass data between programs within a transaction. We'll see how to use it for both purposes in "Application programming" in topic 3.0.

There are other facilities for storing data between transactions as well. One of these is a CICS facility known as **Temporary Storage**, which can be used as a sort of application scratchpad. This facility will do nicely for keeping track of the account numbers being updated. We'll see how to use it in "Application programming" in topic 3.0.

A less obvious place to store data between transactions is the screen itself. You may recall from our discussion of the 3270 data stream (see "3270 terminals" in topic 2.2) that the modified data tag governs whether or not a field on the screen is transmitted back to the processor. One way to ensure that an item of data gets from one transaction to another, then, is simply to store it on the screen, with the modified data tag on and the field protected, so that the user cannot change it. You can even prevent users from seeing the data (if that might confuse them), by using the dark attribute.

This method isn't appropriate to large amounts of data, of course, because we don't want to send *much* extra data over a communications link.

2.9.2 Handling errors and exceptional conditions

Before we get down to specifying our programs, we need to say a few preliminary words about errors and error recovery. Recovering from errors in online programs is a topic given a complete guidance book in the CICS library--the &rrgc.. For now, however, we'll just state some guidelines here before we start to specify our programs. (We'll return to the topic from the point of view of our example application in "Errors and exceptional conditions" in topic 3.8.)

We can divide the errors that can occur in a CICS transaction into five categories:

1. **Conditions that aren't normal from CICS's point of view but that are expected in the program.**

   There's an example in transaction **AC01,** when we test to be sure the record to be added isn't already there and get the "not found" response.

   Errors in this category should be handled by explicit logic in the program.

2. **Conditions caused by user errors and input data errors.**

   We'd have an error of this kind in our example application if a user tried to add an account number that already existed, or used the wrong key to send the data on the screen.

   Errors in this category should also be handled by explicit logic in your program. Ideally, no errors of either of these types should be allowed to stop the program, or do anything else to upset the user.

3. **Conditions caused by omissions or errors in the application code.**

   These may result in the immediate failure of the transaction (**ABEND**) or simply in a condition that we believed "could not happen" according to our program logic. In our example application, a "duplicate record" response in **AC02,** on adding a record to the account file, would represent this kind of error. We don't expect it, because we've already tested in transaction **AC01** to ensure that no record with the same key is in the file.

   For errors in this category, you'll want to terminate your transaction abnormally, in case CICS doesn't do it for you first. The resulting dump should enable you to find out why the condition occurred, and we'll give you more guidance on this in the CICS/ESA Problem Determination Guide. One of the main goals of the debugging process should be to get rid of this type of error.

4. **Errors caused by mismatches between applications and CICS tables, generation parameters, and JCL.**

   An example is when CICS responds "no such file exists" to your read or write request. When you are first debugging an application, these problems are almost invariably your fault. (This may sound harsh, but we're afraid it's true.) Perhaps the entry got left out of the File Control Table, or you spelled a name differently in the table from the program, or asked for the wrong set of services in selecting CICS modules.

   These conditions sometimes occur after the system has been put into use, as well. In this stage they are usually the result of changes to a CICS table, or an installed definition, or services parameters, or JCL, usually related to some other application.

This category needs the same treatment as the third while you are debugging. Once the program is in actual use, however, something more is needed when one of these conditions arises. You must give users an intelligible message that they or their supervisors can relay to the operations staff, to help in identifying and correcting the problem. For example, if a machine room operator has disabled a file for some reason and forgotten to reopen it, you want a message that says that the problem is caused by a disabled file (and *which* file, of course). Moreover, you should program for these eventualities right away, as this part of the program will need debugging just as well as the rest.

5. **Errors related to hardware or other system conditions beyond the control of an application program.**

   The classic example of this is an "input/output error" while accessing a file.

   As far as the application programs are concerned, this category needs the same treatment as the fourth. Systems or operations personnel will still have to analyze the problem and fix it. The only differences are that they probably didn't cause it directly, and it may take much more effort to put right.

The need to produce an appropriate message when an error in one of these last two categories occurs (or when one in category 3 slips through the debugging) will mean an additional program in our example application.

Subtopics
2.9.2.1 A "catch-all" error program--ACCT04

2.9.2.1 A "catch-all" error program--ACCT04

Since there are CICS commands in every program, we'll need this message logic in each. Rather than repeat the code in each, we'll put it in a separate program (**ACCT04**). This will not only avoid repetition, but will remove a long section of rarely-used code from the mainline programs. (The code itself isn't long, but the error message tables are.)

2.9.3 Transactions and terminals

There's one additional complication to think about in defining our transactions for this application program. This is the relationship between transactions and terminals in CICS. As we explained earlier, most CICS transactions (tasks) are invoked when CICS receives unsolicited input from a terminal. On receiving such input, CICS creates a task to process it. Which type of task is determined from the transaction identifier at the start of the input or the next transid that was set by the previous transaction at this terminal.

The task and the terminal that invoked it have a special relationship in CICS: the task essentially "owns" the terminal for its duration; it can write to it and read from it directly, and no other task can do so during this time. Conversely, the task owns *only* this terminal and cannot read from or write to any other terminal directly (another task might own that terminal at the time, and a sudden message from a second task might disrupt the owning task hopelessly).

You may be asking at this point "how can transaction **AC01** in the example do all the steps of print processing?" as we proposed earlier, since step 3 of "Print processing" in topic 2.5.6 (send this image to the indicated printer) seems to violate this restriction. The answer is that it can't. The same task cannot own the display terminal from which the input was received and a printer terminal.

Subtopics
2.9.3.1 A printer program--ACCT03

2.9.3.1 A printer program--ACCT03

What we do to get around this restriction is to have transaction **AC01** do the other steps of the print processing and then create a second transaction (task), which *does* own the necessary printer terminal, to do step 3. CICS provides a command called **START** expressly for this purpose, as we'll see in "Application programming" in topic 3.0. So we must add another transaction to our list, namely the one that does step 3 of "Print processing" in topic 2.5.6. Let's call it **AC03.** We'll also need a program to go with it, albeit a very short one; this we'll call **ACCT03.**

Now clearly the same problem will arise with printing the log of changes to the account file. The input that invokes this transaction is clearly not going to come from the terminal required to execute it (printers not being strong on input) and so again we'll need two transactions. One will accept the request from an input terminal and start a second, which will have the necessary printer at its disposal.

Let's call this first transaction **ACLG** and the second **AC05.** (We're reverting to a transaction identifier that's easier to remember, because the supervisor will have to remember it.)

Finally, we'll have transactions **ACEL** and **AC06.** **ACEL** will accept the input request, and will start **AC06** to print the error log.

We'll also need a program for each of these transactions. We could define a separate one for each, but the code required for these functions turns out to be so short, in fact, that we'll include it in the little program we defined for transaction **AC03,** and use a single program for four different functions.

Figure 27 shows the program structure we've now arrived at. The five programs in support of these transactions are examined one last time in "Defining the programs--a final look" in topic 2.10. You can either read this topic to consolidate your ideas about the programs, or move straight on to the next part of the Primer: "Application programming" in topic 3.0.

Id	**Transaction**	**Programs Used**
ACCT	Displays menu	ACCT00
AC01	Analyzes requests; processes name search and display requests fully; does first part of update and print requests	ACCT01/04*
AC02	Completes update requests	ACCT02/04*
AC03	Completes print requests	ACCT03/04*
ACLG	Invokes AC05	ACCT03/04*
AC05	Prints the log	ACCT03/04*
ACEL	Invokes AC06	ACCT03/04*
AC06	Prints the error log	ACCT03/04*

* Note: ACCT04 is used only if an error occurs.

Figure 27. The six transactions and five programs

2.10 Defining the programs--a final look

We've now defined five programs in support of our transactions. In this topic, we'll describe briefly what each program does. This material repeats that in "Refining the transaction design" in topic 2.5, but it's arranged somewhat differently. Feel free to move on to "Application programming" in topic 3.0 if you already feel comfortable with the program structure that we've defined.

Subtopics

2.10.1 Program ACCT00: menu display

This program is the first one executed when transaction **ACCT** is entered. It displays the menu screen, which prompts the operator for request input, and then ends (it returns control to CICS). In returning, it specifies that transaction **AC01** is to be executed when the next input is received from this terminal, which means that program **ACCT01** will be invoked to process the input from the menu. The processing steps are:

1. Display the menu on the screen

2. Go back to CICS, setting the next transid to **AC01.**

2.10.2 Program ACCT01: initial request processing

This program analyzes requests that are entered through the menu screen (all requests except those for printing the log). It processes name search and record display requests completely, does update requests up to the point where the user has to enter more information, and does print requests except for the step that requires access to a printer terminal. It's the first program invoked when transaction **AC01** is executed. The main steps in the program are:

1. Find out what the user wants to do. This involves looking at the input, both the actual data and the attention identifier (the key used to send the data). The possibilities are:

   a. A request to leave the application (indicated by use of the **CLEAR** key). Here control is returned to CICS, without any next transid.

   b. A request to cancel the previous (partially completed) request and start again with a menu screen. This means sending a new menu screen and then returning control to CICS with the next transid set to **AC01** (so that this same program will process the input from that menu when it arrives).

   c. A request to continue a name search that produced more matching records than would fit on a single screen (indicated by the user pressing the **PA2** key to move on from the current (full) screen, and view more records on the next). In this event, processing resumes at step 5, using search control information that was saved in the **COMMAREA** when this transaction was last executed for this terminal.

   d. A corrected request or a completely new request.

2. For a new request, get the input and examine the contents. The first decision is whether the user wants a name search or one of the other functions.

3. If the user entered a name, check it for reasonableness. If there's an error, write the appropriate error information to the screen and return control to CICS. Once again, set the next transid to **AC01,** so that this same program will get invoked to process the corrected input.

4. If the names are correct, build the control information we need to do the search, namely:

   An index file key that is equal to or just before the input in alphabetical sequence, so that we know where in the file to start reading,

   A limiting value for that key to tell us when we've read too far (alphabetically) in the file, and

   A range of alphabetical values for the given name, so that we can exclude records which do not meet that criterion, if any was specified.

5. Point to the first eligible record in the index file and begin reading sequentially. For each record read, check to see if the given name is within the required range. If it is, build an output line for the screen from the information in the record and then go on reading. If not, skip the record and go on reading. Continue this process until the surname in the file exceeds the one we're looking for, or the end of the file is reached, or there is no more room on the screen.

6. When this happens, send the results back to the user. If we ran out of space on the screen, add a message saying that there are more names and that they can be seen by using the **PA2** key. Then return control to CICS, again setting the next transid to **AC01.** If there are more matching names, save the search control information in **COMMAREA** as well.

7. If the request was other than a name search (display, print, add, modify, delete, or even an error), check the request type, account number and printer name (if applicable) for correctness. Checking the account number involves reading the account file. We check to make sure the record isn't there for an add request but that it *is* there for all the other request types. If any of the checks fail, or if the request itself is unrecognizable, write the appropriate error information back to the screen and return to CICS, once again with the next transid set to **AC01.**

8. If the request is an update (add, modify or delete), read the scratchpad to ensure that no other terminal is currently updating the same account number. If one is, treat the situation as an error in the account number and proceed as in the previous step. Otherwise, write the necessary scratchpad record to reserve the number for this terminal.

9. Build a screen image to send to the user (or the printer). For add requests, this will simply be a skeleton screen, with only the account number filled in. For the others, however, it will involve moving the information from the account file record (read in step 7) into the detail screen. Also, the title, message area and certain other items in the screen need to be customized to the particular type of request.

10. For all requests except print requests, send this screen back to the input terminal. Then return to CICS. The next transid for display requests will be **ACCT,** as the next thing the user will want after looking at the record is a fresh menu screen. For the update requests, the next transaction should be **AC02.**

11. For print requests, ask CICS to start another task (**AC03**) with the required printer as its terminal. Pass the screen image built in step 9 as data to that task. Then add a message to the menu currently on the screen saying that the printing has been scheduled, and return to CICS. Set the next transid to **AC01,** as the menu is still on the screen and therefore the next input should be processed by this same program.

2.10.3 Program ACCT02: update processing

ACCT02 is the first program invoked by transaction **AC02.** It completes update transactions, using the information supplied by the user on the detail screen. The main steps are as follows:

1. Make sure that the user wants to complete the update request. (It is important in a situation like this to allow users some means of escape, in case they change their mind about a file update they started or in case they simply don't have the right information to complete it. This application observes the convention that using the **CLEAR** key at any time means that the user wants to cancel the current operation.)

   If the user wants to quit, release control of the account number, send a fresh menu screen with a message that the previous request has been canceled, and return to CICS. Set the next transid to **AC01,** since the next input to be processed will come in on that menu screen.

2. Otherwise, get the input. If the request is to add a record, build a new record from the information on the screen. If the request is a modification, read the old record and build a new record by merging it with the changes entered on the screen.

3. Check the input for correctness. For delete requests, the only requirement is that the user confirm the deletion with a **Y** in the "verify" field. For add and modify requests, all the fields entered must meet their respective edit requirements. If there are any errors, send the appropriate error information to the screen. Then return control to CICS with the next transid set to **AC02,** so that this same program processes the corrected input.

4. Read the scratchpad to make sure that the input terminal still has control of the account number it is trying to update. (In other words, check that the scratchpad has neither been erased nor altered. Check back to "...and how to avoid it" in topic 2.7.3.2, if you need reminding about the scratchpad.)

   If not, treat the situation in the same way as an input error (see step 3 above), but with a different error message, of course.

5. Otherwise, write the update information to the log of changes. For additions, this will be an image of the new record. For modifications, it will be both the old and the new versions, and for deletions, it will be the record being deleted.

6. Do the actual updates. For adds, this means adding the new record to the account file and the corresponding index record to the index file. For deletes, it means removing a record from each file. For modifications, it means rewriting the record in the account file. The corresponding index record may have to be rewritten as well, depending on which fields in the account record changed. If the surname changed, for example, the old index record must be deleted and a new one added, because the key will have changed. (The first 12 characters of the surname, together with the account number, form the key, remember.)

7. Release ownership of the account number by erasing it from the scratchpad.

8. Send a fresh menu screen to the input terminal, with a message saying that the requested update has been completed. Then return control to CICS with the next transid set to **AC01.**

2.10.4 Program ACCT03: requests for printing

ACCT03 does several jobs, all related to printed output (as opposed to display output). When it is invoked by transaction **AC03,** it completes the request for printed output of a record in the account file. Transaction **AC01** processed the initial stages of the print request, checking the input, reading the record to be printed, and building the detail screen from the information in the file record. It then requested that transaction **AC03** be started with the required printer as its terminal. The processing in **AC03** is:

1. Retrieve the screen image prepared and saved for this purpose in transaction **AC01.**

2. Send this screen to the terminal owned by this transaction (the printer named by the user in the print request).

3. Return control to CICS. Don't set any next transid because there's no need to do so for terminals that never send unsolicited input. Also, we don't know what transaction should be executed next at this printer.

Transactions **ACLG** and **AC05** together process a user request to print the log of changes to the account file. The user invokes transaction **ACLG** directly, by entering this identifier at a display terminal. When invoked by **ACLG,** the program simply requests CICS to start transaction **AC05,** with the hardcopy printer as its terminal. **ACLG** then sends a message to the user saying that the printing has been scheduled, and returns control to CICS. No next transid is set, because we're not controlling the flow of transactions at the input terminal, as we do when input requests are entered through the menu screen.

Finally, transaction **ACEL** is a user request to print the error log. It does so by requesting that transaction **AC06** be started when the log printer is available. **AC06** transfers the error log data from temporary storage to the printer.

We'll format our log as follows:

For additions, we'll print the new record, using the same format that we use on the screen (the "detail" map).

For modifications, we'll print both the old version of the record and the new one, again using the map format. In the message area of the old record we'll note the areas that were changed (name, address, and so on), to make it easy for the supervisor to check.

For deletions, we'll print the old record.

For all types:

1. We'll note the contents of the screen in the title line of the map: **NEW RECORD** for additions, **BEFORE CHANGE** and **AFTER CHANGE** for the two images printed on a modification, and **DELETION** on a delete.

2. We'll show the time and date of the update and the name of the terminal at which it was entered. We'll put this information in the message area (for modifications, it will be in the "new" record image).

As a result of executing transaction **ACLG**, CICS starts **AC05** as soon as the requested printer is available. When invoked in this way, the program reads through the data set containing the hard-copy log sequentially, transferring each entry to the printer. After the last item is printed,

```
it deletes the log.  Then it returns control to CICS.  Again, no next
transid is set, because there's no need to do so for terminals that never
send unsolicited input.
```

2.10.5 Program ACCT04: error processing

This program is a general-purpose error routine. It isn't invoked directly by any transaction, but instead receives control from programs **ACCT01, ACCT02,** and **ACCT03** when they meet a condition from which they cannot recover. (Program **ACCT00** is so simple that no such situation arises.)

The program sends a screen to the terminal user (see Figure 28) with a text description of the problem and a request to report it. The text is based on the CICS command that failed and the particular error that occurred on it. The name of the transaction and the program (and if applicable, the file) involved are also shown. The command, error type, and program name are passed to **ACCT04** from the program which transferred control to it; we get the other items from the CICS **Exec Interface Block** (EIB). The EIB is a CICS control block associated with a task, containing information accessible to the application program. We'll look at it in more detail in "The EXEC Interface Block (EIB)" in topic 3.3.6.1.

After writing the screen, the program terminates itself abnormally (it **abends**), so that any updates to recoverable resources done in the half-completed transaction get **backed out**.

You'll see **ACCT04** in action in the EDF session described in "A session with EDF" in topic 5.1.3.1.7.

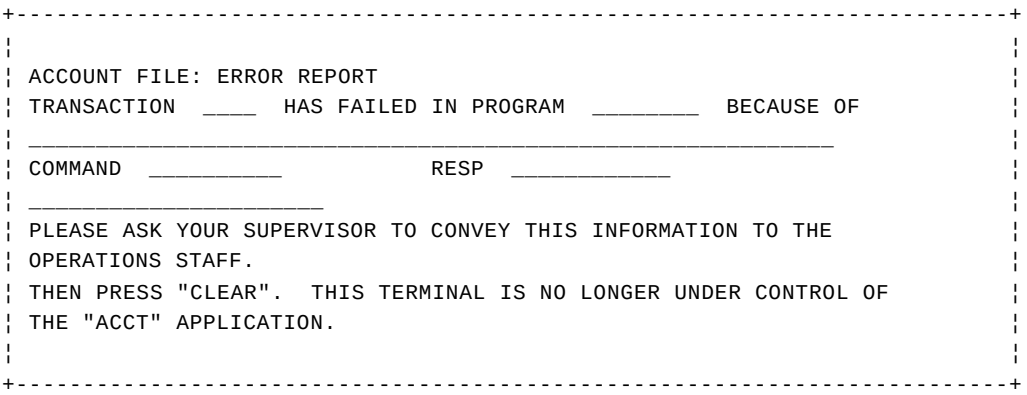


Figure 28. The transaction error screen

3.0 Application programming

This part of the Primer:

- Describes CICS COBOL application programs
- Examines the features of Basic Mapping Support (BMS)
- Deals with reading and writing files
- Explains a scratchpad mechanism that uses temporary storage
- Covers communication and control between application and tasks
- Explains how to use CICS commands such as **START** and **RETRIEVE**
- Covers errors and error recovery.

Subtopics

3.1 Writing CICS programs in COBOL

In this topic we'll begin by explaining the basic differences between batch and CICS programs. In later topics, we'll describe, by function, the services that CICS provides: first terminal services, then file services, and so on.

To show you how to use these services, we'll be coding parts of our example application as we go. In "The COBOL code of our example application" in topic 4.0, we list the programs in their entirety, with a step-by-step description of what the code does.

Appendix A, "Getting the application into your CICS system" in topic A.0 tells you where to find out how to prepare these programs for execution under CICS.

Subtopics

3.1.1 What's different about CICS programs?

Well, not **so** much **is** different. Here, for instance, are the steps a typical batch program goes through:

Operations
1. Initialize for the whole run (set all the counters to zero and open the files).
2. Initialize for the next input.
3. Read it.
4. Process it.
5. Write the related outputs.
6. Repeat Steps 2 to 5 until you run out of input.
7. Finish (add up the counters, print any summary results and close the files).

Figure 29. The steps of a typical batch program

A typical CICS transaction is very similar, but it includes only steps 2 through 5. That is, it's like the core of a batch program, where a single input is processed. CICS takes care of opening and closing the files for you. The reports and summaries associated with batch jobs can often be dispensed with in an online environment, or they may be produced periodically by a different transaction or even by a batch job.

The other big differences are:

- You request "operating system" services, such as file input/output, b issuing a CICS command instead of using the corresponding language facility (**READ, WRITE,** and so on).

- You aren't allowed to use the language facilities for which CICS ha provided substitutes.

- You cannot use language features and compiler options that nee operating system services during execution. The **SORT** and **TRACE** facilities are examples.

3.1.2 How to invoke CICS services

When you need a CICS system service, for example when reading a record from a file, you just include a CICS command in your code. Don't forget: throughout this book, we're only dealing with the command-level or exec-level programming interface. In COBOL, command-level CICS commands look like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS function option option ... END-EXEC.                        ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The "function" is the thing you want to do. Reading a file is **READ,** writing to a terminal is **SEND,** and so on.

An "option" is some specification that's associated with the function. Options are expressed as keywords, some of which need a value in parentheses after the keyword. For example, the options for the **READ** command include **FILE, RIDFLD, UPDATE,** and others. **FILE** tells CICS which file you want to read, and is always followed by a value indicating or pointing to the file name.

RIDFLD (record identification field, that is, the key) tells CICS which record and likewise needs a value. The **UPDATE** option, on the other hand, simply means that you intend to change the record (thereby invoking the CICS protections we discussed earlier) and doesn't take any value. So, to read, with intent to modify, a record from a file known to CICS as **ACCTFIL,** using a key that we've stored in working storage at **ACCTC,** we'd issue a command that looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS READ FILE('ACCTFIL') RIDFLD(ACCTC)                          ¦
¦             UPDATE  ...  END-EXEC.                                      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

When you specify a value, you may either use a literal, as we did for **FILE** above, or you may point to a data area in your program where the value you want is stored, as we did for **RIDFLD** above. In other words, we might have written:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   MOVE 'ACCTFIL' TO DSNAME                                              ¦
¦   EXEC CICS READ FILE(DSNAME) RIDFLD(ACCTC)                             ¦
¦             UPDATE  ...  END-EXEC.                                      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

instead of our earlier command. If you use a literal, follow the usual COBOL rules and put it in quotes unless it's a number. In other types of commands, these values may be paragraph names in your program, telling CICS where to go if a certain type of exceptional condition arises. Don't use quotes around paragraph names.

You may be curious about what the COBOL compiler does with what is (to it) a strange-looking English-like statement like the one above. The answer? The compiler doesn't *see* that statement. Processing a CICS program for execution starts with a translation step. The translator converts your CICS commands into COBOL, in the form of **CALL** statements. You then compile and link edit this in the normal way. The generated **CALL** statements never contain periods, by the way, unless you include one

explicitly after the **END-EXEC.** This means you can use CICS commands within **IF** statements (by leaving the period out of the command), or you can end a sentence with the command (by including the period).

3.1.3 Restrictions in CICS COBOL

1. The biggest difference between batch and CICS COBOL programs is that you don't define your files in a CICS program. Instead, they are defined using either RDO FILE definitions or DFHFCT macro statements that are stored in a CICS table, the file control table, which we cover in "Handling files" in topic 3.4. So:

   You cannot use the entries in the **ENVIRONMENT DIVISION** and the **DATA DIVISION** that are normally associated with files. In particular, the entire **FILE SECTION** is omitted from the **DATA DIVISION.** Put the record formats that usually appear there in either the **WORKING-STORAGE** or **LINKAGE** sections.

   You cannot use the COBOL **READ, WRITE, OPEN,** and **CLOSE** statements.

2. You cannot use compiler features that require the use of operating system facilities. For example:

   Special features of the COBOL compilers, namely:

   ACCEPT DISPLAY EXHIBIT REPORT WRITER
   SEGMENTATION SORT TRACE

   Features that require an operating system **GETMAIN** (the most common of which is **CURRENT-DATE**).

   Certain compiler options:

   COUNT ENDJOB FLOW DYNAM STOP RUN
   SYMDUMP STATE SYST TEST

3. Your program must be what CICS calls "quasi-reentrant." Technically, this means your program must not modify itself between calls for CICS services. For this purpose, in command-level CICS, your **WORKING-STORAGE** section is *not* considered part of the program (neither is anything in the **LINKAGE** section). Consequently, you rarely have a chance to break the "quasi-reentrant" rule.

4. There are significant differences between VS COBOL II and other levels of the COBOL language. For example, unless you are using VS COBOL II, the following restriction is in force:

   When separate COBOL programs are link edited together, only the first may invoke CICS services.

These are the major restrictions, and the only ones you are likely to encounter using the commands described in this Primer. The CICS/ESA Application Programming Reference contains definitive application programming interface information on this subject. We'll often cite this manual in this part of the Primer.

3.2 Defining screens with basic mapping support (BMS)

It may be that your DP department currently uses a screen definition program product such as Screen Definition Facility, or perhaps the screen painting facility of Cross Systems Product. However, we're going to assume you'll be using basic mapping support (BMS) and the BMS macros.

That said, let's now plunge in and try to code our example application. If we start at the beginning of the first program we specified (**ACCT00**), the first thing we need is to write a formatted screen to the input terminal. This requires the use of CICS terminal input/output services. In particular, we'll need to use Basic Mapping Support (BMS).

First, some background: CICS supports a wide variety of terminals, from teletypewriters to subsystems such as intelligent cluster controllers, under a variety of communications access methods. In this Primer, however, we cover only the most common CICS terminals, those of the IBM 3270 system. Specifically the 3277 and 3278 display devices (with a screen size of 24 lines and 80 columns) and the associated printer terminals: 3284, 3286, 3287 and 3289.

We don't use features that depend on a particular terminal access method, and we only cover formatted output. Nor do we cover many of the formatting services; instead we concentrate on the basic things you need to get an ordinary application going. After we've explained these fundamentals, we'll tell you what else you can do when you're feeling adventurous, and where to look for guidance on how to do it.

Subtopics

3.2.1 What BMS does

As you read through this topic (and the next) you may start to feel a bit overwhelmed by all the detail you'll be learning about BMS. So let's get a couple of things straight right from the word "go". BMS simplifies your programming job, keeping your code largely independent of any changes in your network of terminals and of any changes in the terminal types. And after you've written your first few maps, you'll find they aren't so bad!

Before we start to look at the BMS commands, we need to explain in a little more detail what BMS does for you. It's probably easiest to define what BMS does by examining the menu screen we need. You can see what it looks like in Figure 30.

To help us in this discussion, we've added row and column numbers to the figure and underlined the fields that would otherwise not show unless filled in with data. We've also marked the position of the attribute byte for the "stopper" fields with a vertical bar (|) and for other fields with a plus sign (+). These markers won't show up on the screen we're building; it will look just as it did in Figure 12 in topic 2.3.2.3.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦           1         2         3         4         5         6         7 ¦
¦  12345678901234567890123456789012345678901234567890123456789012345678901234567890 ¦
¦ 1+ACCOUNT FILE: MENU                                                   ¦
¦ 2                                                                      ¦
¦ 3   +TO SEARCH BY NAME, ENTER:                     +ONLY SURNAME      ¦
¦ 4                                                  +REQUIRED.  EITHER ¦
¦ 5      +SURNAME:+_________+  FIRST NAME:+________| +MAY BE PARTIAL.   ¦
¦ 6                                                                      ¦
¦ 7   +FOR INDIVIDUAL RECORDS, ENTER:                                   ¦
¦ 8                                                    +PRINTER REQUIRED ¦
¦ 9      +REQUEST TYPE:+_+  ACCOUNT:+___+  PRINTER:+__+ ONLY FOR PRINT  ¦
¦10                                                    +REQUESTS         ¦
¦11      +REQUEST TYPES:  D = DISPLAY    A = ADD     X = DELETE          ¦
¦12                      +P = PRINT      M = MODIFY                      ¦
¦13                                                                      ¦
¦14   +THEN PRESS "ENTER"            +-OR-   PRESS "CLEAR" TO EXIT       ¦
¦15                                                                      ¦
¦16+ACCT    SURNAME     FIRST   MI  TTL    ADDRESS             ST  LIMIT  ¦
¦17+_____   __________  _______  _  _____  __________________   __  _______¦
¦18+_____   __________  _______  _  _____  __________________   __  _______¦
¦19+_____   __________  _______  _  _____  __________________   __  _______¦
¦20+_____   __________  _______  _  _____  __________________   __  _______¦
¦21+_____   __________  _______  _  _____  __________________   __  _______¦
¦22+_____   __________  _______  _  _____  __________________   __  _______¦
¦23                                                                      ¦
¦24+_____________________________(msg area)_____________________________ ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 30. A detailed look at the menu screen

You define this screen with **BMS macros**, which are a form of assembler language. When you've defined the whole map, you put some job control language (JCL) around it and assemble it. You assemble it twice, in fact. One of the assemblies produces the **physical map**. This gets stored in one of the execution-time libraries, just like a program, and CICS uses it when it executes a program using this particular screen.

The physical map contains the information BMS needs to:

Build the screen, with all the titles and labels in their prope places and all the proper attributes for the various fields.

Merge the variable data *from* your program in the proper places on the screen when the screen is sent to the terminal.

Extract the variable data *for* your program when the screen is read.

The information is in an encoded form comprehensible only to BMS, but fortunately we never need to examine this ourselves.

The other assembly produces a COBOL structure which we call the **symbolic description map** or **DSECT** (an assembly language term for this type of data structure, standing for dummy control section). This structure defines all of the variable fields (the ones you might read or write in your program), so that you can refer to them by name. The data structure gets placed in a library along with similar **COPY** structures like file record layouts, and you simply copy it into your program.

3.2.2 The BMS macros

To show you how this works, let's go ahead and define the menu map. We'll explain the three map-definition macros as we go. Don't be put off by the syntax; it's really quite simple when you get used to it. We'll go from the inside out, starting with the individual fields.

Subtopics

3.2.2.1 The DFHMDF macro: generate BMS field definition

For each field on the screen, you need one **DFHMDF** macro, which looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ fldname DFHMDF POS=(line,column),LENGTH=number,                         ¦
¦                INITIAL='text',OCCURS=number,                            ¦
¦                ATTRB=(attr1,attr2,....)                                 ¦
¦ (You need a continuation character--any character except a space--in    ¦
¦ column 72 of each line except the last.)                                ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The items in this macro have the following meanings:

fldname
This is the name of the field, as you'll use it in your program (or almost so, as we'll explain). Name every field that you intend to read or write in your program, but don't name any field that's constant (**ACCOUNT FILE: MENU...** and other labels, or the stopper fields in this screen). The name must begin with a letter, contain only letters and numbers, and be no more than seven characters long.

DFHMDF
This is the macro identifier, which must be present. It shows that you are defining a field.

POS=(line,column)
This is the position on the screen where the field should appear. (In fact, it's the position relative to the beginning of the map. For the purposes of this Primer, however, screen and map position are the same.) Remember that a field starts with its attribute byte, so if you code **POS=(1,1),** the attribute byte for that field is on line 1 in column 1, and the actual data starts in column 2. For the type of maps in this Primer, you need this parameter for every field.

LENGTH=number
This is the length of the field, *not* counting the attribute byte. You'll have to specify length for the type of maps in this Primer.

INITIAL='text'
This is the character data for an output field. It's how we specify labels and titles for the screen and keep them independent of the program. For the first field in the menu screen, for example, we'll code:

```
INITIAL='ACCOUNT FILE:  MENU'
```

ATTRB=(attr1,attr2,...)
These are the attributes of the field, and there are four different characteristics you can specify. The first is the display intensity of the field, and your choices are:

NORM
Normal display intensity.

BRT
Bright (highlighted) intensity.

DRK
Dark (not displayed).

The second characteristic governs what the user can do at the

keyboard. Here your choices are:

ASKIP
The field cannot be keyed into, and the cursor will skip over it if the user fills the preceding field.

PROT
The field cannot be keyed into, but the cursor will not skip over it if the user fills the preceding field.

UNPROT
The field can be keyed into.

NUM
The field can be keyed into, but only numbers, decimal points and minus signs are allowed, if you have the **NUM LOCK** feature.

The third characteristic governs the modified data tag that we discussed in "3270 input data stream" in topic 2.2.4:

FSET
Turns on the modified data tag. This causes the field to be sent on the subsequent read whether or not the user keys into it. If you don't specify this, the field is sent only if the user changes it.

The fourth characteristic that you can specify as part of the "attributes" has nothing to do with the attribute byte on the screen. It gives you a way of specifying that you want the cursor to be in this field. To do so, code:

IC
Places the cursor under the first position of the field. Since there is only one cursor, you should specify **IC** for only one field. If you specify it for more than one, the last one specified will be the one used.

You don't need the **ATTRB** parameter. If you omit it, the field will be **ASKIP** and **NORM,** with no **FSET** and no **IC** specified. If you specify either the protection or the intensity characteristics, however, it will be clearer if you specify both, because the specification of one can change the default for the other.

OCCURS=number
This parameter gives you a way to specify several fields at once, provided they all have the same characteristics and are adjacent. If you specify a field of length 10 at position (4,1) that is **ASKIP** and **NORM** with **OCCURS=3,** you'll get three fields of length 10, autoskip and normal intensity, at positions (4,1), (4,12), and (4,23). This is an exception to the "one **DFHMDF** macro for every field" rule we gave you earlier.

Now we can define the fields in our menu map. We'll "do" the fields in order. Although this is no longer required in CICS, it's a good idea for clarity. Figure 31 shows the **DFHMDF** macros for the menu map.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ Col     Col    Col                                              Col     ¦
¦ 1       9      16                                               72      ¦
¦ *       MENU MAP.                                                       ¦
¦ ACCTMNU DFHMDI SIZE=(24,80),CTRL=(PRINT,FREEKB)                         ¦
¦         DFHMDF POS=(1,1),ATTRB=(ASKIP,NORM),LENGTH=18,          X       ¦
¦                INITIAL='ACCOUNT FILE: MENU'                             ¦
¦         DFHMDF POS=(3,4),ATTRB=(ASKIP,NORM),LENGTH=25,          X       ¦
```

```
                INITIAL='TO SEARCH BY NAME, ENTER:'
         DFHMDF POS=(3,63),ATTRB=(ASKIP,NORM),LENGTH=12,         X
                INITIAL='ONLY SURNAME'
         DFHMDF POS=(4,63),ATTRB=(ASKIP,NORM),LENGTH=16,         X
                INITIAL='REQUIRED. EITHER'
         DFHMDF POS=(5,7),ATTRB=(ASKIP,BRT),LENGTH=8,            X
                INITIAL='SURNAME:'
SNAMEM   DFHMDF POS=(5,16),ATTRB=(UNPROT,NORM,IC),LENGTH=12
         DFHMDF POS=(5,29),ATTRB=(PROT,BRT),LENGTH=13,           X
                INITIAL='  FIRST NAME:'
FNAMEM   DFHMDF POS=(5,43),ATTRB=(UNPROT,NORM),LENGTH=7
         DFHMDF POS=(5,51),ATTRB=(PROT,NORM),LENGTH=1
         DFHMDF POS=(5,63),ATTRB=(ASKIP,NORM),LENGTH=15,         X
                INITIAL='MAY BE PARTIAL.'
         DFHMDF POS=(7,4),ATTRB=(ASKIP,NORM),LENGTH=30,          X
                INITIAL='FOR INDIVIDUAL RECORDS, ENTER:'
         DFHMDF POS=(8,63),ATTRB=(ASKIP,NORM),LENGTH=16,         X
                INITIAL='PRINTER REQUIRED'
         DFHMDF POS=(9,7),ATTRB=(ASKIP,BRT),LENGTH=13,           X
                INITIAL='REQUEST TYPE:'
REQM     DFHMDF POS=(9,21),ATTRB=(UNPROT,NORM),LENGTH=1
         DFHMDF POS=(9,23),ATTRB=(ASKIP,BRT),LENGTH=10,          X
                INITIAL='  ACCOUNT: '
ACCTM    DFHMDF POS=(9,34),ATTRB=(NUM,NORM),LENGTH=5
         DFHMDF POS=(9,40),ATTRB=(ASKIP,BRT),LENGTH=10,          X
                INITIAL='  PRINTER:'
PRTRM    DFHMDF POS=(9,51),ATTRB=(UNPROT,NORM),LENGTH=4
         DFHMDF POS=(9,56),ATTRB=(ASKIP,NORM),LENGTH=21,         X
                INITIAL='       ONLY FOR PRINT'
         DFHMDF POS=(10,63),ATTRB=(ASKIP,NORM),LENGTH=9,         X
                INITIAL='REQUESTS.'
         DFHMDF POS=(11,7),ATTRB=(ASKIP,NORM),LENGTH=53,         X
                INITIAL='REQUEST TYPES: D = DISPLAY A = ADD  X = X
                DELETE'
         DFHMDF POS=(12,23),ATTRB=(ASKIP,NORM),LENGTH=25,        X
                INITIAL='P = PRINT      M = MODIFY'
         DFHMDF POS=(14,4),ATTRB=(ASKIP,NORM),LENGTH=18,         X
                INITIAL='THEN PRESS "ENTER"'
         DFHMDF POS=(14,35),ATTRB=(ASKIP,NORM),LENGTH=28,        X
                INITIAL='-OR-   PRESS "CLEAR" TO EXIT'
SUMTTLM  DFHMDF POS=(16,1),ATTRB=(ASKIP,DRK),LENGTH=79,          X
                INITIAL='ACCT    SURNAME       FIRST   MI  TTL   ADDRESS
                              ST      LIMIT'
SUMLNM   DFHMDF POS=(17,1),ATTRB=(ASKIP,NORM),LENGTH=79,OCCURS=6
MSGM     DFHMDF POS=(24,1),ATTRB=(ASKIP,BRT),LENGTH=60
```

Figure 31. The DFHMDF macros for the menu map

3.2.2.2 The DFHMDI macro: generate BMS map definition

Now that we've sorted out the middle of the map (all the fields) we need to wrap some control information around it. To start any map, you need a different kind of macro:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ mapname DFHMDI SIZE=(line,column),                                      ¦
¦                CTRL=(ctrl1,ctrl2,...)                                   ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The items in this macro are:

mapname
This is the map's name, which you'll use when you issue a CICS command to read or write the map. It's required. Like a field name, it must start with a letter, contain only letters and numbers, and be no more than seven characters long.

DFHMDI
This is the macro identifier, also required. It shows that you're starting a new map.

SIZE=(line,column)
This parameter gives the size of the map. You need it for the type of maps we're using. BMS allows you to build a screen using several maps, and this parameter becomes important when you are doing that. In this Primer, however, we'll keep to the simpler situation where there's only one map per screen. In this case, there's no point in using a size other than the screen capacity (that is, **SIZE=(24,80)** for a 3276, 3277, 3278, or 3279 Model 2).

CTRL=(ctrl1,ctrl2,...)
This parameter shows the screen and keyboard control information that you want sent along with a map. You can specify any combination of the following:

PRINT
Specify this for any map that might be sent to a printer terminal.

Since it costs nothing to add this (and it can cause a lot of grief if you accidentally omit it when you *do* need it), we always try to remember to specify it.

FREEKB
This means "free the keyboard."

The keyboard locks automatically as soon as the user sends any input to the processor, and it stays locked until some transaction unlocks it, or the user presses the **RESET** key. So you'll almost always want to specify **FREEKB** when you send a screen to the terminal, to save the user from having to press **RESET** before making the next entry.

ALARM
This parameter sounds the audible alarm at the terminal (if the terminal has this feature; otherwise it does nothing). You might want to use this when displaying an error map, for example. We chose not to.

The **DFHMDI** macro we need to start our menu map, which we'll call **ACCTMNU**, is shown in Figure 32:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ ACCTMNU DFHMDI SIZE=(24,80),CTRL=(PRINT,FREEKB)                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 32. The DFHMDI macro for the menu map

3.2.2.3 The DFHMSD macro: generate BMS map set definition

You can put several maps together into a **map set** and assemble them all together. In fact, all maps (even a single map) must form a map set. For efficiency reasons, it's a good idea to put related maps that are generally used in the same transactions in the same map set. All the maps in a map set get assembled together, and they're loaded together at execution time as well.

When you've defined all the maps for a set, you put another macro in front of all the others to define the map set. This is the **DFHMSD** macro:

```
setname DFHMSD TYPE=type,MODE=mode,LANG=COBOL,
               STORAGE=AUTO,TIOAPFX=YES,
               CTRL=(ctrl1,ctrl2,...)
```

The items in this macro have the following meanings:

setname
This is the name of the map set. You'll use it when you issue a CICS command to read or write one of the maps in the set. It's required. Like a field name, it must start with a letter, consist of only letters and numbers, and be no more then seven characters long.

Because this name goes into the list of installed program definitions, make sure your system programmer (or whoever maintains these lists) knows what the name is, and that neither of you changes it without telling the other. It's the load module name.

DFHMSD
This is the macro identifier, also required. It shows that you're starting a map set.

TYPE=type
TYPE governs whether the assembly produces the physical map or the symbolic description (**DSECT**). As we pointed out in "What BMS does" in topic 3.2.1, you do your assembly twice, once with **TYPE=MAP** specified and once with **TYPE=DSECT** specified. The **TYPE** parameter is required. See "Symbolic description maps (DSECT structures)" in topic 3.3.1.

MODE=mode
This shows whether the maps are used only for input **(MODE=IN),** only for output **(MODE=OUT),** or for both **(MODE=INOUT).**

LANG=COBOL
This decides the language of the **DSECT** structure, for copying into the application program. For the examples in this Primer, the language will always be COBOL. However, you can program in PL/I as well (in which case you would code **LANG=PLI**), or in assembler (**LANG=ASM**).

STORAGE=AUTO
For a COBOL program, this operand causes the **DSECT** structures for different maps in a map set *not* to overlay each other. If you omit it, storage for each successive map in a map set redefines that for the first map. If you don't use these maps at the same time, you should omit **STORAGE=AUTO** to cut down the size of your **WORKING-STORAGE.** However, when several maps are in the same map set, they're most likely to be used at the same time, and then you should specify **STORAGE=AUTO.** This is the case in the example application, where we use the menu and other maps in the same transaction.

CTRL=(ctrl1,ctrl2,...)

This parameter has the same meaning as in the **DFHMDI** macro. Control specifications in the **DFHMSD** macro apply to all the maps in the set; those on the **DFHMDI** macro apply only to that particular map, so you can use the **DFHMDI** options to override, temporarily, those of the **DFHMSD** macro.

TIOAPFX=YES

Always use this parameter in command-level programs, such as the ones we're writing in this Primer. See the paragraph beginning "The first 12 characters" in topic 3.3.1.2.

Since all the maps in the example application are used together in one transaction or another, we'll put them all into a single map set, and call it **ACCTSET**. The **DFHMSD** macro we need, then, is:

```
ACCTSET DFHMSD TYPE=MAP,MODE=INOUT,LANG=COBOL,
               STORAGE=AUTO,TIOAPFX=YES
```

The only thing now missing from our map definition is the control information to show where the map set ends. This is very simple: It's another macro, **DFHMSD TYPE=FINAL,** followed by the assembler **END** statement:

```
        DFHMSD TYPE=FINAL
        END
```

3.2.2.4 Rules on macro formats

When you write assembler language (which is what you *are* doing when using these macros) you have to observe some syntax rules. Here's a simple set of format rules that works. This is *by no means* the only acceptable format.

Start the map set, map, or field name (if any) in column 1

Put the macro name **DFHMDF, DFHMDI,** or **DFHMSD**) in columns 9 through 14 (**END** goes in 9 through 11).

Start your parameters in column 16. You can put them in any order yo like.

Separate the parameters by one comma (no spaces), but do not put comma after the last one.

If you cannot get everything into 71 columns, stop after the comm that follows the last parameter that fits on the line, and resume in column 16 of the next line.

The **INITIAL** parameter is an exception to the rule just stated, because the text portion may be very long. Be sure you can get the word **INITIAL**, the equal sign, the first quote mark, and at least one character of text in by column 71. If you can't, start a new line in column 16, as you would with any other parameter. Once you've started the **INITIAL** parameter, continue across as many lines as you need, using all the columns from 16 to 71. After the last character of your text, put a final quote mark.

Where you have more than one line for a single macro (because o initial values or any other parameters), put an **X** (or any character except a space) in column 72 of *all lines except the last*. This continuation character is very important. It's easy to forget, but this upsets the assembler.

Always surround initial values by single quote marks. If you need single quote *within* your text, use two successive single quotes, and the assembler will know you want just one. Similarly with a single "&" character. For example:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦        INITIAL='MRS. O''LEARY''S COW && BULL'                          ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

If you want to put a comment into your map, use a separate line. Pu an asterisk (*) in column 1, and use any part of columns 2 through 71 for your text. Do not go beyond 71.

3.2.3 Map definitions for the example

Now that we've all the information we need for building maps, and now that we've done the menu map, let's define the other maps and the map set we need for our example application.

Subtopics

3.2.3.1 Defining the account detail map

Figure 33 shows the map for displaying the detail in an account record. It's used for displaying and printing the record, and for additions, modifications, and deletions. As you can see, the attribute bytes are marked, and we've added line and column numbers as before.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦           1         2         3         4         5         6         7 ¦
¦  12345678901234567890123456789012345678901234567890123456789012345678901234567890 ¦
¦  1+ACCOUNT FILE:+RECORD DISPLAY                                        ¦
¦  2                                                                     ¦
¦  3+ACCOUNT NO:+____    SURNAME:  +_______________|                     ¦
¦  4                      FIRST:    +_________+  MI:+_+ TITLE:+____|     ¦
¦  5+TELEPHONE:+______   ADDRESS:  +____________________|                ¦
¦  6                               +____________________|                ¦
¦  7                               +____________________|                ¦
¦  8+OTHERS WHO MAY CHARGE:                                              ¦
¦  9+_____________________________|  +_____________________________|     ¦
¦ 10+_____________________________|  +_____________________________|     ¦
¦ 11                                                                     ¦
¦ 12+NO. CARDS ISSUED:+_+  DATE ISSUED:+__+__+__+  REASON:+_|           ¦
¦ 13+CARD CODE: C          APPROVED BY:+___|       +SPECIAL CODES:+_+_+_| ¦
¦ 14                                                                     ¦
¦ 15+ACCOUNT STATUS:+__+   CHARGE LIMIT:+__________|                     ¦
¦ 16                                                                     ¦
¦ 17+HISTORY:  BALANCE      BILLED     AMOUNT        PAID        AMOUNT   ¦
¦ 18         +_________      __/__/__   _________     __/__/__    _________ ¦
¦ 19         +_________      __/__/__   _________     __/__/__    _________ ¦
¦ 20         +_________      __/__/__   _________     __/__/__    _________ ¦
¦ 21                                                                     ¦
¦ 22+_____________________________(message area) _____________________   ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 33. The account detail map

Figure 34 shows the map definition for this screen; after the code there are notes on some of the macros.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ Col     Col    Col                                                 Col¦
¦ 1       9      16                                                  72 ¦
¦ *       DETAIL MAP.                                                   ¦
¦ ACCTDTL DFHMDI SIZE=(24,80),CTRL=(FREEKB,PRINT)                       ¦
¦         DFHMDF POS=(1,1),ATTRB=(ASKIP,NORM),LENGTH=13,              X  ¦
¦                INITIAL='ACCOUNT FILE: '                               ¦
¦ TITLED  DFHMDF POS=(1,15),ATTRB=(ASKIP,NORM),LENGTH=14,  1         X   ¦
¦                INITIAL='RECORD DISPLAY'                  2            ¦
¦         DFHMDF POS=(3,1),ATTRB=(ASKIP,NORM),LENGTH=11,              X  ¦
¦                INITIAL='ACCOUNT NO:'                                  ¦
¦ ACCTD   DFHMDF POS=(3,13),ATTRB=(ASKIP,NORM),LENGTH=5                 ¦
¦         DFHMDF POS=(3,25),ATTRB=(ASKIP,NORM),LENGTH=10,             X  ¦
¦                INITIAL='SURNAME:  '                      3            ¦
¦ SNAMED  DFHMDF POS=(3,36),ATTRB=(UNPROT,NORM,IC),        4         X   ¦
¦                LENGTH=18                                              ¦
¦         DFHMDF POS=(3,55),ATTRB=(PROT,NORM),LENGTH=1     5            ¦
¦         DFHMDF POS=(4,25),ATTRB=(ASKIP,NORM),LENGTH=10,             X  ¦
¦                INITIAL='FIRST:    '                                   ¦
¦ FNAMED  DFHMDF POS=(4,36),ATTRB=(UNPROT,NORM),LENGTH=12               ¦
¦         DFHMDF POS=(4,49),ATTRB=(PROT,NORM),LENGTH=6,    6         X   ¦
```

```
                INITIAL='   MI:'
MID     DFHMDF POS=(4,56),ATTRB=(UNPROT,NORM),LENGTH=1
        DFHMDF POS=(4,58),ATTRB=(ASKIP,NORM),LENGTH=7,              X
                INITIAL=' TITLE:'
TTLD    DFHMDF POS=(4,66),ATTRB=(UNPROT,NORM),LENGTH=4
        DFHMDF POS=(4,71),ATTRB=(PROT,NORM),LENGTH=1
        DFHMDF POS=(5,1),ATTRB=(ASKIP,NORM),LENGTH=10,              X
                INITIAL='TELEPHONE:'
TELD    DFHMDF POS=(5,12),ATTRB=(NUM,NORM),LENGTH=10
        DFHMDF POS=(5,23),ATTRB=(ASKIP,NORM),LENGTH=12,             X
                INITIAL='  ADDRESS:  '
ADDR1D  DFHMDF POS=(5,36),ATTRB=(UNPROT,NORM),LENGTH=24
        DFHMDF POS=(5,61),ATTRB=(PROT,NORM),LENGTH=1
ADDR2D  DFHMDF POS=(6,36),ATTRB=(UNPROT,NORM),LENGTH=24
        DFHMDF POS=(6,61),ATTRB=(PROT,NORM),LENGTH=1
ADDR3D  DFHMDF POS=(7,36),ATTRB=(UNPROT,NORM),LENGTH=24
        DFHMDF POS=(7,61),ATTRB=(PROT,NORM),LENGTH=1
        DFHMDF POS=(8,1),ATTRB=(ASKIP,NORM),LENGTH=22,              X
                INITIAL='OTHERS WHO MAY CHARGE:'
AUTH1D  DFHMDF POS=(9,1),ATTRB=(UNPROT,NORM),LENGTH=32
        DFHMDF POS=(9,34),ATTRB=(PROT,NORM),LENGTH=1
AUTH2D  DFHMDF POS=(9,36),ATTRB=(UNPROT,NORM),LENGTH=32
        DFHMDF POS=(9,69),ATTRB=(PROT,NORM),LENGTH=1
AUTH3D  DFHMDF POS=(10,1),ATTRB=(UNPROT,NORM),LENGTH=32
        DFHMDF POS=(10,34),ATTRB=(PROT,NORM),LENGTH=1
AUTH4D  DFHMDF POS=(10,36),ATTRB=(UNPROT,NORM),LENGTH=32
        DFHMDF POS=(10,69),ATTRB=(PROT,NORM),LENGTH=1
        DFHMDF POS=(12,1),ATTRB=(ASKIP,NORM),LENGTH=17,             X
                INITIAL='NO. CARDS ISSUED:'
Col     Col    Col                                                 Col
1       9      16                                                  72
CARDSD  DFHMDF POS=(12,19),ATTRB=(NUM,NORM),LENGTH=1
        DFHMDF POS=(12,21),ATTRB=(ASKIP,NORM),LENGTH=16,            X
                INITIAL='    DATE ISSUED:'
IMOD    DFHMDF POS=(12,38),ATTRB=(UNPROT,NORM),LENGTH=2  7
IDAYD   DFHMDF POS=(12,41),ATTRB=(UNPROT,NORM),LENGTH=2
IYRD    DFHMDF POS=(12,44),ATTRB=(UNPROT,NORM),LENGTH=2
        DFHMDF POS=(12,47),ATTRB=(ASKIP,NORM),LENGTH=12,            X
                INITIAL='     REASON:'
RSND    DFHMDF POS=(12,60),ATTRB=(UNPROT,NORM),LENGTH=1
        DFHMDF POS=(12,62),ATTRB=(ASKIP,NORM),LENGTH=1
        DFHMDF POS=(13,1),ATTRB=(ASKIP,NORM),LENGTH=10,             X
                INITIAL='CARD CODE:'
CCODED  DFHMDF POS=(13,12),ATTRB=(UNPROT,NORM),LENGTH=1
        DFHMDF POS=(13,14),ATTRB=(ASKIP,NORM),LENGTH=1
        DFHMDF POS=(13,25),ATTRB=(ASKIP,NORM),LENGTH=12,            X
                INITIAL='APPROVED BY:'
APPRD   DFHMDF POS=(13,38),ATTRB=(UNPROT,NORM),LENGTH=3
        DFHMDF POS=(13,42),ATTRB=(ASKIP,NORM),LENGTH=1
        DFHMDF POS=(13,52),ATTRB=(ASKIP,NORM),LENGTH=14,            X
                INITIAL='SPECIAL CODES:'
SCODE1D DFHMDF POS=(13,67),ATTRB=(UNPROT,NORM),LENGTH=1
SCODE2D DFHMDF POS=(13,69),ATTRB=(UNPROT,NORM),LENGTH=1
SCODE3D DFHMDF POS=(13,71),ATTRB=(UNPROT,NORM),LENGTH=1
        DFHMDF POS=(13,73),ATTRB=(ASKIP,NORM),LENGTH=1
STATTLD DFHMDF POS=(15,1),ATTRB=(ASKIP,NORM),LENGTH=15,             X
                INITIAL='ACCOUNT STATUS:'
STATD   DFHMDF POS=(15,17),ATTRB=(ASKIP,NORM),LENGTH=2
LIMTTLD DFHMDF POS=(15,20),ATTRB=(ASKIP,NORM),LENGTH=18,            X
                INITIAL='     CHARGE LIMIT:'
LIMITD  DFHMDF POS=(15,39),ATTRB=(ASKIP,NORM),LENGTH=8
HISTTLD DFHMDF POS=(17,1),ATTRB=(ASKIP,NORM),LENGTH=71,  8        X
                INITIAL='HISTORY:   BALANCE      BILLED       AMOUNT X
                    PAID          AMOUNT'                 9
HIST1D  DFHMDF POS=(18,11),ATTRB=(ASKIP,NORM),LENGTH=61  10
```

```
¦ HIST2D  DFHMDF POS=(19,11),ATTRB=(ASKIP,NORM),LENGTH=61                ¦
¦ HIST3D  DFHMDF POS=(20,11),ATTRB=(ASKIP,NORM),LENGTH=61                ¦
¦ MSGD    DFHMDF POS=(22,1),ATTRB=(ASKIP,BRT),LENGTH=60                  ¦
¦ VFYD    DFHMDF POS=(22,62),ATTRB=(ASKIP,NORM),LENGTH=1   11            ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 34. The account detail map definition

Subtopics

3.2.3.1.1 Notes on the detail map

3.2.3.1.1 Notes on the detail map

The N comments are *not* part of the code.

1 We've put a suffix on each of the labels to tell us which map the field is from; in this map the suffix is **D**, for detail. We did the same thing in the menu (**M**)--see Figure 31 in topic 3.2.2.1--and will do so in subsequent maps. Thus, the account number is **ACCTM** in the menu map and **ACCTD** in the detail map. This is simply for clarity and to avoid having to use COBOL qualifiers to distinguish between fields with the same name. We could just as easily have used a prefix instead of a suffix; neither is a BMS requirement.

2 In this field, we've specified the value for the most common situation: record displays. This initial value is not a constant, as it is in the fields without labels, but a default. The field will be set to a different value by the program for adds, modifies, and other uses of the screen.

Notice that it has a label, so that the program has access to it.

3 Where you have a data field following a constant field, and there are three or fewer space characters between the end of the constant and the attribute byte for the data field, it's a good idea to fill out the constant to meet the data field. This allows BMS to omit the address for the data field (since it is adjacent to the previous field).

You cut down the length of the transmitted datastream this way, although the definition works perfectly well without this nicety, of course.

This field could have a length of 8 and an initial value of **SURNAME:** the appearance of the map would be exactly the same.

4 This is normally the first field into which the user is to enter data, and so we've specified that the cursor should be here. This is a default specification; the program can and often will override it.

5 We've defined this stopper field as protected, rather than autoskip, because the preceding field is of variable length.

As we said earlier, this choice warns users who try to key too many characters for the field, because the keyboard locks as soon as they get to the protected field.

6 We've combined a stopper field with the label field following it here. Since any field that begins right after the input field can act as a stopper, we've simply lengthened the field following the input field (the label **MI** here) with *leading* spaces, to combine our stopper and label in one field.

Generally, if there are fewer than four characters between the end of one field and the start of another, and they are constant (unlabeled) fields with the same attributes, it's better to combine them. The resulting data stream is shorter, and there's less BMS code.

7 You don't need a stopper field for an input field if another input field follows immediately.

8 These title fields are supposed to appear on all the displays except the skeleton screen for adding new records. It's easiest to put them in the map, therefore, and simply knock them out (not allow them to appear) for an add operation.

We'll do this by setting the attribute byte to "nondisplay" in that one case. To enable the program to access the attribute bytes, we have to put

labels on the fields.

9 This field is an example of a long **INITIAL** value parameter, for which two lines are required.

10 These are composite fields. If we wanted, we could define each of the "history" lines on the bottom of the screen as seven different fields, one for each item of data, and we'd do this if data was being entered on this line. However, since it's only being displayed, we don't need the attribute and cursor control that separate fields would provide.

It's easier to treat these seven items as a composite field, formatting the line within the program. If you look back at Figure 31 in topic 3.2.2.1, you'll notice that we used the same technique for the name search output in the menu map.

11 This field is used only for deletions, so the default value for the attribute byte will be autoskip. That way the user won't even be aware of the field when using the map for other transactions. For deletions, the program will change the attribute byte to be unprotected.

3.2.3.2 Defining the error map

Next is the error map, to produce the screen shown in Figure 28 in topic 2.10.5. Figure 35 shows the error screen map, with row and column numbers added.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦             1         2         3         4         5         6        ¦
¦    12345678901234567890123456789012345678901234567890123456789012345678  ¦
¦  1                                                                     ¦
¦  2                                                                     ¦
¦  3                                                                     ¦
¦  4 ACCOUNT FILE: ERROR REPORT                                         ¦
¦  5                                                                     ¦
¦  6 TRANSACTION  ____  HAS FAILED IN PROGRAM  ______  BECAUSE OF        ¦
¦  7                                                                     ¦
¦  8 ____________________________________________________________        ¦
¦  9                                                                     ¦
¦ 10 COMMAND  __________         RESP  __________                       ¦
¦ 11                                                                    ¦
¦ 12 ________________________                                           ¦
¦ 13                                                                    ¦
¦ 14 PLEASE ASK YOUR SUPERVISOR TO CONVEY THIS INFORMATION TO THE       ¦
¦ 15 OPERATIONS STAFF.                                                  ¦
¦ 16                                                                    ¦
¦ 17 THEN PRESS "CLEAR".  THIS TERMINAL IS NO LONGER UNDER CONTROL OF   ¦
¦ 18 THE "ACCT" APPLICATION.                                            ¦
¦                                                                       ¦
+------------------------------------------------------------------------+
```

Figure 35. The error screen map

When CICS abends our transaction, the **ABEND** message appears towards the foot of this screen. It normally appears at the current cursor position, although your system programmer can override this. (If you examine the **ACCT** behavior under EDF in "Execution diagnostic facility (EDF)" in topic 5.1.3.1, you'll see an example of this.)

Figure 36 shows the macro definition we need to produce this error screen.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ Col     Col    Col                                            Col     ¦
¦ 1       9      16                                             72      ¦
¦ *       ERROR MAP.                                                     ¦
¦ ACCTERR DFHMDI SIZE=(24,80),CTRL=FREEKB                                ¦
¦         DFHMDF POS=(4,1),ATTRB=(ASKIP,NORM),LENGTH=26,         X       ¦
¦                INITIAL='ACCOUNT FILE: ERROR REPORT'                    ¦
¦         DFHMDF POS=(6,1),ATTRB=(ASKIP,NORM),LENGTH=12,         X       ¦
¦                INITIAL='TRANSACTION '                                  ¦
¦ TRANE   DFHMDF POS=(6,14),ATTRB=(ASKIP,BRT),LENGTH=4                   ¦
¦         DFHMDF POS=(6,19),ATTRB=(ASKIP,NORM),LENGTH=23,        X       ¦
¦                INITIAL=' HAS FAILED IN PROGRAM '                       ¦
¦ PGME    DFHMDF POS=(6,43),ATTRB=(ASKIP,BRT),LENGTH=8                   ¦
¦         DFHMDF POS=(6,52),ATTRB=(ASKIP,NORM),LENGTH=11,        X       ¦
¦                INITIAL=' BECAUSE OF'                                   ¦
¦ RSNE    DFHMDF POS=(8,1),ATTRB=(ASKIP,BRT),LENGTH=60                   ¦
¦         DFHMDF POS=(10,1),ATTRB=(ASKIP,NORM),LENGTH=8,         X       ¦
¦                INITIAL='COMMAND '                                      ¦
¦ CMDE    DFHMDF POS=(10,10),ATTRB=(ASKIP,BRT),LENGTH=20                 ¦
¦         DFHMDF POS=(10,31),ATTRB=(ASKIP,NORM),LENGTH=5,        X       ¦
¦                INITIAL='RESP '                                         ¦
```

```
¦ RESPE   DFHMDF POS=(10,37),ATTRB=(ASKIP,BRT),LENGTH=12                ¦
¦ FILEE   DFHMDF POS=(12,1),ATTRB=(ASKIP,BRT),LENGTH=22                 ¦
¦         DFHMDF POS=(14,1),ATTRB=(ASKIP,NORM),LENGTH=60,        X       ¦
¦                INITIAL='PLEASE ASK YOUR SUPERVISOR TO CONVEY           ¦
¦                THIS INFORMATION TO THE'                         X       ¦
¦         DFHMDF POS=(15,1),ATTRB=(ASKIP,NORM),LENGTH=17,        X       ¦
¦                INITIAL='OPERATIONS STAFF.'                              ¦
¦         DFHMDF POS=(17,1),ATTRB=(ASKIP,NORM),LENGTH=64,        X       ¦
¦                INITIAL='THEN PRESS "CLEAR".  THIS TERMINAL IS           ¦
¦                NO LONGER UNDER CONTROL OF'                      X       ¦
¦         DFHMDF POS=(18,1),ATTRB=(ASKIP,NORM),LENGTH=23,        X       ¦
¦                INITIAL='THE "ACCT" APPLICATION.'                        ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 36. The error screen map definition

3.2.3.3 Defining the message map

Finally, there's the message map, which has just a single field, in which to send a message to the user.

We need this map in program **ACCT03,** to confirm (at the input terminal) that a request to print the log of changes to the account file has been processed. In other words, it's for the response to an **ACLG** (log print) transaction entered by the supervisor. Figure 37 shows the definition:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ Col     Col    Col                                              Col     ¦
¦ 1       9      16                                               72      ¦
¦ *       MESSAGE MAP.                                                    ¦
¦ ACCTMSG DFHMDI SIZE=(24,80),CTRL=FREEKB                                 ¦
¦ MSG     DFHMDF POS=(1,1),ATTRB=(ASKIP,NORM),LENGTH=79                   ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 37. The message map definition

After we've executed:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ MOVE 'PRINTING OF LOG HAS BEEN SCHEDULED' TO MSGO.                      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

we send this message back to the requesting terminal, confirming that the requested work has been scheduled. Unlike all the other types of requests that make up this application, a request to print the log isn't entered through the menu screen. So it isn't appropriate to use the message area of the menu screen, which is why we need our separate message map to send this message. As you can see, **ACCTMSG** is simply a one-line map consisting of an area for a message.

3.2.3.4 The map set

If we put together the four maps that we've now defined (the menu map, detail map, error map, and message map), Figure 38 shows the result.

```
ACCTSET DFHMSD TYPE=MAP,MODE=INOUT,LANG=COBOL,
               STORAGE=AUTO,TIOAPFX=YES
*       MENU MAP.
ACCTMNU DFHMDI SIZE=(24,80),CTRL=(PRINT,FREEKB)
        DFHMDF ... (all macros for the menu map)
*
*       DETAIL MAP.
ACCTDTL DFHMDI SIZE=(24,80),CTRL=(FREEKB,PRINT)
        DFHMDF ... (all macros for the detail map)
*
*       ERROR MAP.
ACCTERR DFHMDI SIZE=(24,80),CTRL=FREEKB
        DFHMDF ... (all macros for the error map)
*
*       MESSAGE MAP.
ACCTMSG DFHMDI SIZE=(24,80),CTRL=FREEKB
MSG     DFHMDF POS=(1,1),ATTRB=(ASKIP,NORM),LENGTH=79
        DFHMSD TYPE=FINAL
        END
```

Figure 38. All four maps

3.2.4 Summary

Item	**Comments**
fldname	Use only on fields your program will access
mapname	1-7 characters, starting alpha, no special characters
setname	As mapname. Co-ordinate setname with the entry in the list of installed program definitions
POS	Gives position of attribute byte, not first data character
LENGTH	Does not include attribute byte.

3.2.5 Optional exercise

For those of you with a terminal, the CICS COBOL sample programs, and a running CICS system.

You can use the CICS command interpreter CECI (not covered in this Primer) to see what a map looks like on the screen:

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ CECI SEND MAP ('DFH$AGA') MAPONLY                                     ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

This will display the operator instructions menu for the assembler language version of the File A sample that's supplied with CICS as part of its own sample transaction set. Don't worry about trying to decipher the map now, though--wait until you've read the next topic.

Alternatively, you can get a rough idea of how the **ACCT** example application behaves by skimming through the EDF session shown in "Execution diagnostic facility (EDF)" in topic 5.1.3.1.

3.3 Using BMS: more detail

Subtopics

3.3.1 Symbolic description maps (DSECT structures)

As we said earlier, assembling the macros with **TYPE=MAP** specified in the **DFHMSD** macro produces the physical map that CICS uses at execution time. After you've done this assembly, you do it all over again, this time specifying **TYPE=DSECT.** This second assembly produces the **symbolic description map**, a COBOL structure that you copy into your program. It's stored in the copybook library specified in the JCL, and its name in that library is the map set name specified in the **DFHMSD** macro.

This structure is a set of data definitions for all the display fields on the screen, plus information about those fields. It allows your program to refer to these display data fields by name and to manipulate the way in which they are displayed, without worrying about their size or position on the screen.

Subtopics
3.3.1.1 Copying the map DSECT into a program
3.3.1.2 The generated subfields

3.3.1.1 Copying the map DSECT into a program

To copy the **DSECT** structures for the maps in a map set into a program, you write a **COPY** statement like this:

```
                  COPY setname.
```

Here, "setname" is the name of the map set. This **COPY** statement usually appears in **WORKING-STORAGE**, although later you may find reasons to put it in the **LINKAGE SECTION.** We'll cover only the **WORKING-STORAGE** situation. To get the symbolic descriptions for our maps in a program, we'll write:

```
                 COPY ACCTSET.
```

Figure 39 shows you the first few lines of what is copied into your program as a result of this **COPY** statement. The part shown is generated by the first map in the set, the menu map. It's followed by similar structures for the other maps. We've not shown all of them here because they're very long and very similar in form. They're all in "The result of the SYSPARM=DSECT assembly" in topic A.2.1.

```
        01  ACCTMNUI.
            02  FILLER PIC X(12).
            02  SNAMEML    COMP  PIC  S9(4).
            02  SNAMEMF    PICTURE X.
            02  FILLER REDEFINES SNAMEMF.
              03 SNAMEMA    PICTURE X.
            02  SNAMEMI  PIC X(12).
            02  FNAMEML    COMP  PIC  S9(4).
            02  FNAMEMF    PICTURE X.
            02  FILLER REDEFINES FNAMEMF.
              03 FNAMEMA    PICTURE X.
            02  FNAMEMI  PIC X(7).
            02  REQML    COMP  PIC  S9(4).
            02  REQMF    PICTURE X.
            02  FILLER REDEFINES REQMF.
              03 REQMA    PICTURE X.
            02  REQMI  PIC X(1).
            02  ACCTML    COMP  PIC  S9(4).
            02  ACCTMF    PICTURE X.
            02  FILLER REDEFINES ACCTMF.
              03 ACCTMA    PICTURE X.
            02  ACCTMI  PIC X(5).
            02  PRTRML    COMP  PIC  S9(4).
            02  PRTRMF    PICTURE X.
            02  FILLER REDEFINES PRTRMF.
        .
        .
        .
```

Figure 39. Copying the menu map into your program

Because we asked for a map to be used for both input and output (by coding **MODE=INOUT** in the **DFHMSD** macro), the resulting structure has two parts.

The first part corresponds to the input screen, and is always labelled (at the **01** level) with the map name, suffixed by the letter **I** (for "input"). The second part corresponds to the output screen, and is labeled with the map name followed by the letter **O.** The output map always redefines the input map. If we'd specified **MODE=IN,** only the input part would have been generated, and similarly, **MODE=OUT** would've produced only the output part.

3.3.1.2 The generated subfields

We gave names to eight field definitions in the menu map: **SNAMEM, FNAMEM, REQM, ACCTM, PRTRM, SUMTTLM, SUMLNM,** and **MSGM.** (One of these, **SUMLNM,** has an **OCCURS** clause causing it to define six different fields, but we'll get to that shortly.) Notice that for each of these map fields, five data subfields are generated. Each subfield has a name consisting of the field name in the map and a one-letter suffix. (We're using "subfields" to distinguish them from the single "map" field from which they originate.)

We can explain the contents of the subfields better by using a specific set of data. Suppose someone has filled in the menu screen, as shown in Figure 40:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ ACCOUNT FILE: MENU                                                      ¦
¦    TO SEARCH BY NAME, ENTER:                            ONLY SURNAME    ¦
¦                                                         REQUIRED. EITHER ¦
¦       SURNAME: SMITH          FIRST NAME: J             MAY BE PARTIAL.  ¦
¦    FOR INDIVIDUAL RECORDS, ENTER:                                       ¦
¦                                                         PRINTER REQUIRED ¦
¦       REQUEST TYPE:     ACCOUNT:         PRINTER:       ONLY FOR PRINT   ¦
¦                                                         REQUESTS         ¦
¦       REQUEST TYPES:  D = DISPLAY  A = ADD    X = DELETE                ¦
¦                       P = PRINT    M = MODIFY                           ¦
¦    THEN PRESS "ENTER"             -OR-   PRESS "CLEAR" TO EXIT          ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 40. The menu screen at work

Ultimately, BMS puts the user's data into our program's **WORKING-STORAGE**, along with some control information. Look at Figure 39 as you study what follows.

The first 12 characters in the **DSECT** (**FILLER**) are there because we said **TIOAPFX=YES** when we defined the map set. They're reserved for CICS control information, and are of no concern to the application program.

The first suffix is **L,** which stands for "length." **SNAMEML** is the number of characters that the user keyed into the **SNAMEM** field (or, if the program put some data there and turned on the modified data tag, the length of that data). In the example shown above, **SNAMEML** will be 5 (the length of "**SMITH**"), **FNAMEML** will be 1, and **REQML, ACCTML** and all the others will be zero.

The second suffix is **F** (meaning "flag"), and this subfield tells you whether or not the user changed the corresponding field on the screen *by erasing it* (setting it to nulls with the **ERASE EOF** key). Such a subfield of course always has a length (**L** subfield) of zero; the flag allows you to tell whether it was written on the screen that way or whether the user erased something that was there. A flag value of X'80' shows that the field was changed by erasing; otherwise the flag value is X'00' (nulls, or **LOW-VALUE** in COBOL). In the filled-in menu screen, all the flag fields will contain X'00', because there was no field sent which could be erased.

Pressing **ERASE EOF** causes the flag to be set *even if the field was empty to start with*, and whether or not you type in some data before changing your mind and erasing the field.

The flag value becomes important in connection with modifications, as we'll see later. The other suffix is **I,** for "input." This is the actual

content of the field on the screen, provided that the modified data tag is on for the field. The tag will be on if the user changed the field or if it was sent with the **FSET** attribute specified. If the tag isn't on, the program doesn't read what's on the screen, and the **I** subfield will contain nulls.

The **I** subfield is defined as a character string of the length you specify in the map. Because the **SNAMEM** field in the menu map has a length of 12, the **SNAMEMI** subfield is given a **PICTURE** value of **X(12)** in the symbolic map description. (BMS provides a parameter called **PICIN** that you can use in the **DFHMDF** macro for a field that changes the picture generated, however, if you wish to do so.)

If the user doesn't fill in the whole field, as in the case of the two name fields here, BMS pads out the field to its maximum length. If a field has the **NUM** attribute, it's filled on the left with leading (decimal) zeros; otherwise it's filled on the right with spaces. In this screen, then, **SNAMEMI** would equal "**SMITH** ", and **FNAMEMI** would be **J**--the unkeyed part of each field being filled with spaces.

The remaining two data fields for a map field concern output rather than input, even though one of them appears in the "input" part of an **INOUT** map. This is the one suffixed by **A** (for "attribute"). When you're sending a map, and you want a field to have a different set of attributes than you specified in the map, you can override the map specification by setting this field. For example, suppose the user had typed **SM1TH** instead of **SMITH**. We'd want to bounce the menu screen straight back to the user with the surname field highlighted, to show our displeasure at finding the numeric character **1** there. To do so, we'd simply need to move the character that represented the attributes we wanted to **SNAMEMA.**

The character we need to do this is the one actually used in the 3270 output data stream. These character representations are quite hard to remember, so CICS provides you with a library member containing most of the useful combinations, defined with meaningful names. To get access to it, you simply put the statement:

```
                COPY DFHBMSCA
```

in your **WORKING-STORAGE.** This generates a list of definitions like the one shown in Figure 41:

```
 01  DFHBMSCA.
     02  DFHBMPEM    PICTURE X   VALUE IS ' '.
     02  DFHBMPNL    PICTURE X   VALUE IS ' '.
     02  DFHBMASK    PICTURE X   VALUE IS '0'.
     02  DFHBMUNP    PICTURE X   VALUE IS ' '.
     02  DFHBMUNN    PICTURE X   VALUE IS '&'.
     02  DFHBMPRO    PICTURE X   VALUE IS '-'.
     02  DFHBMBRY    PICTURE X   VALUE IS 'H'.
     02  DFHBMDAR    PICTURE X   VALUE IS '<'.
     02  DFHBMFSE    PICTURE X   VALUE IS 'A'.
     02  DFHBMPRF    PICTURE X   VALUE IS '/'.
         . . .
```

Figure 41. Attribute values for the IBM 3270 data stream

You'll find a complete list of these definitions in the CICS/ESA

Application Programming Reference. The values which appear to be spaces are not; they are bit combinations that do not represent a printed character, although they are all valid EBCDIC characters. The definitions generated (that apply to this Primer) are shown in Figure 42.

Variable	**Protection**	**Intensity**	**Modified Data Tag**
DFHBMUNP	Unprotected	Normal	Off
DFHBMUNN	Numeric	Normal	Off
DFHBMPRO	Protected	Normal	Off
DFHBMASK	Autoskip	Normal	Off
DFHBMBRY	Unprotected	Bright	Off
DFHPROTI	Protected	Bright	Off
DFHBMASB	Autoskip	Bright	Off
DFHBMDAR	Unprotected	Non-display	Off
DFHPROTN	Protected	Non-display	Off
DFHBMFSE	Unprotected	Normal	On
DFHUNNUM	Numeric	Normal	On
DFHBMPRF	Protected	Normal	On
DFHBMASF	Autoskip	Normal	On
DFHUNIMD	Unprotected	Bright	On
DFHUNINT	Numeric	Bright	On
DFHUNNOD	Unprotected	Non-display	On
DFHUNNON	Numeric	Non-display	On

Figure 42. Attribute values used in the Primer

Referring back to our example, to highlight the surname we:

```
MOVE DFHBMBRY TO SNAMEMA
```

before sending the map back to the terminal. We're using **DFHBMBRY,** rather than one of the other "bright" variables because, unlike some other high-intensity values, **DFHBMBRY** leaves the field unprotected, so the user will be able to rekey the name properly. It also sets the modified data tag off (a choice we'll discuss later).

The last of the five data subfields for a map field is named with a suffix of **O** (for "output"). It's the data that you want displayed in the map field when you send it. Like the input subfield, the output subfield defaults to a character string of the length specified in the map; you can specify some other **PICTURE** by using the **PICOUT** parameter in the **DFHMDF** macro that defines the field. For programming interface information on **PICOUT** and **PICIN**, see the section on the **DFHMDF** macro in the CICS/ESA Application Programming Reference.

Subtopics
3.3.1.2.1 Fields defined with the OCCURS= parameter
3.3.1.2.2 Some things to keep in mind about these DSECTs

3.3.1.2.1 Fields defined with the OCCURS= parameter

The only field on the screen that has generated a slightly different structure from what we've just described is the **SUMLNM** field, and this is because we've said it **OCCURS** six times.

Have another look at the **DSECT**. This time, you'll need to look at the full version, in "The result of the SYSPARM=DSECT assembly" in topic A.2.1.

For the **SUMLNM** field there's another level to the COBOL structure, a group named **SUMLNMD,** with an **OCCURS** value of 6. This group contains the **SUMLNML, SUMLNMF,** and **SUMLNMI** fields, which represent the length, flag value, and input for **SUMLNM,** just as you'd expect. The attribute field appears in the output section, where an extra group level is also introduced. This one's called **DFHMS1** (an arbitrarily generated name); it, too, **OCCURS** six times and contains the **SUMLNMA** and **SUMLNMO** fields. So you refer to the attribute value of the fourth occurrence of this field as **SUMLNMA(4)**, the input for the second occurrence as **SUMLNMI(2)**, and so on.

3.3.1.2.2 Some things to keep in mind about these DSECTs

Because of the way the input and output parts of the map structur overlay each other, the **-I** and the **-O** subfields for a given map field always redefine each other. That is, **SNAMEMI** and **SNAMEMO** occupy the same storage, **FNAMEMI** and **FNAMEMO** do also, and so on. This turns out to be convenient in coding.

The attribute and flag subfields occupy the same space **REQMF** overlays **REQMA, ACCTMF** overlays **ACCTMA,** and so on). You don't have to worry about removing these flags when you're sending output, however. Since the two input flag values (X'80' and X'00') don't represent acceptable output attribute byte values, BMS can distinguish on output between a leftover flag and a new attribute.

When you write a map, you don't have to put anything in the lengt field. BMS knows how long the field is from the information in the physical map. The only time you use the length field for an output field is to set the cursor position, a matter we'll explain shortly.

3.3.2 Sending a map to a terminal

Now that we've defined our maps, we can think about writing them to the terminal.

The terminal to which we'll write, of course, is the one that sent the input and thereby invoked the transaction. This is the only terminal to which a transaction can write directly, as mentioned in "Transactions and terminals" in topic 2.9.3.

Subtopics
3.3.2.1 The SEND MAP command
3.3.2.2 Using SEND MAP in the ACCT example

3.3.2.1 The SEND MAP command

The **SEND MAP** command writes formatted output to a terminal. It looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS SEND MAP(mapname) MAPSET(setname)                           ¦
¦       option option ... END-EXEC.                                       ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

mapname
is the name of the map you want to send. It's required. Put it in quotes if it's a literal.

setname
is the name of the map set that contains the mapname. Put the name in quotes if it's a literal. The map set name is needed unless it's the same as the map name. Code it for documentation purposes, anyway.

Note: It's inadvisable to use the same name for the map and the map set. (If you generate the map set with the suffix for **ALL** of 3270--the default--the map suffix used is a blank. This means the map name and map set name are identical, and causes the subsequent assembly to fail because two labels in the code are the same.)

option
There are a number of options that you can specify; they affect what's sent and how it is sent. Except where noted, you can use any combination of them. The possibilities are:

MAPONLY
means that no data from your program is to be merged into the map; only the information in the map is transmitted. In our example application, we'll use this option when we send the menu map the first time, because we'll have no information to put into it.

DATAONLY
is the logical opposite of **MAPONLY.** You use it to modify the variable data in a display that's already been created. Only the data from your program is sent to the screen. The constants in the map aren't sent; so you can use this option only *after* you've sent the same map without using the **DATAONLY** option. We'll see an example when we send the results of a name search to the terminal in program **ACCT01.**

ERASE
causes the entire screen to be erased before what you're sending is shown.

ERASEAUP
(erase all unprotected fields), in contrast to **ERASE,** causes just the unprotected fields on the screen (those with either the **UNPROT** or **NUM** attribute) to be erased before your output is placed on the screen. It's most often used in preparing to read new data from a map that's already on the screen. Don't use it at the same time as **ERASE**; **ERASE** makes **ERASEAUP** meaningless.

FRSET
(flag reset) turns off the modified data tag in the attribute bytes for all the fields on the screen before what you're sending is placed there. (Once set on, whether by the user or the program, a modified data tag stays on until turned off explicitly, even over several transmissions of the screen. It can be turned

off by the program sending a new attribute byte, an **FRSET** option, or an **ERASE,** or an **ERASEAUP,** or by the user pressing the **CLEAR** key.) Like **ERASEAUP,** the **FRSET** option is most often used in preparing to read new data from a map already on the screen. It can also reduce the amount of data re-sent on an error cycle, as we'll explain in coding our example.

CURSOR

can be used in two ways to position the cursor. If you specify a value after **CURSOR,** it's the relative position on the screen where the cursor is to be put. Use a single number, such as **CURSOR(81)** for line 2, column 2 (counting starts at zero and goes across the lines, which on an IBM 3270-system display Model 2 are 80 characters wide). Why column 2? Because the attribute byte goes in column 1, and we want the cursor to appear under the first character of data.

Some people prefer to put the attribute at the end of the previous line (for example, **POS=(1,80)**) to let the data in the field start in screen column 1.

Alternatively, you can specify **CURSOR** without a value, and use the length subfields in the output map to show which field is to get the cursor. See "Positioning the cursor" in topic 3.3.3. In general, we recommend you to position the cursor in this second manner, rather than the first, so that changes in the map layout don't lead to changes in the program. Both kinds of **CURSOR** specification override the cursor placement specified in the map.

ALARM

means the same thing in the **SEND** command as it does in the **DFHMSD** and **DFHMDI** macros for the map: it causes the audible alarm to be sounded. The alarm will sound if you specify **ALARM** in either the map definition or the **SEND** command.

FREEKB

likewise means the same thing as it does in the map definition: the keyboard is unlocked if you specify **FREEKB** in either the map or the **SEND** command.

PRINT

allows the output of a **SEND** command to be printed on a printer, just as it does in the map definition. It is in force if specified in either the map or the command.

FORMFEED

causes the printer to restore the paper to the top of the next page before the output is printed. This specification has no effect on maps sent to a display, to printers without the features which allow sensing the top of the form, or to printers for which the "formfeed" feature is not specified in the CICS Terminal Control Table.

3.3.2.2 Using SEND MAP in the ACCT example

The first time we need to send a map to a terminal occurs in program **ACCT00,** where we display the menu screen. The command we need is:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET') MAPONLY              ¦
¦        ERASE FREEKB END-EXEC.                                          ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

This is a very simple situation. Because we don't have any variable data to put in the map, we can use the **MAPONLY** option, and we don't have to worry about preparing variable data for merging with the physical map.

If we were sending some data to the screen with the map, we could not use **MAPONLY,** and CICS would expect the data to be used for filling in the map to be in a structure whose name is the map name (as specified in the **MAP** option) suffixed with the letter **O.** So, when we issue the command:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET')                      ¦
¦        END-EXEC.                                                       ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

CICS expects the data for the map to be in a structure within the program (of exactly the sort generated by the **DSECT** assembly) named **ACCTMNUO.** This structure is usually in your **WORKING-STORAGE** Section, but it might be in a **LINKAGE** area instead. (There's an option on the **SEND MAP** command that lets you specify a data structure other than the one assumed by CICS. We won't cover it here, but you can find guidance on using it in the CICS/ESA Application Programming Guide. under "Sending Data to a Display.")

Let's look at the more common situation in which we're merging program data into the map. In program **ACCT01,** we're supposed to build a detail display map for one record and send it to the screen. Since the contents of the screen vary somewhat with the type of request, and we're using the same screen for all types, this will entail the following:

1. Putting the appropriate title on the map (add, modify, or whatever it happens to be).

2. Moving the data from the file record to the symbolic map (except for adds).

3. Adjusting the attribute bytes. The input fields must be protected in a display or delete operation; the "verify" field must be unprotected for deletes, and the titles at the bottom of the screen must be made nondisplay for adds.

4. Putting the appropriate user instructions (about what to do next) into the message area.

5. Putting the cursor in the right place.

Figure 43 shows how the necessary code might look.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ Col  Col                                                               ¦
¦ 7    12                                                                ¦
```

```
 BUILD-MAP.
     IF REQC = 'X' MOVE 'DELETION' TO TITLEDO,
         MOVE -1 TO VFYDL, MOVE DFHBMUNP TO VFYDA,
         MOVE 'ENTER "Y" TO CONFIRM OR "CLEAR" TO CANCEL'
             TO MSGDO,
     ELSE MOVE -1 TO SNAMEDL.
     IF REQC = 'A' MOVE 'NEW RECORD' TO TITLEDO,
         MOVE DFHPROTN TO STATTLDA, LIMTTLDA, HISTTLDA,
         MOVE ACCTC TO ACCTDI,
         MOVE 'FILL IN AND PRESS "ENTER," OR "CLEAR" TO CANCEL'
             TO MSGDO,
         GO TO SEND-DETAIL.
     IF REQC = 'M' MOVE 'RECORD CHANGE' TO TITLEDO,
         MOVE 'MAKE CHANGES AND "ENTER" OR "CLEAR" TO CANCEL'
             TO MSGDO,
     ELSE IF REQC = 'D',
             MOVE 'PRESS "CLEAR" OR "ENTER" WHEN FINISHED'
                 TO MSGDO.
     MOVE CORRESPONDING ACCTREC TO ACCTDTLO.
     MOVE CORRESPONDING PAY-HIST (1) TO PAY-LINE.
     MOVE PAY-LINE TO HIST1DO.
     MOVE CORRESPONDING PAY-HIST (2) TO PAY-LINE.
     MOVE PAY-LINE TO HIST2DO.
     MOVE CORRESPONDING PAY-HIST (3) TO PAY-LINE.
     MOVE PAY-LINE TO HIST3DO.
     IF REQC  = 'M' GO TO SEND-DETAIL,
     ELSE IF REQC = 'P' GO TO PRINT-PROC.
     MOVE DFHBMASK TO
         SNAMEDA, FNAMEDA, MIDA, TTLDA, TELDA, ADDR1DA,
         ADDR2DA, ADDR3DA, AUTH1DA, AUTH2DA, AUTH3DA,
         AUTH4DA, CARDSDA, IMODA, IDAYDA, IYRDA, RSNDA,
         CCODEDA, APPRDA, SCODE1DA, SCODE2DA, SCODE3DA.
*    SEND THE RECORD DETAIL MAP TO THE TERMINAL.
 SEND-DETAIL.
     EXEC CICS SEND MAP('ACCTDTL') MAPSET('ACCTSET') ERASE FREEKB
         CURSOR END-EXEC.
```

Figure 43. Building the detail display map

Here are some explanatory notes.

REQC (request code) was moved to a working-storage field earlier in the program. It holds the user's "request code."

What is happening in this code is as follows:

If the user request is to delete a record **IF REQC** = X):

1. The map title is changed from its default to **DELETION**.

2. The cursor is placed under the "verify" field **(MOVE -1 TO VFYDL)** by a technique we'll explain shortly.

3. The attribute byte for that field is changed from its map default of autoskip to unprotected.

4. Instructions for what to do next are put in the message area.

The cursor is placed under the surname field for all other types o user requests **(ELSE MOVE -1 to SNAMEDL)**.

If the request is for an addition

1. The title is made **NEW RECORD**.
2. The titles at the bottom of the screen are given a nondisplay attribute.
3. The account field (from the request input) is placed in the output map.
4. Instructions are put into the message area.

If the request is a modification, the title and the message area ar set appropriately.

If the request is a display, instructions for what to do after th display are put in the message area.

For all types of requests except adds, the display is built from th record on file (**MOVE CORRESPONDING ACCTREC ...** through ... **MOVE PAY-LINE TO HIST3DO**).

If the request is to print a record, control goes to code a **PRINT-PROC** that will do the special processing required to write to a terminal other than the input terminal.

If the request is to display or delete, the attribute bytes of all th data fields that can be entered or changed on an addition or a modification are changed to autoskip. This makes it clear to users that they cannot change these fields in the current transaction.

For all request types except printing, the map is sent to the inpu terminal.

We need to use a somewhat different type of **SEND MAP** command later in the same program, when we have to redisplay the input (menu) map because of some error, or to put a message on the screen. Because the map is already on the screen, it is unnecessary (and wasteful of line capacity) to send what is already there again. So we use the **DATAONLY** option, and we do not erase the screen:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET')                       ¦
¦       CURSOR DATAONLY FRSET ERASEAUP FREEKB END-EXEC.                   ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

We also specify **FRSET** in this command. This prevents fields that were entered during the previous terminal interaction, and not rekeyed, from being sent on the next transmission. That is, only fields that the user changes (probably because of an error) will be transmitted the next time the terminal sends. This reduces line transmission, but it requires the transaction to save the input from the previous execution for the next one. We've a bit more to say about how to use **FRSET** in the notes that accompany Line 163 of the program source code of **ACCT01**. You'll find these in "Program ACCT01: initial request processing" in topic 2.10.2.

3.3.3 Positioning the cursor

We said earlier how vital it is to put the cursor where the user will want to start entering data on the screen. One small piece of source code from you can save hundreds of users a couple of seconds each and every time they use your application.

In the first **SEND MAP** example, we relied on the cursor position specified in the map definition. This puts the cursor under the first data position of the surname field, which is where we want it. In the second and third examples, however, we don't necessarily want the cursor where the map definition puts it. In the second example, where we're using the detail map, we want to use the map default (the **SNAMED** field) for adds and modifies. For display operations, it doesn't much matter, since there are no fields into which the user may key. For deletes, however, the cursor should be under the verify (**VFY**) field. In the third example, we want the cursor under the first field where the user entered incorrect information.

As we said, there are two ways to override the position specified by the **IC** specification in the map definition:

1. You can specify a screen position, relative to line 1, column 1 (that is, position 0) in the **CURSOR** option on the **SEND MAP** command (the procedure we advised against earlier).

2. You can show that you want the cursor placed under a particular field by setting the associated length subfield to minus one (-1) and specifying **CURSOR** without a value in your **SEND MAP** command. This causes BMS to place the cursor under the first data position of the field with this length value. If several fields are flagged by this special length subfield value, the cursor is placed under the first one (as opposed to the last one with **ATTRB=IC**).

The second procedure is called **symbolic cursor positioning**, and is a very handy method of positioning the cursor for, say, correcting errors. As the program checks the input, it sets the length subfield to -1 for every field found to be in error. Then, when the map is redisplayed for corrections, BMS automatically puts the cursor under the first field that the user will have to correct.

To place the cursor under the verify field on a delete, therefore, all we have to do is:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   MOVE -1 TO VFYDL                                                      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

and specify **CURSOR** in our **SEND MAP** command.

3.3.4 Sending control information without data

In addition to the SEND MAP command, there is another terminal output command called SEND CONTROL. It allows you to send control information to the terminal without sending any data. That is, you can open the keyboard, erase all the unprotected fields, and so on, without sending a map.

Subtopics
3.3.4.1 The SEND CONTROL command

3.3.4.1 The SEND CONTROL command

The **SEND CONTROL** command looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS SEND CONTROL option option  ...  END-EXEC.                  ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The options you can use are the same as on a **SEND MAP** command: **ERASE, ERASEAUP, FRSET, ALARM, FREEKB, CURSOR, PRINT,** and **FORMFEED.**

There's an example of this command in program **ACCT01.** The terminal user has just cleared the screen (of the menu map) to indicate that he or she wants to exit from the control of the online account application. The program is supposed to open the keyboard before returning control to CICS.

Normally, you would do this when writing a message to the terminal. But since we're not doing that at this point, we must unlock the keyboard by an explicit command, instead. The command is:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS SEND CONTROL FREEKB END-EXEC.                               ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

If we didn't know the user had just cleared the screen, we'd probably want to add the **ERASE** option to the command above, so that the user would be all ready to start a new transaction.

3.3.5 Receiving input from a terminal

Subtopics

3.3.5.1 The RECEIVE MAP command

3.3.5.1 The RECEIVE MAP command

When you want to receive input from a terminal, you use the **RECEIVE MAP** command, which looks like this:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   EXEC CICS RECEIVE MAP(mapname) MAPSET(setname)                       ¦
¦             END-EXEC.                                                  ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

The **MAP** and **MAPSET** parameters have exactly the same meaning as for the **SEND MAP** command. **MAP** is required and so is **MAPSET,** unless it is the same as the map name. Again, it does no harm to include it for documentation purposes.

We're showing you a form of the **RECEIVE MAP** command that does not specify where the input data is to be placed. This causes CICS to bring the data into a structure whose name is the map name suffixed with the letter **I,** which is assumed to be in either your **WORKING-STORAGE** or **LINKAGE** Section.

For example, program **ACCT02** requires that we receive the filled-in detail map. The command to do this:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦  EXEC CICS RECEIVE MAP('ACCTDTL') MAPSET('ACCTSET')                    ¦
¦             RESP(RESPONSE) END-EXEC.                                   ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

will bring the input data into a data area named **ACCTDTLI,** which is expected to have exactly the format produced by the **DSECT** for map **ACCTDTL** (We'll explain **RESP(RESPONSE)** in "Errors and exceptional conditions" in topic 3.8.)

As soon as the map is read in, we have access to all the data subfields associated with the map fields. For example, we can test whether the user made any entry in the request field of the menu map:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   IF REQML > 0, MOVE ...                                               ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Or we could examine the input in that field:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   IF REQMI = 'A' GO TO ...                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Note: Although it generally will not affect your program logic, you should be aware that the first time in a transaction that you use the **RECEIVE MAP** command, it has a slightly different effect from subsequent times. Since it is input from the terminal that causes a transaction to get started in the first place, CICS has always read the first input by the time the transaction starts to execute. Therefore, on this first **RECEIVE MAP** command, CICS simply arranges the input it already has into the format dictated by your map, and puts the results in a place accessible to your program.

On subsequent **RECEIVE MAP** commands in the same task, CICS actually waits for and reads input from the terminal. These subsequent **RECEIVE MAPs** are what make a task conversational. By contrast, a pseudoconversational task executes at most one **RECEIVE MAP** command.

3.3.6 Finding out what key the operator pressed

There is another technique you may wish to use for processing input from a terminal. As we pointed out in "3270 input data stream" in topic 2.2.4, the 3270 input stream contains an indication of what **attention** key caused the input to be transmitted (**ENTER, CLEAR,** or one of the **PA** or **PF** keys).

You can use the **EIBAID** field to cause your program to change the flow of control in your program based on which of these attention keys was used. (**AID** stands for attention identifier.)

Subtopics
3.3.6.1 The EXEC Interface Block (EIB)

3.3.6.1 The EXEC Interface Block (EIB)

Before we explain how to find out what key was used to send the input, we need to introduce one CICS control block. This is the **EIB,** which stands for EXEC Interface Block, and it is the only one that you need to know anything about for the type of applications described in this Primer.

You can write programs without using even this one, but it contains information that can be very useful and is worth knowing about.

There is one **EIB** for each task, and it exists for the duration of the task. Every program that executes as part of the task has access to the same **EIB.** You can address the fields in it directly in your COBOL program, without any preliminaries. You should only read these fields, however, not try to modify them. All of the **EIB** fields are discussed in detail in the CICS/ESA Application Programming Reference manual, but the ones that apply to the commands and options in this Primer are:

EIBAID
The attention identifier (AID), which tells you which keyboard key was used to transmit the last input. This field is one byte long ("PIC X(1)"). It is encoded as shown in "AID byte definitions" in topic 3.3.6.1.1.

EIBCALEN
The length of the communication area (COMMAREA) that has been passed *to* this program, either from a program that invoked it using a CICS command (LINK or XCTL--see "Commands for passing program control" in topic 3.6.2), or from a previous transaction in a pseudoconversational sequence. It is in halfword binary form (**PIC S9(4) COMP**). See "Program control" in topic 3.6 and "Saving data and communicating between transactions" in topic 3.5 for more information on COMMAREA.

EIBCPOSN
The position of the cursor at the time of the last input command, for 3270-like devices only. This position is expressed as a single number relative to position zero on the screen (row 1, column 1), in the same way that you specify the CURSOR parameter on a SEND MAP command. It's also in halfword binary form ("PIC S9(4) COMP").

After a RECEIVE MAP command, your program can find the inbound cursor position by inspecting the value held in EIBCPOSN.

EIBDATE
The date on which the current task started, in Julian form, with two leading zeros. The COBOL "PICTURE" for the field is "S9(7) COMP-3", and the format is: "00YYDDD+".

EIBDS
The name of the last file used in a file command (for example, read a record, write a record). This field is eight characters long ("PIC X(8)") and is the value in the "FILE" parameter of the most recent file command.

EIBFN
A code indicating the last command that was issued by the task, in "PIC X(2)" form. The first byte of this two-byte field indicates the type of command. File commands have a code of X'06', BMS commands are 18, and so on. The second byte tells which particular command: 0602 means READ, 0604 means WRITE, and so on. A full list of the codes appears in the CICS/ESA Application Programming Reference , and the subset that applies to the command and option combinations we use also appears in "Program ACCT04: error processing" in topic 2.10.5. The codes involved appear in the table HEX-LIST (line 27). in "Program ACCT04: error processing" in topic 2.10.5 of this Primer. The codes

involved appear in the table HEX-LIST (line 27).

EIBRCODE

The response code resulting from executing the last command. This is a six-byte field ("PIC X(6)"), but for the command types covered in this Primer, you need concern yourself only with the first byte. The HEX-LIST table we mentioned above also contains a list of all the codes that can result from our subset of commands and options. The CICS/ESA Application Programming Reference contains a full list of the possibilities.

EIBRESP

This contains a number corresponding to the condition that has been raised. **DFHRESP**, which you'll see in "Program ACCT04: error processing" in topic 2.10.5, contains the numbers and their meanings. There is a complete list of these numbers in the CICS/ESA Application Programming Reference.

EIBRESP2

This contains more detailed information that may help explain why the RESP condition has been raised. This field contains meaningful values (as decimal numbers) for specific commands. Values relating to general-usage programming interface commands such as INQUIRE, SET, and JES spooler commands are in the CICS/ESA System Programming Reference. Values relating to product-sensitive programming interface commands are in the in the CICS/ESA Application Programming Reference.

EIBRSRCE

The name of the resource used in the most recent command that used such a resource. For file commands, this value is the FILE parameter, so that EIBRSRCE has the same value as EIBDS after such a command. For temporary storage commands, it is the name of the queue (the QUEUE parameter), and for BMS commands it is the name of the terminal (the four-character name of the input terminal, or EIBTRMID in the context of this Primer). Eight characters are provided for this information ("PIC X(8)"), although some names, like those of terminals, fill only the first four positions.

EIBTASKN

The task number, as a seven-digit packed decimal number ("PIC S9(7) COMP-3"). CICS assigns a sequential number to each task it executes, and this number is used to identify entries for the task in the Trace Table (for further guidance, see the sections on dump and trace in the CICS/ESA Problem Determination Guide).

EIBTIME

The time at which the current task started, also in "PIC S9(7) COMP-3" form, with one leading zero: "0HHMMSS+".

EIBTRMID

The name of the terminal associated with the task (the input terminal, usually, or sometimes a printer, as in our AC03 and AC05 transaction types). This name is four characters long, and the COBOL "PICTURE" is "X(4)".

EIBTRNID

The transaction identifier of the current task, four characters long ("PIC X(4)").

Subtopics
3.3.6.1.1 AID byte definitions

3.3.6.1.1 AID byte definitions

Getting back to the attention identifier, we can also tell what key was used to send the input by looking at the EIBAID field, as noted above.

When a transaction is started, EIBAID is set according to the key used to send the input that caused the transaction to get started. It retains this value through the first RECEIVE command, which only formats the input already read, until after a subsequent RECEIVE, at which time it is set to the value used to send that input from the terminal.

EIBAID is one byte long and holds the actual attention identifier value used in the 3270 input stream. As it is hard to remember these values and hard to understand code containing them, it is a good idea to use symbolic rather than absolute values when testing EIBAID. CICS provides you with a precoded set which you simply copy into your program by writing:

```
COPY DFHAID
```

Figure 44 shows some of the definitions this brings into your program:

```
01  DFHAID.
    02  DFHNULL       PIC X VALUE IS ' '.
    02  DFHENTER      PIC X VALUE IS ''''.
    02  DFHCLEAR      PIC X VALUE IS '_'.
    02  DFHCLRP       PIC X VALUE IS ' '.
    02  DFHPEN        PIC X VALUE IS '='.
    02  DFHOPID       PIC X VALUE IS 'W'.
    02  DFHMSRE       PIC X VALUE IS 'X'.
    02  DFHSTRF       PIC X VALUE IS ' '.
    02  DFHTRIG       PIC X VALUE IS '"'.
    02  DFHPA1        PIC X VALUE IS '%'.
    02  DFHPA2        PIC X VALUE IS '>'.
    02  DFHPA3        PIC X VALUE IS ','.
    02  DFHPF1        PIC X VALUE IS '1'.
    02  DFHPF2        PIC X VALUE IS '2'.
        . . .
        . . .
    02  DFHPF23       PIC X VALUE IS '.'.
    02  DFHPF24       PIC X VALUE IS '<'.
```

Figure 44. The standard attention identifier values

DFHENTER is the ENTER key, DFHPA1 is Program Access (PA) Key 1, DFHPF1 is Program Function Key 1, and so on. As in the case of the DFHBMSCA macro, any values above that appear to be spaces are not; they correspond to bit patterns for which there is no printable character.

3.3.7 Errors on BMS commands

As we cover each group of commands in this Primer, we'll discuss what can go wrong. We'll classify errors according to the categories described in "Handling errors and exceptional conditions" in topic 2.9.2, and suggest how you might want to handle them in your coding. Later, in "Passing control to a specified label" in topic 3.8.2, we'll explain how to branch when an error occurs.

There are two types of errors that can occur in the subset of BMS commands and map options that we've covered here. They are known as MAPFAIL and INVMPSZ. (Others may occur if you use the additional features of BMS outlined in the next section. They are all listed in the CICS/ESA Application Programming Reference .)

Subtopics
3.3.7.1 MAPFAIL errors
3.3.7.2 INVMPSZ errors

3.3.7.1 MAPFAIL errors

MAPFAIL occurs on a RECEIVE MAP command when there are no fields at all on the screen for BMS to map for you. This will happen if you issue a RECEIVE MAP after the user has used one of the "short-read" keys (CLEAR or a program access key) that we discussed in "3270 input data stream" in topic 2.2.4. It can also occur even if the user does not use a short-read key. If, for example, you send a screen to be filled in (without any fields in which the map or the program turns on the modified-data tag), and the user presses the ENTER key or one of the program function keys without keying any data into the screen, you'll get MAPFAIL.

The reason for the failure is essentially the same in both cases. With the short read, the terminal does not send any screen data; hence no fields. In the other case, there are no fields to send, because no modified-data tags have been turned on.

MAPFAIL is almost invariably a user error (or an expected program condition). It may occur on almost any RECEIVE MAP, and therefore you should handle it explicitly in the program. For instance, Figure 45 shows the code that the example application contains to deal with a MAPFAIL that occurs when the menu map is received:

```
MENU-RESEND.
    MOVE REQC TO REQMI.
    MOVE ACCTC TO ACCTMI.
    MOVE PRTRC TO PRTRMI.
    MOVE SNAMEC TO SNAMEMI.
    MOVE FNAMEC TO FNAMEMI.
    MOVE MSG-TEXT (MSG-NO) TO MSGMO.
    EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET')
        CURSOR DATAONLY FRSET ERASEAUP FREEKB END-EXEC.
   . . .
NO-MAP.
    MOVE 2 TO MSG-NO, MOVE -1 TO SNAMEML, GO TO MENU-RESEND.
   . . .
NEW-MENU.
    EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET')
        FREEKB ERASE END-EXEC.
```

Figure 45. Code to handle MAPFAIL

This code is executed if the MAPFAIL condition is raised because the test of the RESP value brings us here. It first tests what key was used to send. (We know it isn't the CLEAR key, having checked that point earlier, to find out if the user wanted to "escape" from the current procedure.) If it was one of the other short-read keys, or if it was ENTER without any data, we know that the screen is still intact and we simply write a message into the message area of the screen reminding the user to use only ENTER or CLEAR, and to key some data in unless he or she is using the CLEAR key to escape. If the failure has some other cause, the program writes the whole map back to the screen, including a similar message, to ensure that the user is looking at a good screen and knows what to do next.

3.3.7.2 INVMPSZ errors

INVMPSZ usually results from a coding error. It occurs on either SEND MAP or RECEIVE MAP if the size of the map specified is too wide for the screen. Therefore, you usually do not need to write code to handle it. If it occurs during debugging, the transaction will end abnormally with a code indicating this error. The cause is either that the SIZE parameter on the DFHMDI macro is wrong, or the terminal is defined incorrectly, or the application is being used from a terminal it does not support. Note that if the last-mentioned cause was a possibility, you might want to write code to send the user a message explaining the problem.

3.3.8 Other features of BMS

BMS is a very powerful component of CICS and offers many facilities beyond those we've discussed so far. We'll list some of the more interesting ones here. You can find guidance on more of the features of BMS in the CICS/ESA Application Programming Guide , and a list of all its macros, commands, and options in the CICS/ESA Application Programming Reference. These features of BMS let you do the following:

Copy what is on a screen to a printer. You can use the ISSUE PRINT command or the local copy facilities of CICS to do this.

Send formatted data to printers in other formats. In this Primer, we discuss only one method of formatting the data for a printer, which is to use a map just like the display screen for the printer, in combination with the PRINT option. However, there are other ways to control the format of printed output, by inserting new-line characters where you want them, and so on.

Build a single screen with a series of SEND MAP commands, using more than one map in the process. This is done with the ACCUM option of SEND MAP.

Build output messages of more than one screen. You can send output messages that consist of a series of screens, which can be stored away by BMS until the entire sequence is complete. Then BMS provides a method for the user to display these screens and page backward and forward through them at will, without any support from your program. Multiple-screen outputs use the PAGING option of SEND MAP. The PAGING and ACCUM options can be used together, incidentally.

Partition a single screen into sections, and treat each of these areas as a separate screen. This needs a terminal with the appropriate partition support, of course. You can write to and read from one of these mini-screens (or **partitions**, as the devices call them) without affecting any of the others.

Send output to terminals other than the one associated with the transaction. This is called **routing.** It provides a second way, different from the technique we'll use, to deal with the requirement in the example application of sending output to a printer.

Switch messages. The routing facility provides a basis for a transaction that can be used to send a message from one terminal to another. Not surprisingly, this is called **message switching.** CICS provides the transaction, which has the identifier CMSG. Any CICS system that includes full-function BMS can make this transaction available.

Write formatted data to terminals without using maps (the SEND TEXT command).

Support additional 3270 features, such as color, the extended attributes (extended color, programmed symbols, extended highlighting, data validation), light pen, cursor select key, and magnetic slot reader.

Support special facilities provided by VTAM, such as **outboard formatting** and **logical device controls**. Guidance information on these facilities is the in sections of the same name in the CICS/ESA Application Programming Guide.

Support a wide variety of terminals with different physical characteristics. BMS even provides facilities for limiting the dependence of the program on the device characteristics; for guidance,

see the section on map set suffixing in the CICS/ESA Application Programming Guide.

Now that you know how to talk to a terminal, the next thing you need to know about is how to get something worthwhile to talk about. This means accessing files, and is the next category of CICS services that we'll cover.

3.4 Handling files

CICS allows you to access file data in a variety of ways. In an online system, most file accesses are random, because the transactions to be processed aren't batched and sorted into any kind of order. Therefore CICS supports the usual direct access methods: VSAM, and DAM. It also allows you to access data using database managers.

Of these, we'll cover only VSAM key-sequenced data sets, accessed by key, in this Primer. Most of the material applies to DAM and other forms of VSAM, however. CICS also supports sequential access in several forms; one of these, browsing, we'll cover in the coming section. The others we'll touch on later.

Before describing how you read and write files, we should explain briefly about an important CICS table, the File Control Table (FCT). This table contains one entry for each file used in any application in the system. (The entries in the FCT can come from RDO FILE definitions that are installed in the CICS system, or from DFHFCT macro statements.) The most important information kept for each file is the symbolic file name. This must match the MVS DDNAME that you use in the JCL defining the file. The JCL statement, in turn, is what connects the name with a real file. When a CICS program makes a file request, it always uses the symbolic file name. CICS looks up this name in the FCT, and from the information there makes the appropriate request of the operating system. This technique keeps CICS programs independent not only of specific data sets (the JCL does that), but of the JCL as well. Usually the symbolic file names are assigned by the CICS systems staff.

In the examples which follow we'll use the symbolic file name "ACCTFIL" for the account file and "ACCTIX" for its index.

Subtopics

3.4.1 Read commands

The read commands that you can use are READ and READNEXT.

Subtopics
3.4.1.1 Reading a file record
3.4.1.2 Browsing a file
3.4.1.3 Using the browse commands in the example application

3.4.1.1 Reading a file record

The command to read a single record from a file is:

```
EXEC CICS READ FILE(filename) INTO(recarea)
    LENGTH(length) RIDFLD(keyarea) option
    option ... END-EXEC.
```

filename
is the name of the file from which you wish to read. It is required in all READ commands. This is the CICS symbolic file name which identifies the FCT entry for the file. File names can be up to 8 characters long and, like any parameter value, should be enclosed in quotes if they are literals.

recarea
is the name of the data area into which the record is to be read, usually a structure in working storage. The INTO is required for the uses of the READ command discussed in this Primer.

length
is the maximum number of characters that may be read into the data area specified. The LENGTH parameter is required for the uses of the READ command we're covering in this Primer, and it must be a halfword binary value (that is, it must have a PICTURE of "S9(4) COMP"). After the READ command is completed, CICS replaces the maximum value you specify with the true length of the record. For this reason, you must specify LENGTH as the name of a data area rather than a literal. For the same reason, you must re-initialize this data area if you use it for LENGTH more than once in the program. An overlength record will raise an error condition.

keyarea
is the name of the data area containing the key of the record you wish to read. This parameter is also required.

option
can be any of the following options which apply to this command. Except where noted, you can use them in any combination.

UPDATE
means that you intend to update the record in the current transaction. Specifying UPDATE gives your transaction exclusive control of the requested record (possibly the whole control interval in the case of VSAM) and invokes the file protection mechanisms we discussed in "Pseudoconversational or not?" in topic 2.7. Consequently, you should use it only when you actually need it; that is, when you are ready to modify and rewrite the record.

EQUAL
means that you want only the record whose key exactly matches that specified by RIDFLD. This is a default option, which you get if you either specify it or fail to specify GTEQ.

GTEQ
means that you want the first record whose key is greater than or equal to the key you specified. You cannot use this option at the same time as EQUAL. It provides one means of doing a **generic read** (a read where only the first part of the key is required to match) and we use it for this purpose in our application.

So, how do we read an account file record? Well, in program ACCT01, we need to read the account file to find out whether the requested record is there or not. The command we need is:

```
EXEC CICS READ FILE('ACCTFIL') RIDFLD(ACCTC) RESP(RESPONSE)
    INTO(ACCTREC) LENGTH(ACCT-LNG) END-EXEC.
```

Here ACCTC is where we've stored the account number taken from the menu map, and ACCT-LNG is a constant in working storage defined as the expected length of a record in the account file:

```
02  ACCT-LNG    PIC S9(4) COMP VALUE +383.
```

We've asked that the record be placed in the data area named "ACCTREC," so ACCTREC should be a data structure corresponding to the file record. We could define this structure directly in the program, but we'll also need it in program ACCT02. So we'll put the record definition into a library and copy it into this program instead:

```
01 ACCTREC.  COPY ACCTREC.
```

In any application, in fact, it is a good idea to keep your record layouts in a library and copy them into the programs that need them. Even in the simplest of applications, the same record is usually used by several programs, and this procedure prevents programs from using different definitions of the same thing.

This argument applies equally well to any structure used in common by multiple programs. Map DSECTs are a prime example, as are parameter lists and communication areas, which we'll discuss later. Apart from its value in the initial programming stage of an application, this technique greatly reduces the effort and hazards associated with any change to a record or map format. You can make the changes in just one place (your library) and then simply recompile all the affected programs.

Subtopics
3.4.1.1.1 The account file record format
3.4.1.1.2 The index file record format

3.4.1.1.1 The account file record format

Figure 46 shows the COBOL record definition we need for the account file in the example application.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦       *    ACCTREC - ACCOUNT FILE RECORD                                ¦
¦            02  ACCTDO               PIC X(5).                         ¦
¦            02  SNAMEDO              PIC X(18).                        ¦
¦            02  FNAMEDO              PIC X(12).                        ¦
¦            02  MIDO                 PIC X.                            ¦
¦            02  TTLDO                PIC X(4).                         ¦
¦            02  TELDO                PIC X(10).                        ¦
¦            02  ADDR1DO              PIC X(24).                        ¦
¦            02  ADDR2DO              PIC X(24).                        ¦
¦            02  ADDR3DO              PIC X(24).                        ¦
¦            02  AUTH1DO              PIC X(32).                        ¦
¦            02  AUTH2DO              PIC X(32).                        ¦
¦            02  AUTH3DO              PIC X(32).                        ¦
¦            02  AUTH4DO              PIC X(32).                        ¦
¦            02  CARDSDO              PIC X.                            ¦
¦            02  IMODO                PIC X(2).                         ¦
¦            02  IDAYDO               PIC X(2).                         ¦
¦            02  IYRDO                PIC X(2).                         ¦
¦            02  RSNDO                PIC X.                            ¦
¦            02  CCODEDO              PIC X.                            ¦
¦            02  APPRDO               PIC X(3).                         ¦
¦            02  SCODE1DO             PIC X.                            ¦
¦            02  SCODE2DO             PIC X.                            ¦
¦            02  SCODE3DO             PIC X.                            ¦
¦            02  STATDO               PIC X(2).                         ¦
¦            02  LIMITDO              PIC X(8).                         ¦
¦            02  PAY-HIST OCCURS 3.                                     ¦
¦                04  BAL              PIC X(8).                         ¦
¦                04  BMO              PIC 9(2).                         ¦
¦                04  BDAY             PIC 9(2).                         ¦
¦                04  BYR              PIC 9(2).                         ¦
¦                04  BAMT             PIC X(8).                         ¦
¦                04  PMO              PIC 9(2).                         ¦
¦                04  PDAY             PIC 9(2).                         ¦
¦                04  PYR              PIC 9(2).                         ¦
¦                04  PAMT             PIC X(8).                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 46. The COBOL record definition for the account file

We'll not dwell on the naming conventions of the data items that we're leaving to our assumed batch processing system. Nor shall we have anything much to say about the behavior of this batch system. In other words, don't worry about it!

3.4.1.1.2 The index file record format

We also need a record definition for the index file records. See Figure 47.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦       *    ACIXREC - INDEX FILE RECORD                                 ¦
¦            02  SNAMEDO             PIC X(12).                         ¦
¦            02  ACCTDO              PIC 9(5).                          ¦
¦            02  FNAMEDO             PIC X(7).                          ¦
¦            02  MIDO                PIC X.                             ¦
¦            02  TTLDO               PIC X(4).                          ¦
¦            02  ADDR1DO             PIC X(24).                         ¦
¦            02  STATDO              PIC X(2).                          ¦
¦            02  LIMITDO             PIC X(8).                          ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 47. The COBOL record definition for the index file records

You may notice that we've chosen many of the field names in the account record to match the output subfields in the detail map. We did this because when we display a record from the file on the screen, we have to move many fields from the record to the symbolic description map. This choice of names allows us to use MOVE CORRESPONDING instead of writing out the individual moves. It allows us to do the same thing going from the screen to the file, because the input and output fields on the screen overlay each other exactly, as we noted earlier.

3.4.1.2 Browsing a file

In program ACCT01, when we search by name, we need to point to a particular record in the file, based on a random key. Then we start reading the file sequentially from that point on. The need for this combination of random and sequential file access, called **browsing**, arises frequently in online applications. Consequently, CICS provides a special set of browse commands: STARTBR, READNEXT, and ENDBR.

Before we look at these commands, a few words about the performance implications of browsing. Transactions that produce lots of output screens can monopolize system resources. A file browse is often guilty of this. Just having a long browse can put a severe load on the system, locking out other transactions and increasing overall response time.

You see, CICS assumes the terminal operator initiates a transaction that accesses a few data records, processes the information, and returns the results to the operator. This process involves numerous waits that allow CICS to do some multitasking. However, CICS is **not** an interrupt-driven multitasking system; tasks that involve small amounts of I/O relative to processing can monopolize the system regardless of priority. A browse of a highly-blocked file is just such a transaction.

You can issue **DELAY** or **SUSPEND** commands from time to time, so that other tasks can get control. If the browse does indeed produce paged output, you should probably break the transaction up in one of the ways suggested in the topic on designing efficient applications in the CICS/ESA Application Programming Guide.

Subtopics
3.4.1.2.1 Starting the browse operation
3.4.1.2.2 Reading the next record
3.4.1.2.3 Finishing the browse operation

3.4.1.2.1 Starting the browse operation

The STARTBR (start browse) command gets the process started. It tells CICS where in the file you want to start reading. The format is:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS STARTBR FILE(filename)                                      ¦
¦       RIDFLD(keyarea) option END-EXEC.                                  ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The FILE and RIDFLD parameters are the same as in a READ command. The options allowed are GTEQ and EQUAL; you cannot use them both. They are defined as for READ, except that this time GTEQ is assumed by default. UPDATE isn't allowed; file browsing is strictly a read-only operation.

3.4.1.2.2 Reading the next record

Starting a browse does *not* make the first eligible record available to your program; it merely tells CICS where you want to start when you begin issuing the sequential read commands.

To get the first record, and for each one in sequence after that, you use the READNEXT command:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   EXEC CICS READNEXT FILE(filename)                                    ¦
¦       INTO(recarea) LENGTH(length)                                     ¦
¦       RIDFLD(fdbkarea) END-EXEC.                                       ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

The FILE, INTO and LENGTH parameters are defined in the same way as they are in the READ command. You only need the FILE parameter because CICS allows you to browse several files at once, and this tells which one you want to read next. Note, however, that you cannot name a file in a READNEXT command unless you've first issued a STARTBR command for it.

The RIDFLD parameter is used in a somewhat different way. On the READ and STARTBR commands, RIDFLD carries information from the program to CICS; on READNEXT, the flow is primarily in the other direction: RIDFLD points to a data area into which CICS will "feed back" the key of the record it just read. Do make sure that RIDFLD points to an area large enough to contain the full key; otherwise the adjacent field(s) in storage will be overwritten. Don't change it, either, because you'll interrupt the sequential flow of the browse operation.

(There **is** a way to do what is called "skip sequential" processing in VSAM by altering the contents of this key area between READNEXT commands. Although we won't be covering this here, we mention it only to explain why you should not inadvertently change the contents of "fdbkarea" while browsing the file.)

3.4.1.2.3 Finishing the browse operation

When you've finished reading a file sequentially, you terminate the browse with the ENDBR command:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   EXEC CICS ENDBR FILE(filename) END-EXEC.                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Here FILE functions as it did in the READNEXT command; it tells CICS *which* browse is being terminated, and it must name a file for which a STARTBR has been issued earlier.

3.4.1.3 Using the browse commands in the example application

Let's write the code we need to do the example. The first thing we have to do is construct a key that will start the browse in the right place. The key of the index file consists of the first 12 characters of the surname followed by an account number. We want to build a key that consists of the characters the user keyed in as the surname, followed by something smaller than any file key that starts out the same way. Then we can use the GTEQ option on our STARTBR command to get the first qualifying record. If we define:

```
       04  BRKEY.
           06  BRKEY-SNAME PIC X(12).
           06  BRKEY-ACCT  PIC X(5).
```

Then writing:

```
       MOVE SNAMEC TO BRKEY-SNAME.
       MOVE LOW-VALUES TO BRKEY-ACCT.
```

should do the trick. SNAMEC is where we saved the surname from the input menu (SNAMEMI) earlier in the code. Because CICS pads what the user keys with spaces to produce SNAMEMI, and spaces are lower in the collating sequence than any letter, we can be sure that BRKEY will be smaller than the key of any eligible record in the file.

We also need to know where to stop the browse.

Certainly we'll stop when we overflow the display capacity of the screen, but we may run out of eligible names before that. So we need to construct a surname value that is the highest alphabetically that could meet our match criteria. If the surname in the record exceeds this value, we will know that we've read all the (possibly) eligible records. If this limiting value is named MAX-SNAME and has a picture of "X(12)," then:

```
       MOVE SNAMEC TO MAX-SNAME.
       TRANSFORM MAX-SNAME FROM SPACES TO HIGH-VALUES.
```

should give the right cutoff.

Finally, as we read, we need to test whether the first name matches sufficiently to display the record on the screen or not. If we define MIN-FNAME as the smallest allowable value and MAX-FNAME as the largest, and if FNAMEC is where we held the first name from the input screen, then we need the following code:

```
       MOVE FNAMEC TO MIN-FNAME, MAX-FNAME.
       TRANSFORM MIN-FNAME FROM SPACES TO LOW-VALUES.
       TRANSFORM MAX-FNAME FROM SPACES TO HIGH-VALUES.
```

Thus, Figure 48 in topic 3.4.1.3 shows the code we need to produce the name summary.

```
 SRCH-RESUME.
      EXEC CICS STARTBR FILE('ACCTIX') RIDFLD(BRKEY) GTEQ
          RESP(RESP) END-EXEC.
      IF RESP = DFHRESP(NOTFND) GO TO SRCH-ANY.
      IF RESP NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.
      BUILD NAME DISPLAY.
 SRCH-LOOP.
      EXEC CICS READNEXT FILE('ACCTIX') INTO(ACIXREC)
          LENGTH(ACIX-LNG) RIDFLD(BRKEY) RESP(RESP) END-EXEC.
      IF RESP = DFHRESP(ENDFILE) GO TO SRCH-DONE.
      IF RESP NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.
      IF SNAMEDO IN ACIXREC > MAX-SNAME GO TO SRCH-DONE.
      IF FNAMEDO IN ACIXREC < MIN-FNAME OR
          FNAMEDO IN ACIXREC > MAX-FNAME, GO TO SRCH-LOOP.
      ADD 1 TO LINE-CNT.
      IF LINE-CNT > MAX-LINES,
          MOVE MSG-TEXT (15) TO MSGMO,
          MOVE DFHBMBRY TO MSGMA, GO TO SRCH-DONE.
      MOVE CORRESPONDING ACIXREC TO SUM-LINE.
      MOVE SUM-LINE TO SUMLNMO (LINE-CNT).
      GO TO SRCH-LOOP.
 SRCH-DONE.
      EXEC CICS ENDBR FILE('ACCTIX') END-EXEC.
```

Figure 48. The name summary search code

This code first starts a browse on the index file. Then it begins a loop in which it:

1. Reads the next sequential record in the file.

   This may result in an ENDFILE condition, causing a transfer to paragraph SRCH-DONE.

2. Tests whether the surname in the record is beyond the last in the file that might qualify, and exits the loop to SRCH-DONE if so.

3. Otherwise, determines if the record is eligible on the basis of first name and, if not, returns to the beginning of the loop to check the next record.

4. Determines, if the record is eligible, if it will still fit on the screen. (We need to read one "hit" beyond the point of using up all the space on the screen so that we can tell the user whether there are going to be more names or not.)

5. Adds a message to the output map if the current name won't fit, saying there are more names and how to get them, and then exits the loop at SRCH-DONE.

6. Builds an output line for the map if the name will fit, and returns to the beginning of the loop to check for more hits.

After the loop, at SRCH-DONE, when all eligible names have been read or the screen is full, the program terminates the browse. At this point, the name search output is essentially ready to be sent back to the user.

There are two other browse commands. We'll not cover them here, but you

can find a complete list of them in the CICS/ESA Application Programming Reference. The READPREV command is almost like READNEXT, except that it lets you proceed backward through a data set instead of forward. The RESETBR command allows you to reset your starting point in the middle of a browse.

3.4.2 Write commands

There are three file output commands: REWRITE modifies a record that is already on a file, WRITE adds a new record, DELETE deletes an existing record from a file.

Subtopics

3.4.2.1 Rewriting a file record

The REWRITE command updates the record you've just read. You can use it only after you've performed a "read for update" by executing a READ command for the same record with UPDATE specified. REWRITE looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS REWRITE FILE(filename)                                      ¦
¦        FROM(recarea) LENGTH(length) END-EXEC.                           ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

filename
has the same meaning as in the READ command: it is the CICS name of the file you are updating. You must specify it.

recarea
is the name of the data area that contains the updated version of the record to be written to the file. This parameter is also required.

length
is the length of the (updated) version of the record. You must specify length, as in a READ command, and it must be a halfword binary value.

3.4.2.2 Adding (writing) a file record

The WRITE command adds a new record to the file. The parameters for WRITE are almost the same as for REWRITE, except that you have to identify the record with the RIDFLD option. (You do not do this with the REWRITE command because the record was identified by the previous READ operation on the same data set.) The format of the WRITE command is:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS WRITE FILE(filename) FROM(recarea)                          ¦
¦       LENGTH(length) RIDFLD(keyarea) END-EXEC.                          ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

keyarea
is the data area containing the key of the record to be written. The RIDFLD parameter is required on the WRITE command.

3.4.2.3 Deleting a file record

The DELETE command deletes a record from the file, and looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS DELETE FILE(filename)                                       ¦
¦        RIDFLD(keyarea) END-EXEC.                                        ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The parameters are defined in the same way as for the WRITE and REWRITE commands. You can delete a record directly, without reading it for update first. When you do this you must specify the key of the record to be deleted by using RIDFLD. Alternatively, you can decide to delete a record after you've read it for update. In this case, you must omit RIDFLD.

3.4.2.4 Using the write commands in the example application

Program ACCT02 uses all three of the file output commands. For add requests, the program first constructs a new record in a structure named NEW-ACCTREC. It then issues the command:

```
EXEC CICS WRITE FILE('ACCTFIL') FROM(NEW-ACCTREC)
    RIDFLD(ACCTC) LENGTH(ACCT-LNG) END-EXEC.
```

(The variables ACCTC and ACCT-LNG have the same definition as they did in the example of the READ command in "Reading a file record" in topic 3.4.1.1.)

For a modification, the program first reads the record in question, with UPDATE specified:

```
IF REQC NOT = 'A',
    EXEC CICS READ FILE('ACCTFIL') INTO(OLD-ACCTREC)
        RIDFLD(ACCTC) UPDATE LENGTH(ACCT-LNG) END-EXEC.
```

Then it builds a new version of the record, again at NEW-ACCTREC, by combining the new data from the screen with the old record. Finally it replaces the old record with the new one, in the command:

```
  EXEC CICS REWRITE FILE('ACCTFIL') FROM (NEW-ACCTREC)
      LENGTH(ACCT-LNG) END-EXEC.
```

For a deletion, the program uses the same READ command as in a modification. Therefore the key (RIDFLD) isn't specified in the DELETE command, which is:

```
  EXEC CICS DELETE FILE('ACCTFIL') END-EXEC.
```

3.4.3 Errors on file commands

In contrast to the situation with BMS commands, a wide variety of things can go wrong on the file commands. Here are the errors that can arise when you use the subset of file commands that we've just described.

DISABLED
occurs if a file is disabled. A file may be disabled because:

It was initially defined as disabled and has not been enabled since

It has been disabled by an EXEC CICS SET command or by the CEMT transaction.

DUPKEY
means that if a VSAM record is retrieved by way of an alternate index with the NONUNIQUEKEY attribute, and another alternate index record with the same key follows. It does not occur as a result of a READNEXT command that reads the last of the records having the nonunique key.

DUPREC
means that there is already a record in the file with the same key as the one that you are trying to add with a WRITE command. This condition may result from a user error or may be expected by the program. In either of these cases, there should be specific code to handle the situation.

It can also fall into the "should-not-occur" category, the third type in the list under "Handling errors and exceptional conditions" in topic 2.9.2, as it would in our example application. In this case no special code is required beyond identifying the problem to the user. The message to the user should tell him or her what to say to the supervisor (or to the operations staff) and what he or she is allowed to do next.

ENDFILE
means that you've attempted to read sequentially beyond the end of the file in a browse (using the READNEXT command). This is a condition that you should program for in any browse. In the example application, for instance, a search on "Zuckerman" or a similar name might cause ENDFILE, and we'll code for it explicitly by sending control to SRCH-DONE when it occurs.

FILENOTFOUND
means that the symbolic file name in a file command cannot be found in the File Control Table. This is usually a coding error; look for a difference in spelling between the command and the FCT entry. If it happens after the program is put into actual use ("in production"), look for an accidental change to the entry for that file in the FCT.

ILLOGIC
is a catch-all class for errors detected by VSAM that don't fall into one of the other categories that CICS recognizes. The RESP2 value will tell you the specific error.

Note: Before CICS/VS 1.7, by far the most common cause used to be trying to read from or write into a brand-new (empty) VSAM key-sequenced data set (KSDS). In order to use a KSDS in CICS, you had to batch load at least one record into it, because VSAM does not build the index component until the first record arrives, and CICS was unable to cope with a KSDS whose index isn't built.

INVREQ

means that CICS regards your command as an invalid request for one of the following reasons:

You requested a type of operation (add, update, browse, and so on) that wasn't included in the "service requests" (SERVREQ) parameter of the FCT entry for the file in question.

You tried to REWRITE a record without first reading it for update.

You issued a DELETE command without specifying a key (RIDFLD), and without first reading the target record for update.

You issued a DELETE command specifying a key (RIDFLD) for a VSAM file when a read for update command is outstanding.

After one read for update, you issued another read for update for another record in the same file without disposing of the first record (by a REWRITE, UNLOCK, or DELETE command).

You issued a READNEXT or an ENDBR command without first doing a STARTBR on the same file.

Almost all of these INVREQ situations result from program logic errors and should disappear during the course of debugging. The first one, however, can also result from an inadvertent change to the "service requests" parameter in the FCT entry for the file.

IOERR
means that the operating system is unable to read or write the file, presumably because of physical damage. This can happen at any time, and there is usually nothing to do in the program except to abend the transaction and inform the user of the problem.

ISCINVREQ
means that the remote system indicates a failure which does not correspond to a known condition.

LENGERR
could mean one of the following:

You omitted the LENGTH parameter from a READ, READNEXT, WRITE or REWRITE command.

The length you specified on a WRITE or REWRITE operation was greater than the maximum record size for the file. (See the description of LENGTH options in the CICS/ESA Application Programming Reference for a description of a safe upper limit.)

You specified a length shorter than the actual record length on a READ operation to a file of variable length records.

You indicated a wrong length on a READ, READNEXT, WRITE or REWRITE command to a file containing fixed-length records.

LENGERR is usually caused by a coding error.

NOSPACE
means that there's no space in the file to fit the record you've just tried to put there with a WRITE or REWRITE command. This doesn't mean that there's no space at all in the data set; it simply means that the record with the particular key you specified will not fit until the file is extended or reorganized. Like IOERR, this condition may occur at any time, and should be handled accordingly.

NOTAUTH

means that a resource or command security check has failed.

NOTFND condition
means that there is no record in the file with the key specified in the RIDFLD, parameter on a READ, READNEXT, STARTBR, or DELETE command. (4) **NOTFND** may result from a user error, may be expected by the program, or may indicate an error in the program logic. In our example application, we provide code to handle all three of these situations.

In program ACCT01, when we check to see if the requested account record is on file, we expect **NOTFND** if the request is to add a record. However, it shows a user error (in the account number) if it happens on any other type of request. For both these cases, we need to provide recovery code. On the other hand, by the time we get to program ACCT02, we should have removed all the possibilities for getting a "not found" response on a read. So its occurrence here would signal an error in our logic, to be handled like any other unexpected error.

NOTOPEN
occurs if:

The requested file is **CLOSED** and **UNENABLED.** The **CLOSED, UNENABLED** state is reached after a close request has been received against an **OPEN ENABLED** file and the file is no longer in use.

The requested file is still open and in use by other requests, but a close request against the file has been received. Existing users are allowed to complete.

This condition can occur only during the execution of the following commands:

READ
WRITE
The first command in a **WRITE MASSINSERT** sequence
DELETE
The first command in a **DELETE GENERIC** sequence
STARTBR.

Other commands cannot raise this condition because they are part of an active request.

This condition does not occur if the request is made to either a **CLOSED, ENABLED** file or a **CLOSED, DISABLED** file. In the first case, the file is opened as part of executing the request. In the second case, the **DISABLED** condition is raised.

This condition may also occur when a file control command refers to a file defined as **REMOTE,** where the remote system is a release of CICS earlier than 1.7. The condition can then occur in response to any file control command.

As you have probably gathered from this description, **NOTOPEN** usually results from an operations problem, and you may want to notify the operations staff of the problem, or send a message to the user to do so.

SYSIDERR
means that the SYSID option specifies either a name that is not defined in the intersystem table or a system to which the link is closed.

(4) It **is** possible to raise NOTFND on a READNEXT command, but

```
only in connection with skip sequential processing--and
that's beyond the scope of the Primer.
```

3.4.4 Other file services

Before leaving the topic of file commands, we'll list some of the other facilities that are available. You can find guidance information on using file control in the CICS/ESA Application Programming Guide , and a full list of commands, options, and exceptional conditions in the CICS/ESA Application Programming Reference.

You can use relative-record VSAM files (RRDS) as well as key-sequence files (KSDS), and you can access a KSDS by relative byte address (RBA) instead of a key.

You can use VSAM files with alternate indexes

You can use BDAM files

You can specify a partial **generic**) key for a VSAM KSDS. The effect is similar, but not identical, to what we did in the browse example, where we used a full-key filled out with spaces and low-values in combination with the GTEQ option.

You can release a record that you've read for update if you decide no to update after all. The UNLOCK command is the means of doing this.

You can access records without moving them into your program by usin the SET option on the READ command.

You can delete a whole block of adjacent records in a VSAM file with single command (using the "generic delete" option).

You can insert a whole block of records at once into a VSAM fil ("mass insert" option).

You also can use VSAM entry-sequenced data sets (ESDS)

ESDS is another type of sequentially organized data for which support is provided in CICS (the first was browsing). Two other forms of sequential support are also available, but they aren't considered to be part of CICS's file services. One of these is the extrapartition transient data facility, which allows you to read or write SAM files. In addition, the intrapartition transient data and temporary storage facilities provide a means for reading and writing data in queues, providing another form of sequential support. See "Saving data and communicating between transactions" in topic 3.5.

3.5 Saving data and communicating between transactions

Subtopics

3.5.1 The need for scratchpad and queuing facilities

Most of the sequential file facilities we mentioned in the previous topic are provided because we need to save data from the execution of one transaction, passing it on to another that occurs later. We've already seen two instances of this requirement in our example application.

The first resulted from our decision to use pseudoconversational transactions; we need to save data from one interaction with the terminal to the next, even though no task exists for that terminal for most of the intervening time. For this we need some sort of scratchpad facility.

The second requirement came from our need to log the changes to the account file. Here we require some sort of queuing facility: a way to add items to a list (one in each update transaction) and read them later (in the log-print transaction).

There are several different scratchpad areas in CICS that you can use to transfer and save data, within or between transactions. One of them is **temporary storage**, which we'll cover in a moment. Others are listed below. The CICS/ESA Application Programming Reference gives you a complete list of the commands you can use to get access to these areas.

A **Communication Area** or COMMAREA. This is an area used for passing data both between programs within a transaction and between transactions at a given terminal. We'll describe it in connection with the program control commands in "Program control" in topic 3.6. The COMMAREA is the recommended scratchpad area.

It's the COMMAREA that offers an alternative solution to our double updating problem. For example, it would be perfectly feasible for ACCT01 to pass the contents of the account file record over to ACCT02 in the COMMAREA. ACCT02 could then re-retrieve the account record for update and compare it with the version passed in COMMAREA. Any difference would show that some other task had changed the account record.

Although this solution may be easier to code, it isn't as good from the user's point of view. You see, with this scheme, we don't find out about any conflict over the record until we're ready to update it. Unfortunately, that means we then have to tell one user that his or her update cannot be made, but we can't tell them until they've keyed in all the changed data.

The **Common Work Area** (known as the CWA). Any transaction can access the CWA, and since there's only one CWA for the whole system, the format and use of this area must be agreed upon by all transactions in all applications that use it.

The **Transaction Work Area** (TWA). The TWA exists only for the duration of a transaction. Consequently, you can use it to pass data among programs executed in the same transaction (like COMMAREA), but not between transactions (unlike COMMAREA). The TWA isn't commonly used in command level programs.

3.5.2 Temporary storage

CICS provides two queuing facilities: temporary storage and transient data. The following paragraphs tell you how to use temporary storage, both for queuing and as a scratchpad. Later, in "Transient data" in topic 3.5.3, we give a brief description of transient data, outline the differences between the two facilities, and suggest when you might use one or the other.

Temporary storage is just a sequential file; a VSAM data set on a disk, or an area of main storage.

The CICS temporary storage facilities allow a task to create a queue of items, stored under a name selected by the task. This queue, which you can think of as a miniature sequential file, exists until some task deletes it. The task that deletes it isn't usually the same task that created it, although of course it could be. The queue can hold any number of items (from just one to 32767) and any number of different tasks can add to it, read it, or change the contents of items in it.

When there is just one item in a queue, we think of this facility as a scratchpad; when there is more than one, we think of it as a queuing facility. The items can be of almost any length, and they can be of different lengths for the same queue. If you are using the queue as a temporary sequential file, you can think of the items in it as records.

Subtopics

3.5.2.1 Adding to, and creating, a temporary storage queue

The command to add one item to an existing temporary storage queue, or to create a brand new queue with one item in it, looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS WRITEQ TS QUEUE(qname) FROM(recarea)                        ¦
¦       LENGTH(length) option option ... END-EXEC.                        ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

qname
is the name of the queue to which an item is to be added. If there is no queue with the name you specify, CICS will create one, with the item you specified as the first (and only) item in the queue. Queue names are up to eight characters long. CICS imposes no restrictions on what names may be used, but there are some things to be considered in choosing names, as we will point out later. You should put this name in quotes if it is a literal.

recarea
is the name of the data area containing the item to be added.

length
is the length of that item (record). As in the file commands, length is given as a halfword binary value ("PIC S9(4) COMP").

option
may be any of the following:

MAIN
causes the item to be written to an area of main storage rather than to disk. Only use this option for queues of small size and very short lifetimes.

AUXILIARY
is the opposite of MAIN and causes the item to be written to a special VSAM data set on disk. This is the default (you get it if you specify AUXILIARY or if you fail to specify MAIN) and is what you should use in most circumstances.

ITEM(itemno)
causes CICS to feed back the number of items held in the queue after completion of the command. This number is placed in the "itemno" data area, and you can check the contents after issuing the command. Like the length, the item number is always a halfword binary value.

The MAIN or AUXILIARY option is effective only on the initial write that creates a new queue because a single temporary storage queue cannot be split between main storage and auxiliary storage. It is ignored on subsequent writes.

3.5.2.2 Replacing items in a temporary storage queue

Besides adding items to a queue, you can also replace any item in an existing queue by specifying the REWRITE option. The command:

```
EXEC CICS WRITEQ TS QUEUE(qname) FROM(recarea)
    LENGTH(length) ITEM(itemno) REWRITE END-EXEC.
```

replaces the item whose number is stored in the "itemno" data area. Notice that the function of the ITEM option is quite different from its function when you write a new item. On a REWRITE, it is required, and passes information from your program to CICS. When you are adding new items to a queue, it is optional, and is used to return information from CICS to your program. The other parameters have the same meanings as above.

3.5.2.3 Reading temporary storage queues

To read an item from a temporary storage queue, you use:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS READQ TS QUEUE(qname) INTO(recarea)                         ¦
¦       LENGTH(length) option END-EXEC.                                   ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

qname
is the name of the queue you want to read. Put qname in quotes if it is a literal.

recarea
is the name of the data area into which you want to read the item.

length
is the name of a data area (defined as a binary halfword) with two functions:

1. Before issuing the command, you place in this area the maximum length of record that the program will accept (that is, the length of "recarea"), so that storage overlay will not occur if you read an unexpectedly long record. If the record is longer than this length, CICS will truncate it to this size and also turn on the LENGERR condition (about which more later).

2. CICS also returns the true length of the record (before any truncation) in this area at the completion of the command.

option
may be either of two choices to indicate which record you want:

ITEM(itemno)
indicates that the number of the item to be read is stored at "itemno" (in halfword binary form).

NEXT
means that the next item on the queue is to be read. The first time a READQ TS NEXT is issued for a queue by any transaction, the first item is provided. The next time this command is issued, *by any transaction*, the second item is provided, and so on. Moreover, the use of the ITEM option *by any transaction* resets what CICS considers the "next" item to the one following that specified in the ITEM option. Therefore, if more than one transaction can be reading a single queue, you may want to use the ITEM option to ensure that you read the intended item. NEXT is the default, if you do not indicate either NEXT or ITEM.

You can read temporary storage queues, wholly or in part, any number of times. So, reading the queue does not affect the contents of the queue.

3.5.2.4 Deleting temporary storage queues

Once a temporary storage queue has been created, it stays in existence until explicitly deleted by some transaction. The command to delete a queue is:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS DELETEQ TS QUEUE(qname) END-EXEC.                           ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

where "qname" has the same meaning as on a READQ or WRITEQ command.

Notice that you cannot delete individual items from a temporary storage queue; you have to delete the whole queue.

3.5.2.5 Naming temporary storage queues

In writing any application that uses temporary storage, you should choose your queue names with care. First of all, you should follow a convention for constructing names to ensure that unrelated transactions don't inadvertently use the same queue name. For this reason, many installations insist that all queue names begin with characters that identify the application involved. Usually two to four characters are reserved for this purpose, depending on the installation. In our example, for instance, we start all our temporary storage queue names with the letters *AC*.

Queue names in CICS also provide a means of random access to scratchpad information. In our example, we're interested in keeping information about account numbers in a scratchpad area. If we include the account number in the queue name, we can read the scratchpad information concerning that account number directly, without any need to search the scratchpad.

Another example of using the queue name as an index occurs when you store data between transactions for a particular terminal. In this case, the first of two transactions stores the data to be passed in a queue whose name is formed from the terminal name plus some constant. The last four letters of the queue name are most often used for the terminal identifier. Then the second transaction can find the data for its terminal directly, by constructing the queue name from the name of its own input terminal plus the same constant.

3.5.2.6 Using temporary storage in the example application

Let's see how we'll use temporary storage in the example application for our scratchpad requirements. In program ACCT01, we need to find out whether any other task is currently updating the account record that our terminal has asked to update.

We want to observe the house rule that all temporary storage for this particular application should start with the letters "AC", and at the same time take advantage of the indexing aspect of temporary storage names; so we'll do as follows: we'll have one temporary storage queue for each account number in use. The name of the queue will be "AC0" followed by the account number, defined as follows in working storage. (The 0 merely fills out the queue name to the allowed eight characters.)

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦       02  USE-QID.                                                    ¦
¦           04  USE-QID1    PIC X(3) VALUE 'AC0'.                       ¦
¦           04  USE-QID2    PIC X(5).                                   ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

The queue will contain just one item, which will tell what terminal is updating the record for that account number, and the date and time at which it started doing so. The definition of this record, also in working storage, will be:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦       02  USE-REC.                                                    ¦
¦           04  USE-TERM    PIC X(4) VALUE SPACES.                      ¦
¦           04  USE-TIME    PIC S9(7) COMP-3.                           ¦
¦           04  USE-DATE    PIC S9(7) COMP-3.                           ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

We include the date and time along with the terminal name in the scratchpad entry, so that we can find out whether the account number is currently in use, or whether the scratchpad record is there because of an earlier update attempt that wasn't completed properly. See "Pseudoconversational or not?" in topic 2.7 for a discussion of this possibility.

The first test to check whether the record is in use, then, is:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦  MOVE ACCTC TO USE-QID2.                                               ¦
¦  EXEC CICS READQ TS QUEUE(USE-QID) INTO(USE-REC)                       ¦
¦      ITEM(USE-ITEM) LENGTH(USE-LNG) RESP(RESPONSE) END-EXEC.           ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Here USE-ITEM and USE-LNG are defined in working storage and have initial values of 1 and 12, respectively.

The response we're hoping for on this command is that the read failed because no such queue exists. This will raise the QIDERR exception condition. If we do not get this response, we'll have to look at the scratchpad entry that we read to see whether this is a recent entry or an old, expired one. To do this we'll simply compare the time and date in the scratchpad entry with the time and date when the current transaction started (information that is available in the EIB).

If we find out that the account number is not in use, then the next step is to claim it for the terminal that entered the input. If there is no scratchpad record for this number, then we need:

```
MOVE EIBTRMID TO USE-TERM, MOVE EIBTIME TO USE-TIME.
MOVE EIBDATE TO USE-DATE.
EXEC CICS WRITEQ TS QUEUE(USE-QID) FROM(USE-REC)
    LENGTH(12) END-EXEC.
```

If, on the other hand, there was an old, expired record in temporary storage for this number, then the code required is:

```
MOVE EIBTRMID TO USE-TERM, MOVE EIBTIME TO USE-TIME.
MOVE EIBDATE TO USE-DATE.
EXEC CICS WRITEQ TS QUEUE(USE-QID) FROM(USE-REC)
    LENGTH(12) ITEM(USE-ITEM) REWRITE END-EXEC.
```

Here again USE-ITEM is defined to be a halfword binary value of 1, because we want to rewrite the first (and presumably only) item in the queue.

This same scratchpad entry gets erased in program ACCT02 when we've finished updating, with the command:

```
EXEC CICS DELETEQ TS QUEUE(USE-QID) END-EXEC.
```

where the data area USE-QID has been defined and set up in the same way as it was in program ACCT01.

3.5.2.7 Errors on temporary storage commands

You can experience six different types of error on the temporary storage commands that we've described:

INVREQ
means that the record length you specified is invalid (zero or negative). This is almost always the result of a problem in the code.

IOERR
means the same thing on a temporary storage command as it does on a file command. It means that there is an unrecoverable input/output error, in this case on the temporary storage file, a VSAM entry-sequenced data set (ESDS).

ISCINVREQ
means that the remote system indicates a failure that does not correspond to a known condition.

ITEMERR
means that you specified an item number that does not exist. This can happen on either a READQ TS command or a WRITEQ TS with REWRITE specified. ITEMERR may be a condition the program expects, such as when a program reads until it exhausts a queue, or it may result from an error in the program logic.

LENGERR
occurs when you read an item that is longer than the maximum specified in the LENGTH parameter. It usually means a problem in the program logic.

NOSPACE
means that there isn't enough space left in the temporary storage data set, or in main storage (if MAIN is specified) for the record you just wrote. Unlike what happens with most other error conditions, CICS does not terminate your task when this occurs. If you provide code to handle the possibility, CICS sends control there, as it does for any unusual condition. If you don't, CICS simply suspends the task until some other task in the system releases enough temporary storage space for your record to fit.

NOTAUTH
means that a resource or command security check has failed. There is a complete list of reasons for such failures in the section on NOTAUTH in the CICS/ESA Application Programming Reference.

QIDERR
means that the queue that you've named in a READQ command, or in a WRITEQ with REWRITE specified, does not exist. It might indicate a program error, or it might be a condition expected by the program. When we read temporary storage to find out whether a particular account number is in use, for example, QIDERR is the expected response and indicates that the account number in question is not in use.

SYSIDERR
means that the SYSID option specifies either a name which is not defined in the intersystem table, or a system to which the link is closed.

3.5.3 Transient data

There is another facility in CICS, called **transient data**, one form of which is very similar to temporary storage. It comes in two flavors--**intrapartition** and **extrapartition**--and it is intrapartition transient data that is so much like temporary storage. Both temporary storage and transient data allow you to write and read queues of data items, which are often essentially small sequential files. Like temporary storage queues, intrapartition transient data queues are kept in a single VSAM data set managed by CICS.

There are some important differences, however:

You must define the name and certain other characteristics of ever transient data queue to CICS in the **Destination Control Table** (DCT). This means that the names must be known before CICS is brought up, so you cannot just create a transient data queue with an arbitrary name, as we did for temporary storage in the example.

You cannot modify an item in a transient data queue; you can only ad new items to the end of the queue. The Write Transient Data command has nothing corresponding to the ITEM option.

Transient data queues must be read sequentially. That is, the Rea Transient Data command has nothing corresponding to the ITEM option.

Furthermore, a read operation on transient data is a **destructive read**. That is, once a transaction has read an item on the queue, that item cannot be read again by that transaction or by any other.

Transient data comes with a very useful mechanism known as a **trigger**. You can request, in the DCT, that CICS initiate a transaction whenever the number of items in a transient data queue reaches a certain value. The DCT entry for the queue tells what this critical number of items is (the "trigger level"), and the name of the transaction to be initiated. You can also specify that a particular terminal must be available to this transaction. (You do this simply by giving the same name to both the terminal and the queue.) In this case, the transaction doesn't start until both the trigger level is reached and the terminal in question is available.

This can be very useful for printing, as you'll soon see.

Transient data queues are always written to a file; there is n counterpart to the MAIN option that is used in temporary storage commands.

The recovery options for transient data are more varied

Extrapartition transient data is the means by which CICS supports standard sequential (SAM) files. The commands used for extrapartition queues are the same as for intrapartition queues, and each queue requires a DCT entry. In this case, however, a read or write operation is actually a read or write to a sequential file, and each queue is a file. You can either read or write an extrapartition queue, but not both. The trigger mechanism and the recovery options mentioned above do not apply to extrapartition queues.

In the example application, we could have used transient data instead of temporary storage for our log of changes, and it would have been a natural choice. If we had chosen an intrapartition queue, then we'd still need a transaction to print the log (very similar to the one we defined using temporary storage). We might even have specified in the DCT that we wanted that transaction started every time the number of items logged (the length of the queue) reached 100, or some other limit.

Alternatively, we might have selected an extrapartition queue. In this case we'd be creating a SAM file, which could be printed by a batch program. In fact, if you need to use or create SAM files in a CICS application, you must use transient data.

On the other hand, transient data isn't appropriate for our scratchpad use of temporary storage. Because all the queue names have to be defined beforehand, we could not use the trick of including the account number in the name to get direct access to the scratchpad item we want. Moreover, the fact that an item on the queue can be read only once would have caused us trouble.

3.6 Program control

As we explained earlier, a transaction (task) may execute several programs in the course of completing its work.

Subtopics

3.6.1 Associating programs and transactions

The **installed program definition** contains one entry for every program used by any application in the CICS system. Each entry holds, among other things, three particularly important pieces of information:

1. The language in which the program is written, which CICS needs to know in order to set up its linkages and control blocks properly

2. How many tasks are using the program at the moment

3. Where the program is (in main storage and/or on disk).

In addition to the executable programs, anything that CICS must load in order to respond to a command needs an entry in this installed program definition. For example, a physical map.

The **installed transaction definition** has an entry for every transaction identifier in the system (using "transaction" in the CICS sense of the word). The important information kept about each transaction is the transaction identifier and the name of the *first* program to be executed on behalf of the transaction.

You can see how these two sets of definitions work in concert:

1. The user types in a transaction identifier at the terminal (or the previous transaction determined it).

2. CICS looks up this identifier in the list of installed program definitions.

3. This tells CICS which program to invoke first.

4. CICS looks up this program in the list of installed transaction definitions, finds out where it is, and loads it if it isn't already in main storage.

5. CICS builds the control blocks necessary for this particular combination of transaction and terminal, using information from both sets of definitions. For programs in command-level COBOL, like ours, this includes making a private copy of working storage for this particular execution of the program.

6. CICS passes control to the program, which begins running using the control blocks for this terminal. This program may pass control to any other program in the list of installed program definitions, if necessary, in the course of completing the transaction.

3.6.2 Commands for passing program control

There are two CICS commands for passing control from one program to another. One is the LINK command, which is similar to a CALL statement in COBOL. The other is the XCTL (transfer control) command, which has no COBOL counterpart. When one program links to another, the first program stays in main storage. When the second (linked-to) program finishes and gives up control, the first program resumes at the point after the LINK. The linked-to program is considered to be operating at one logical level lower than the program that does the linking.

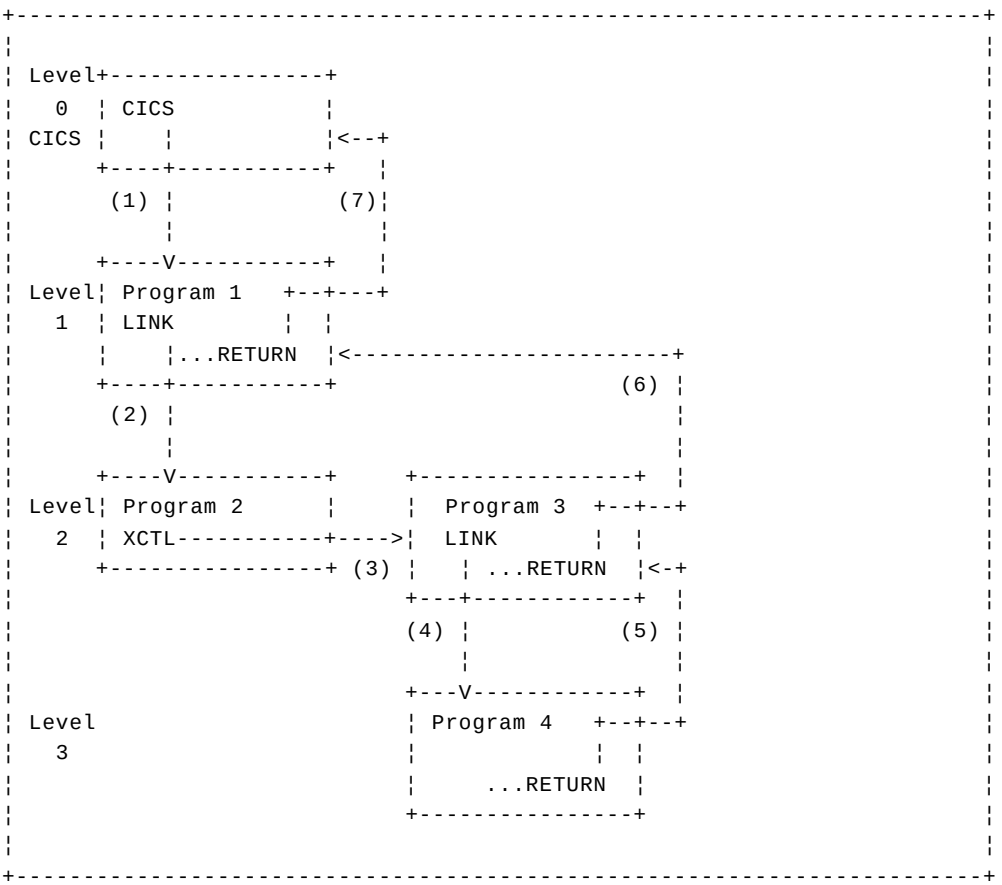


Figure 49. Transferring control between programs (normal returns)

In contrast, when one program transfers control to another, the first program is considered terminated, and the second program operates at the same level as the first. When the second program finishes, control is returned not to the first program, but to whatever program last issued a LINK command.

Some people like to think of CICS itself as the highest program level in this process, with the first program in the transaction as the next level down, and so on. If you look at it from this point of view, CICS links to the program named in the list of installed transaction definitions when it initiates the transaction. When the transaction is complete, this program (or another one operating at the same level) returns control to the next higher level, which happens to be CICS itself. Figure 49 may help.

Subtopics
3.6.2.1 The LINK command
3.6.2.2 The XCTL command
3.6.2.3 The RETURN command
3.6.2.4 The COBOL CALL statement
3.6.2.5 Subroutines revisited

3.6.2.1 The LINK command

The LINK command looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS LINK PROGRAM(pgmname)                                       ¦
¦       COMMAREA (commarea) LENGTH(length) END-EXEC.                      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

pgmname
is the name of the program to which you wish to link. If the name is a literal, enclose it in quotes. Program names can be up to eight characters long.

commarea
is an optional parameter. It is the name of the area containing the data to be passed and/or the area to which results are to be returned. You use it only if you want to pass information to or receive information from the program being linked to.

length
is the length of "commarea." This parameter is required only if COMMAREA is present. Otherwise don't use it. Like the length parameter in other commands, it must be a halfword binary value.

3.6.2.2 The XCTL command

The XCTL command to transfer control is identical to the LINK command except for the command verb itself:

```
EXEC CICS XCTL PROGRAM(pgmname)
    COMMAREA(commarea) LENGTH(length) END-EXEC.
```

3.6.2.3 The RETURN command

The command to return control to the next higher level within a transaction is simply:

```
EXEC CICS RETURN END-EXEC.
```

When the program at the highest level for the transaction (Level 1 in the diagram) returns control to CICS, however, there are two additional options that you can specify:

1. You can say what transaction is to be executed when the next input comes from the same terminal. (This is how we get into pseudoconversational mode.)

2. You can specify data that's to be passed on to that next transaction.

In this case the RETURN command has a slightly different form:

```
EXEC CICS RETURN TRANSID(nextid)
    COMMAREA(commarea) LENGTH(length) END-EXEC.
```

nextid
is the identifier of the next transaction (next transid) to be executed from the terminal associated with the current transaction. This next transaction is the one that gets executed the next time the terminal sends input, regardless of any transaction identifier in that input. (Here's a way of overriding any user's input.) The identifier should be enclosed in quotes if it is a literal. TRANSID is an optional parameter.

commarea
is the name of the data area containing the data to be passed to the next transaction. COMMAREA is also optional.

length
is the length of "commarea." LENGTH is required if COMMAREA is present, and must not be there if COMMAREA was not specified.

3.6.2.4 The COBOL CALL statement

As well as passing control to other programs by means of LINK and XCTL commands, a CICS COBOL program can invoke another program with a COBOL CALL statement. Although there's somewhat less system overhead (in other words, a shorter path length) with this method, there are some considerations that may count against it. For example:

A CALLed program remains in its last-used state after it return control, so a second CALL finds the program in this state. LINK and XCTL commands, on the other hand, always find the "new" program in its initial state.

With static calls, you must link-edit the calling and called program together and present them to CICS as a single unit, with one name and one entry in the list of installed program definitions. This has two consequences:

- It may result in a module that is quite large
- It prevents two programs that call the same program from sharing a copy of the called program.

3.6.2.5 Subroutines revisited

Now, the answer to that problem we met earlier--whether and how to break off a substantial routine. For single-task efficiency, *generally* in-line code is best, PERFORM next, straight CALL third, XCTL next, and LINK last. However, any of the first three choices may make for a very long load unit, and that can impact system behavior and response to other users.

Always use XCTL if it will do, of course, rather than LINK. That's just a program logic issue; you either need control back or you don't. In our example, as you'll see, we've broken our own rule and used a LINK (rather than an XCTL) to the error-handling program. However, we *do* have an excuse ready.... See "Errors within the example application" in topic 3.8.3.

The probability of the code getting used is another issue. If you have a long complex routine for calculating withholding tax for veterans in a payroll system, but you use it only if salary or dependents change and you have hardly any veterans, then by all means put it in a separate routine and LINK to it.

Finally, how about breaking code into two parts? For example, let's take a standard "edit and update if OK" module, like ACCT02 in our application. Figure 50 shows the outline logic.

Figure 50. Outline logic of a standard "edit and update" module.

If the edit and update logic are short, then it makes sense for the whole thing to be one module. If both are rather long, on the other hand, there's a natural break after the edit has been declared okay; the first program does up to point "A" and then there's an XCTL to a second program.

3.6.3 Passing control and data between programs and transactions

Now that we've explained how to pass data from one transaction to another, you may be wondering how the receiving program accesses this data. To show this, let's code a few program control commands for the example application.

In several of the programs, when we meet an error from which we cannot recover, we transfer control to the general-purpose error program, ACCT04. We pass three items of information to ACCT04:

1. The name of the program that passed control (and where the error was detected)

2. The function that failed

3. The return code from the command that failed.

Figure 51 shows how this information looks in program ACCT01's working storage:

```
02  COMMAREA-FOR-ACCT04.
    04  ERR-PGRMID         PIC X(8) VALUE 'ACCT01'.
    04  ERR-FN             PIC X.
    04  ERR-RCODE          PIC X.
    04  ERR-COMMAND        PIC XX.
    04  ERR-RESPONSE       PIC 99.
```

Figure 51. Passing information to the error program

The code in ACCT01 to pass control to ACCT04 is:

```
EXEC CICS LINK PROGRAM('ACCT04')
    COMMAREA(COMMAREA-FOR-ACCT04) LENGTH(14) END-EXEC.
```

Notes:

1. VS COBOL II avoids the need for the programmer to compute LENGTH.

2. We'll discuss the use of LINK rather than XCTL in "Errors within the example application" in topic 3.8.3.

The program receiving control, ACCT04 in this case, defines this same area in its Linkage Section, as shown in Figure 52.

```
LINKAGE SECTION.
01  DFHCOMMAREA.
    02  ERR-PGRMID          PIC X(8).
    02  ERR-CODE.
        04  ERR-FN          PIC X.
        04  ERR-RCODE       PIC X.
    02  ERR-COMMAND         PIC XX.
    02  ERR-RESPONSE        PIC 99.
```

Figure 52. Receiving information in the error program

This area *must* be the first 01 level in the Linkage Section, and you must call it DFHCOMMAREA as shown in the example. You can then use the contents directly, as follows:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦     MOVE ERR-PGRMID TO PGMEO.                                          ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Subtopics
3.6.3.1 Communicating between transactions in the example application

3.6.3.1 Communicating between transactions in the example application

Apart from the LINK to our error-handling program, ACCT04, which is something of a special case, there's no instance of one program linking to another in the example application, and so no instance of return to a higher level within the transaction either.

However, there are several different types of return to CICS. The simplest occurs in program ACCT01, after the user has indicated a wish to exit from the application. No next transid is set, and no data is passed forward to the next transaction. The return command is just:

```
EXEC CICS RETURN END-EXEC.
```

In program ACCT00, in contrast, we need to indicate that the next transaction to be executed from the same terminal is AC01, so the RETURN command is written:

```
EXEC CICS RETURN TRANSID('AC01') END-EXEC.
```

Later, in program ACCT01, after we complete the initial processing of an update request, we need to show that the next transaction to be executed is AC02. Not only that, but we need to pass data to it as well. The data is the request-type code and the account number that came in on the original map. The communications area in Working-Storage where we've stored this information looks like this:

```
04  IN-REQ.
    06  REQC          PIC X VALUE SPACES.
    06  ACCTC         PIC X(5) VALUE SPACES.
    06  PRTRC         PIC X(4) VALUE SPACES.
```

And the code needed is:

```
EXEC CICS RETURN TRANSID('AC02')
    COMMAREA(IN-REQ) LENGTH(6) END-EXEC.
```

When program ACCT02 is invoked, it finds the data passed to it in the same way as a program to which control is passed by means of an XCTL or LINK command. That is, the area is defined in the first 01 level in the Linkage Section, which is named DFHCOMMAREA and has the same format as it did in the passing program. (We happened to use the same names in these programs for the items passed, but that, of course, isn't required.) So program ACCT02 contains the following:

```
LINKAGE SECTION.
01  DFHCOMMAREA.
```

```
¦       02  REQC               PIC X.                                   ¦
¦       02  ACCTC              PIC X(5).                                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

These variables are directly available to the program (the translator generates the code necessary to make this happen).

Incidentally, if you wanted to pass a communications area from, say, program 1 to program 3, you can simply define the area in the linkage section of program 2, even though it's not used in that program, and pass it as COMMAREA on the LINK (or XCTL) to program 3.

3.6.4 Errors on the program control commands

CICS recognizes the following exceptional conditions on program control commands:

INVREQ
means that one of two things happened. Either (1) you specified COMMAREA or LENGTH on a RETURN command in a program that was not at the highest level (that is, a RETURN that would not terminate the transaction by returning control to CICS), or (2) you specified the TRANSID option on a RETURN from a task that had no terminal associated with it. (There are such tasks; see "Starting another task, and other time services" in topic 3.7.) In either form, INVREQ usually means a programming error.

LENGERR
means that the length of the data, specified using the RETURN command with the length option, is outside the valid range of 1 to 32763.

NOTAUTH
means that a resource or command security check has failed. There is a complete list of reasons for such failures in the section on NOTAUTH in the CICS/ESA Application Programming Reference.

PGMIDERR
means that the program to which control was passed, on a LINK or an XCTL command, cannot be found in the list of installed program definitions or isn't in the library, or has been disabled. It corresponds to FILENOTFOUND on a file command, and has similar causes. If it occurs during the testing phase, look for a spelling mismatch; if it occurs once the system has been put into actual use ("in production"), have your systems people check the list of installed program definitions for damage.

3.6.5 Abending a transaction

In addition to the normal return sequences that we've described, there is another command that you use in *abnormal* circumstances. This is the ABEND command. It returns control to CICS directly. Figure 49 showed a normal return from program 4 to program 3, and from program 3 to program 1. If, in contrast, an ABEND command had been issued in program 4, the picture would then be as shown in Figure 53:

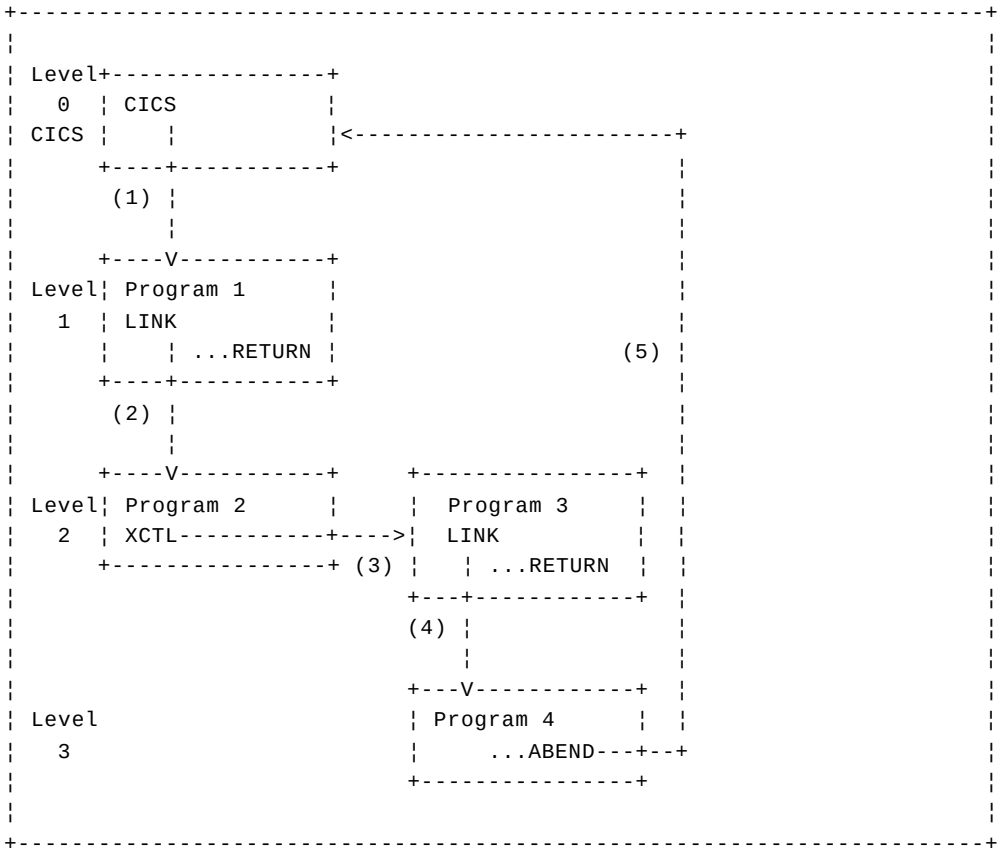


Figure 53. Transferring control between programs (after an abend)

Use the ABEND command when a situation arises that the program cannot handle. This may be a condition beyond control of the program, such as an input/output error on a file, or it may simply be a combination of circumstances that "should not occur" if the program logic is correct. In either case, ABEND is the right command to terminate the transaction. The format is:

```
EXEC CICS ABEND ABCODE(abcode) END-EXEC.
```

abcode
is simply a four-character code identifying the particular ABEND command. It does two jobs: it tells CICS that you want a dump of your transaction, and it identifies the dump. Enclose it in quotes if it is a literal.

In addition to returning control to CICS, the ABEND command has another very important property: it causes CICS to back out all of the changes made by this transaction to recoverable resources (see "Maintaining file integrity" in topic 2.7.3 if you've forgotten what "back out" means).

In our example application, we use this command at the end of program ACCT04, where we send control when we've encountered a situation which

prevents us from continuing the requested transaction. The code is:

```
EXEC CICS ABEND ABCODE('EACC') END-EXEC.
```

Suppose, for example, that program ACCT02 successfully adds a new record to the account file, but meets a "no-space" condition when trying to add the corresponding new record to the index file. The resulting ABEND command issued in program ACCT04 will:

Produce a dump of all the main storage areas related to th transaction

Remove the new record from the account file, so that the two files ar still synchronized with each other, even after the failure

Return control to CICS

3.6.6 Other program control commands

There are two other program control commands that we'll mention here, but not cover in detail.

The LOAD command brings a "program" (any phase or load module in the list of installed program definitions) into main storage but doesn't give it control. This is useful for tables of the type that are assembled and stored in a program library, but that don't contain executable code.

The RELEASE command tells CICS that you've finished using such a "program".

3.7 Starting another task, and other time services

CICS allows one transaction (task) to start another one, as we noted in our discussion about printed output. The usual reason for doing this is the one that arose in our example: the originating task needs access to some facility it does not own, usually a terminal other than the input terminal. In our case, we needed a printer to print the log of account file changes.

There are sometimes other reasons as well. You might want a task to be executed at a particular time, or you might want it to run at a different priority from the original task, for instance.

Subtopics

3.7.1 Starting another task

The command to start another task is:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ EXEC CICS START TRANSID(transid) TERMID(termid)                         ¦
¦     FROM(recarea) LENGTH(length) option END-EXEC.                       ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

transid
is the identifier of the transaction that is to be started. This parameter is required. If the identifier is a literal, enclose it in quotes.

termid
is the identifier of the terminal that must be made available to the task being started. This parameter is optional, and should only be specified if the transaction requires a terminal. Again, if it is a literal, it must be enclosed in quotes.

You may have to get this name from your systems people. It's the name they put in the Terminal Control Table (TCT).

recarea
is the name of the data area that contains data to be passed to the transaction being started. This parameter is optional.

length
is the length of the data being passed (that is, the length of RECAREA), in halfword binary form. The LENGTH parameter is required if FROM is present, but should not be present otherwise.

option
can be either INTERVAL or TIME:

INTERVAL(hhmmss)
tells CICS to start the transaction in hh hours, mm minutes and ss seconds from the current time. The hours may be from 0 to 99, but the minutes and seconds should not exceed 59. To start a task in 40 hours and 10 minutes, you would write "INTERVAL(401000)" in your START command.

TIME(hhmmss)
tells CICS to start the transaction at a specific time, namely "hh:mm:ss." Write the start time in the same format as the interval, using 24-hour military time.

Note: Whereas an INTERVAL always specifies a time in the future (the current time plus the interval specified), the time given in a TIME parameter may be in either the future or the past relative to the time at which the command is executed. The rules that CICS uses are as follows:

If the current time is 060000 (6 a.m.) or later, and the TIME value is less than 6 hours before the current time, CICS assumes that you mean a time in the past, and so the transaction is started as soon as possible, just as if you had specified INTERVAL(0).

If the current time is less than 060000, and the expiration time is less than the current time, then the TIME is also considered to be in the past. Note, however, that the TIME given is never taken to be before midnight of the current day.

```
    Otherwise, CICS assumes that the time is in the future.

    If you specify a time with an hours component greater than 23,
    you are specifying a time on a day following the current one.
     That is: a TIME of 250000 means 1 a.m. on the day following
    the current one, and 490000 means 1 a.m. on the day after
    that.

If you don't specify either INTERVAL or TIME, CICS assumes that
you would like INTERVAL(0), which means right away.
```

3.7.2 Retrieving data passed in the START command

If data is passed in the START command, the transaction that gets started uses the RETRIEVE command to get access to this data. The RETRIEVE command looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   EXEC CICS RETRIEVE INTO(recarea) LENGTH(length)                       ¦
¦       END-EXEC.                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Notice the difference between this RETRIEVE command and the RECEIVE command described in "The RECEIVE MAP command" in topic 3.3.5.1. Both commands may be used to get the initial input to a transaction, but they aren't interchangeable: RECEIVE must be used in transactions that are initiated by input from a terminal, and RETRIEVE must be used in transactions that were STARTed by another transaction.

recarea
is the name of the data area into which the data is to be placed. This parameter is required.

length
is the maximum length of data that can be read into recarea (that is, the length of recarea). LENGTH is also required, and must be a halfword binary value.

3.7.3 Using the START and RETRIEVE commands in the example application

In our example application, program ACCT01 uses the START command when a user asks for a record to be printed:

```
EXEC CICS START TRANSID('AC03') FROM(ACCTDTLO)
    LENGTH(DTL-LNG) TERMID(PRTRC) RESP(RESPONSE) END-EXEC.
```

This START command tells CICS to start transaction AC03 as soon as possible after the printer whose name is in data area PRTRC is available to be its terminal.

Program ACCT03, running on behalf of this transaction, in turn issues the following RETRIEVE command to retrieve the data passed from program ACCT01:

```
EXEC CICS RETRIEVE INTO(ACCTDTLI) LENGTH(TS-LNG) END-EXEC.
```

ACCTDTLO and ACCTDTLI refer to the symbolic map structure, located in Working-Storage in both programs. The map, of course, contains the data read by transaction AC01. This data is to be printed by AC03. DTL-LNG is in the Working-Storage of program ACCT01 and is defined to be

```
PIC S9(4) COMP VALUE +751
```

which happens to be the length of the symbolic map area. TS-LNG has the same definition in the Working-Storage of program ACCT03.

3.7.4 Errors on the START and RETRIEVE commands

A number of different problems may arise in connection with the START and RETRIEVE commands that we've described.

INVTSREQ
means that the CICS system support for temporary storage, which is required for START commands that specify the FROM option, was not present when a RETRIEVE command was issued. This error is an example of the system/application mismatch (category 4) described in "Handling errors and exceptional conditions" in topic 2.9.2.

IOERR
on a RETRIEVE or START command means exactly what it does on a temporary storage command: an input/output error on the temporary storage data set where the data to be passed is stored.

LENGERR
occurs when the length of the data retrieved by a RETRIEVE command exceeds the value specified in the LENGTH parameter for the command. LENGERR usually means an error in the program logic.

NOTFND
on a RETRIEVE command means that the requested data could not be found in temporary storage. If a task issuing a RETRIEVE command was not started by a START command, or if it was started by a START command with no FROM parameter (in other words, no data), this condition will occur. Again, it usually means a programming error.

TERMIDERR
occurs when the terminal specified in the TERMID parameter in a START command cannot be found in the Terminal Control Table. TERMIDERR is like FILENOTFOUND for files and PGMIDERR on Program Control commands. During the test phase it usually indicates a problem in the program logic; on a production system, it usually means that something has happened to the TCT.

TRANSIDERR
means that the transaction identifier specified in a START command cannot be found in the list of installed transaction definitions. Like TERMIDERR, it usually means a programming error during the development of an application, or table damage if it occurs on a production system.

3.7.5 Other time services

CICS provides a number of other time services, as well as some extra bits and pieces on the START and RETRIEVE commands. Among other things, a transaction in execution can:

Synchronize its operations with those of other tasks. Three differen commands are provided for this purpose:

- The DELAY command suspends the processing of the issuing task until some specified time or for a specified interval.
- The POST command requests that the issuing task be notified when a particular interval of time has elapsed or when some event has occurred.
- The WAIT command suspends the issuing task until some specified event occurs.

Cancel the request issued in a previous START command, or in a POS command, through the use of the CANCEL command.

Ask for the time and date to be updated in the EIB (through the use o the ASKTIME command).

Assign a name to the data to be passed from the originating task t the started task, through the use of the REQID option on the START and RETRIEVE commands.

Queue up multiple items of data for a single task to be started through the use of the QUEUE option on the START command.

We don't use any of these in our example application, but at least you now know they exist.

3.8 Errors and exceptional conditions

Throughout the previous sections, we've cited ways in which CICS commands may produce results other than those you intended (what CICS cheerfully calls "exceptional conditions"). These are passed back by the CICS Exec interface program to your application. By looking at the condition raised, you'll be able to tell what failed, and possibly why it failed.

Commands are checked for validity as far as possible by the CICS translator. If errors are detected at translate time the translator issues a suitable diagnostic and gives a return code greater than 4. Such commands are said to be "syntactically invalid." Programs containing syntactically invalid commands should never be executed and we'll not discuss them any further.

Commands which are syntactically valid may nevertheless fail to execute successfully for a variety of reasons. (And how!)

If a CICS command executes successfully, the command is said to have a normal response. Unless you take special action, CICS will check that a command executes normally. If it doesn't, CICS will take some appropriate action and will not, in general, return control to the application. The special action is called "system default action" and is usually to abend ("abnormally end") the transaction. As we pointed out in "Handling errors and exceptional conditions" in topic 2.9.2, this is almost never what you want in these situations.

For many applications the CICS system default action will be inappropriate and you'll need to write some special code to be invoked in the event of non-normal response. What sort of code?

Basically, you have three choices when an exceptional condition arises:

1. Let the program continue

2. Pass control to a specified label

3. Do nothing, and rely on the system default action.

CICS provides you with a number of programming options applicable to each choice, and the CICS/ESA Application Programming Guide gives you full details of all these options. To save you reading through the whole of the relevant topic in that book, however (although you'll probably need to study that book when you come to write your own application programs), here's the information that specifically relates to the example application programs in **this** book.

When you look through the example COBOL programs described in this book to find out what they do when an exceptional condition arises, you'll find that only the first two choices have been used: to let the program continue, or to pass control to a specified label.

Subtopics

3.8.1 Letting the program continue

Letting the program continue means allowing control to return from CICS to the next instruction in the program immediately following the one that has failed. At the same time, CICS sets a return code in **EIBRESP** so that you can test for particular conditions right after each command. (This approach is particularly useful when you are structuring your code, incidentally.)

CICS makes it very easy to test the **RESP** value by supplying a built-in function called **DFHRESP** for you to use. So you can execute each CICS command and then immediately find out what the **RESP** value was for it. If the **RESP** value is **NORMAL**, this means the command worked. (Even if the value isn't **NORMAL**, this may be both expected and acceptable.)

And that's not all. Your code can also examine **RESP** values by their symbolic names (for example, **DFHRESP(LENGERR)** when testing for a condition by the symbolic name of **LENGERR**). This avoids having to mess around with hexadecimal values.

Let's have a look at a section of the ACCT02 code where we've used the **RESP** option:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 110  *                                                                 ¦
¦ 111  *    GET INPUT AND BUILD NEW RECORD.                              ¦
¦ 112       EXEC CICS RECEIVE MAP('ACCTDTL') MAPSET('ACCTSET')           ¦
¦ 113                 RESP(RESPONSE) END-EXEC.                           ¦
¦ 114       IF RESPONSE = DFHRESP(MAPFAIL) GO TO NO-MAP.                 ¦
¦ 115       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO NO-GOOD.             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

A **MAPFAIL** condition can be raised on this command, as indeed can several other conditions. So we've specified the **RESP** option to find out, after execution, what condition has been raised on the **RECEIVE MAP**. The program can then check the value of **RESP** in the **RESPONSE** variable (defined earlier in the program) to see if any errors have occurred.

What conditions can we provide for? Well, there are six exceptional conditions that we've chosen to deal with in this way:

1. A **no input** (**MAPFAIL**) condition when we read the input map.

   This generally results from a keying error, and we would certainly annoy the user if we allowed CICS to abend the transaction for this comparatively minor slip. Therefore, we want to send a message to let the user correct the input instead.

   (In the section of the ACCT02 code given above, we start by looking explicitly for the **MAPFAIL** condition, because this condition can occur without there being any serious error (if, for example, the user presses CLEAR at this point in the application). If the **MAPFAIL** condition is raised, control will go to **NO-MAP**. If there is some other sort of error (any **NOT NORMAL** condition), control will go to **NO-GOOD**.)

2. A **record not found** (**NOTFND**) condition when we read the index file for a customer name entered by the user.

   This situation isn't an error; it simply means that there are no customers with that particular name, and so we'll inform the user.

3. A **record not found** (**NOTFND**) condition when we try to read the account

file record named in the input.

NOTFND in this instance may actually be correct (if the user is trying to add a record) and is at worst an error in the account number, to be treated like any other input error.

4. An **end of file** (**ENDFILE**) condition when we're browsing through the index file looking for all the matching records on a name search.

   This isn't an error either, just a sign that we've run out of candidate names.

5. A **no such entry** (**QIDERR**) response to reading the scratchpad.

   This is the expected result when we read temporary storage to see if anyone else is updating the record we want to update. It means no one is using "our" record.

6. A **terminal id error** (**TERMIDERR**) when we start the AC03 transaction to print a record.

   This condition means that the user entered a printer name that is unknown to CICS. We'll treat it like any other type of input error.

3.8.2 Passing control to a specified label

There are two ways you can do this:

HANDLE CONDITION condition(label) command

where **condition** is the name of the condition you want to handle.
HANDLE CONDITION ERROR(label) command.

The only **HANDLE CONDITION** command that we use in the example COBOL program is the **HANDLE CONDITION ERROR(label)** command, however, so that's the one we'll be concentrating on.

The **HANDLE CONDITION** command tells CICS where to go when an exceptional condition occurs. It looks like this:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ EXEC CICS HANDLE CONDITION condition(label)                             ¦
¦     condition(label) ...                                                ¦
¦     condition  condition ... END-EXEC.                                   ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

condition
is the CICS name of the unusual condition for which you wish to establish special processing (or return to default processing, as explained below). It can be any of the exceptions that we've described in this part: **IOERR, LENGERR, NOTFND**, and so on, and you can name up to 16 conditions in one **HANDLE CONDITION** command.

label
is the name of the paragraph in your program to which CICS is to pass control when the condition occurs. The paragraph name following a condition is optional; if you specify it, you are saying that you want to deal with the condition in question with code in the program. If you omit it, you are saying that you want CICS to use its default procedure for the condition (or, more likely, that you want to reestablish the CICS default action after you had specified other handling for the condition earlier).

For the handling code to take effect, a **HANDLE CONDITION** command must be issued *before* you execute any command on which one of the conditions you list might arise. Nothing visible happens when you execute the **HANDLE CONDITION** command, although CICS updates its table of conditions, of course. The effects are seen later, when a command is executed that produces one of the exceptional conditions now covered by the **HANDLE CONDITION**.

The **ERROR** condition in this command covers all exceptional conditions, except:

Those cited by name in this command or another **HANDLE CONDITION** command executed previously in the program, and

Those for which the CICS default action is *not* abnormal termination of the program.

We've specified **ERROR** here because there are many other exceptional conditions that can arise on the commands that we'll issue in this program, besides those listed above. (Figure 54 in topic 3.8.3 shows which conditions apply to each command.) These conditions are all serious enough to prevent successful completion of the transaction, and we don't want to deal with each one individually, but we do want our program to regain control long enough to send the user a message saying what happened

and what to do next.

Subtopics
3.8.2.1 Changing the HANDLE CONDITION "destinations"

3.8.2.1 Changing the HANDLE CONDITION "destinations"

Making control go to different places on different occasions is no problem if you use the **RESP** option. It can also be managed, albeit somewhat more awkwardly, with **HANDLE CONDITION** commands. With these, if we wanted control to go to different places on different occasions, we'd have to do one of two things:

Issue a single **HANDLE CONDITION** command and test which file was involved at, for example, the beginning of the paragraph named to deal with the **NOTFND** condition. The **EIBDS** field in the EIB tells which data set was used most recently in a command and can be used for such a test.

Issue a **HANDLE CONDITION** command appropriate for a **NOTFND** on the first command issued that may encounter it (in our case, the **READ** of the index file) and then, before the next command on which we want to specify a different paragraph name for that same condition, issue another **HANDLE CONDITION** command.

3.8.3 Errors within the example application

To summarize, we've designed our error handling as follows:

1. Using **RESP**, we specifically deal with exceptional conditions if they are expected and can be dealt with in the application's logic.

   For example, we expect a **NOTFND** condition when the user tries to add a new customer account--we read the account record just to make sure that it's not already in the file.

2. We use a **HANDLE CONDITION ERROR** (whatever) command as a catch-all to deal with **unexpected** exceptional conditions. We've put this command near the start of ACCT01, ACCT02, and ACCT03.

3. If and when something unexpected happens, CICS passes control to our error routine (either as a result of an **IF RESPONSE NOT = DFHRESP(NORMAL) GO TO...** test or to the paragraph named in the **HANDLE CONDITION ERROR** "catch-all" code). The first thing the error routine must do is issue **another HANDLE CONDITION ERROR**, but *without a label*, to prevent a possible error handling loop.

   Next, the error routine gives control to ACCT04, passing the first byte of **EIBFN** and **EIBRCODE**. We use a **LINK**, rather than an **XCTL**, so that we'll get the failing program and its Working-Storage in the transaction dump. (If we use **XCTL**, CICS releases the storage associated with the program we're "XCTLing" from.)

   ACCT04 finds out what's wrong, builds and displays an appropriate error screen, and finally issues an **ABEND** command with a code of **EACC**, telling CICS to produce the transaction dump.

   So the dump will contain a *predictable sequence of actions* between the occurrence of the actual error and ACCT04's last act. We'll show you how to follow this sequence of events in "A session with EDF" in topic 5.1.3.1.7.

   There *is* a way of using **XCTL** rather than **LINK** when transferring control to our error-handling program. It's also a perfectly reasonable alternative: put an **EXEC DUMP** command immediately before each appropriate **XCTL** command in programs ACCT01, ACCT02, and ACCT03.

   Of course, you'd probably want to remove these **DUMP** commands before putting the system into production.

   Our solution manages with just one **ABEND** command (a side effect of which is the transaction dump we want) but has to use a **LINK** instead of the more efficient **XCTL**.

Because of our "catch-all" **HANDLE CONDITION ERROR** command, we should be protected against the results of an unexpected CICS abend.

Figure 54 lists which unusual conditions may occur for the commands and options covered in this Primer. Note that other exceptions may arise if you use options or facilities of CICS beyond the scope of this Primer.

Command	**Conditions**
SEND MAP	INVMPSZ
SEND CONTROL	(none)
RECEIVE MAP	INVMPSZ, MAPFAIL
HANDLE AID	(none)
READ	FILENOTFOUND, ILLOGIC, INVREQ, IOERR,
	LENGERR, NOTFND, NOTOPEN

REWRITE, WRITE	FILENOTFOUND, DUPREC, ILLOGIC, IOERR, INVREQ, LENGERR, NOSPACE, NOTOPEN
DELETE, STARTBR	FILENOTFOUND, ILLOGIC, INVREQ, IOERR, NOTFND, NOTOPEN
READNEXT	FILENOTFOUND, ENDFILE, ILLOGIC, IOERR, INVREQ, LENGERR, NOTOPEN
ENDBR	FILENOTFOUND, ILLOGIC, INVREQ, NOTOPEN
WRITEQ TS	INVREQ, IOERR, ITEMERR, QIDERR, NOSPACE **(See below.)**
READQ TS	IOERR, ITEMERR, LENGERR, QIDERR
DELETEQ TS	QIDERR
LINK, XCTL	PGMIDERR
RETURN	INVREQ
ABEND	(none)
START	INVREQ, IOERR, TERMIDERR, TRANSIDERR
RETRIEVE	INVREQ, INVTSREQ, IOERR, LENGERR, NOTFND
HANDLE CONDITION	(none)

Figure 54. The exception conditions for the Primer's subset of CICS commands

Of all these conditions, **NOSPACE** on the **WRITEQ TS** command is the only one for which CICS default processing is *not* to terminate the transaction. When this condition is encountered, the default processing is for CICS to suspend the transaction until space becomes available. (The theory is that since many transactions use temporary storage, others will eventually give up enough space for this one to continue.)

3.8.4 Other facilities for exceptional conditions

As mentioned at the start of the topic, CICS provides other means to control the processing sequence when exception conditions occur:

- There's a command to intercept control directly when CICS determine that a transaction should be terminated abnormally (the **HANDLE ABEND** command). This is rather a last-ditch method in most cases.

- The set of paragraph names specified to deal with exceptiona conditions in a program can be suspended temporarily (the **PUSH HANDLE** command), replaced by others (with **HANDLE CONDITION** commands) and then restored (with a **POP HANDLE** command). This is useful for closed subroutines within a program, especially if they contain error-processing code.

- **PUSH HANDLE** and **POP HANDLE** apply to the paragraph names specified on **HANDLE AID** and **HANDLE ABEND** conditions, as well as those specified with **HANDLE CONDITION**.

These facilities are all described in the CICS/ESA Application Programming Guide.

4.0 The COBOL code of our example application

+--- **This part of the Primer lists:** ----------+
¦ ¦ **ACCT00**--menu display ¦
¦ ¦ **ACCT01**--initial request analysis ¦
¦ ¦ **ACCT02**--update processing ¦
¦ ¦ **ACCT03**--requests for printing ¦
¦ ¦ **ACCT04**--error processing ¦
¦ ¦ Other items. ¦
+---+

Subtopics
4.1 Program ACCT00: menu display
4.2 Program ACCT01: initial request analysis
4.3 Program ACCT02: update processing
4.4 Program ACCT03: requests for printing
4.5 Program ACCT04: error processing

4.1 Program ACCT00: menu display

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 001   IDENTIFICATION DIVISION.                                         ¦
¦ 002   PROGRAM-ID. ACCT00.                                              ¦
¦ 003  *REMARKS. THIS PROGRAM IS THE FIRST INVOKED BY THE 'ACCT'         ¦
¦ 004  *         TRANSACTION.  IT DISPLAYS A MENU SCREEN FOR THE ONLINE  ¦
¦ 005  *         ACCOUNT FILE APPLICATION, WHICH PROMPTS THE USER FOR    ¦
¦ 006  *         INPUT.  TRANSACTION 'AC01' IS INVOKED WHEN THAT INPUT   ¦
¦ 007  *         IS RECEIVED.                                            ¦
¦ 008   ENVIRONMENT DIVISION.                                           ¦
¦ 009   DATA DIVISION.                                                  ¦
¦ 010   PROCEDURE DIVISION.                                            ¦
¦ 011   INITIAL-MAP.                                                    ¦
¦ 012       EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET') MAPONLY      ¦
¦ 013            ERASE FREEKB END-EXEC.                                 ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 12 through 13 (INITIAL-MAP): This command sends the menu map to the input terminal. We use the MAPONLY option because the map itself is the only thing being sent; we have no variable data from the program to merge into it. We also specify the ERASE option, to clear the screen of the input and anything else left over from previous activity.

FREEKB unlocks the keyboard for the user's next input. We're specifying it just for documentation purposes.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 014       EXEC CICS RETURN TRANSID('AC01') END-EXEC.                   ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 14: After sending the map, we return control to CICS. In doing so, we specify that the next transaction to be executed from the terminal that sent this one should be AC01, which analyzes inputs sent through the menu map.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 015       GOBACK.                                                      ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 15: This COBOL statement is never executed, because control does not return to a CICS program after it executes a RETURN command. However, the translator expands all CICS commands to COBOL CALL statements, and although CICS does not return to the program from this call, the compiler does expect control to be returned. Consequently, you need this logical "end of program" to keep the compiler happy.

4.2 Program ACCT01: initial request analysis

```
001   IDENTIFICATION DIVISION.
002   PROGRAM-ID. ACCT01.
003  *REMARKS. THIS PROGRAM IS THE FIRST INVOKED BY THE 'AC01'
004  *         TRANSACTION. IT ANALYZES ALL REQUESTS, AND COMPLETES
005  *         THOSE FOR NAME INQUIRIES AND RECORD DISPLAYS.  FOR
006  *         UPDATE TRANSACTIONS, IT SENDS THE APPROPRIATE DATA ENTRY
007  *         SCREEN AND SETS THE NEXT TRANSACTION IDENTIFIER TO
008  *         'AC02', WHICH COMPLETES THE UPDATE OPERATION. FOR PRINT
009  *         REQUESTS, IT STARTS TRANSACTION 'AC03' TO DO THE ACTUAL
010  *         PRINTING.
011   ENVIRONMENT DIVISION.
012   DATA DIVISION.
013   WORKING STORAGE SECTION.
014   01  MISC.
015       02  RESPONSE              PIC S9(8) COMP.
016       02  MSG-NO                PIC S9(4) COMP VALUE +0.
017       02  ACCT-LNG              PIC S9(4) COMP VALUE +383.
018       02  ACIX-LNG              PIC S9(4) COMP VALUE +63.
019       02  DTL-LNG               PIC S9(4) COMP VALUE +751.
020       02  STARS                 PIC X(12) VALUE '************'.
021       02  USE-QID.
022           04  USE-QID1          PIC X(3) VALUE 'AC0'.
023           04  USE-QID2          PIC X(5).
024       02  USE-REC.
025           04  USE-TERM          PIC X(4) VALUE SPACES.
026           04  USE-TIME          PIC S9(7) COMP-3.
027           04  USE-DATE          PIC S9(7) COMP-3.
028       02  USE-LIMIT             PIC S9(7) COMP-3 VALUE +1000.
029       02  USE-ITEM              PIC S9(4) COMP VALUE +1.
030       02  USE-LNG               PIC S9(4) COMP VALUE +12.
031       02  IN-AREA.
032           04  IN-TYPE           PIC X VALUE 'R'.
033           04  IN-REQ.
034               06  REQC          PIC X VALUE SPACES.
035               06  ACCTC         PIC X(5) VALUE SPACES.
036               06  PRTRC         PIC X(4) VALUE SPACES.
037           04  IN-NAMES.
038               06  SNAMEC        PIC X(18) VALUE SPACES.
039               06  FNAMEC        PIC X(12) VALUE SPACES.
040       02  COMMAREA-FOR-ACCT04.
```

```
041           04  ERR-PGRMID        PIC X(8) VALUE 'ACCT01'.
042           04  ERR-FN            PIC X.
043           04  ERR-RCODE         PIC X.
044           04  ERR-COMMAND       PIC XX.
045           04  ERR-RESP          PIC 99.
046       02  LINE-CNT              PIC S9(4) COMP VALUE +0.
047       02  MAX-LINES             PIC S9(4) COMP VALUE +6.
048       02  IX                    PIC S9(4) COMP.
049       02  SRCH-CTRL.
050           04  FILLER            PIC X VALUE 'S'.
051           04  BRKEY.
052               06  BRKEY-SNAME   PIC X(12).
053               06  BRKEY-ACCT    PIC X(5).
054           04  MAX-SNAME         PIC X(12).
055           04  MAX-FNAME         PIC X(7).
056           04  MIN-FNAME         PIC X(7).
057       02  SUM-LINE.
```

```
¦ 058           04  ACCTDO             PIC X(5).                        ¦
¦ 059           04  FILLER             PIC X(3) VALUE SPACES.           ¦
¦ 060           04  SNAMEDO            PIC X(12).                       ¦
¦ 061           04  FILLER             PIC X(2) VALUE SPACES.           ¦
¦ 062           04  FNAMEDO            PIC X(7).                        ¦
¦ 063           04  FILLER             PIC X(2) VALUE SPACES.           ¦
¦ 064           04  MIDO               PIC X(1).                        ¦
¦ 065           04  FILLER             PIC X(2) VALUE SPACES.           ¦
¦ 066           04  TTLDO              PIC X(4).                        ¦
¦ 067           04  FILLER             PIC X(2) VALUE SPACES.           ¦
¦ 068           04  ADDR1DO            PIC X(24).                       ¦
¦ 069           04  FILLER             PIC X(2) VALUE SPACES.           ¦
¦ 070           04  STATDO             PIC X(2).                        ¦
¦ 071           04  FILLER             PIC X(3) VALUE SPACES.           ¦
¦ 072           04  LIMITDO            PIC X(8).                        ¦
¦ 073       02  PAY-LINE.                                                ¦
¦ 074           04  BAL                PIC X(8).                        ¦
¦ 075           04  FILLER             PIC X(6) VALUE SPACES.           ¦
¦ 076           04  BMO                PIC 9(2).                        ¦
¦ 077           04  FILLER             PIC X VALUE '/'.                 ¦
¦ 078           04  BDAY               PIC 9(2).                        ¦
¦ 079           04  FILLER             PIC X VALUE '/'.                 ¦
¦ 080           04  BYR                PIC 9(2).                        ¦
¦ 081           04  FILLER             PIC X(4) VALUE SPACES.           ¦
¦ 082           04  BAMT               PIC X(8).                        ¦
¦ 083           04  FILLER             PIC X(7) VALUE SPACES.           ¦
¦ 084           04  PMO                PIC 9(2).                        ¦
¦ 085           04  FILLER             PIC X VALUE '/'.                 ¦
¦ 086           04  PDAY               PIC 9(2).                        ¦
¦ 087           04  FILLER             PIC X VALUE '/'.                 ¦
¦ 088           04  PYR                PIC 9(2).                        ¦
¦ 089           04  FILLER             PIC X(4) VALUE SPACES.           ¦
¦ 090           04  PAMT               PIC X(8).                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 13 through 90: These lines are the working storage area of the program. Individual variables will be explained in the comments below, as they are used.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 091       COPY DFHBMSCA.                                               ¦
¦ 092       COPY DFHAID.                                                 ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 91 through 92: These two lines bring in the definitions of the attribute bytes and attention identifiers that CICS provides for COBOL programmers. See "Symbolic description maps (DSECT structures)" in topic 3.3.1 and "Finding out what key the operator pressed" in topic 3.3.6.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 093   01  ACCTREC. COPY ACCTREC.                                       ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 93: This line fetches the record format for the account file, copying it from the library in which it was stored.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 094   01  ACIXREC. COPY ACIXREC.                                       ¦
```

```
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 94: Similarly, this line fetches the record format for the index file, copying it from the library in which it was stored.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 095       COPY ACCTSET.                                                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 95: This line copies in the symbolic map structure (DSECT) for the BMS maps for the application.

We created the library member that we're copying here by assembling the map with **TYPE=DSECT** specified. (See "Symbolic description maps (DSECT structures)" in topic 3.3.1.)

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 096   01  MSG-LIST.                                                    ¦
¦ 097       02  FILLER                PIC X(60) VALUE                   ¦
¦ 098           'NAMES MUST BE ALPHABETIC, AND SURNAME IS REQUIRED.'.    ¦
¦ 099       02  FILLER                PIC X(60) VALUE                   ¦
¦ 100           'ENTER SOME INPUT AND PRESS "CLEAR" OR "ENTER".'.        ¦
¦ 101       02  FILLER                PIC X(60) VALUE                   ¦
¦ 102       'REQUEST TYPE REQUIRED; MUST BE "D", "P", "A", "M" OR "X".'. ¦
¦ 103       02  FILLER                PIC X(60) VALUE                   ¦
¦ 104           'PRINTER NAME REQUIRED ON PRINT REQUESTS'.               ¦
¦ 105       02  FILLER                PIC X(60) VALUE                   ¦
¦ 106           'ACCOUNT NUMBER REQUIRED (BETWEEN 10000 AND 79999)'.     ¦
¦ 107       02  FILLER                PIC X(60) VALUE                   ¦
¦ 108           'ACCOUNT NO. MUST BE NUMERIC AND FROM 10000 TO 79999'.   ¦
¦ 109       02  FILLER                PIC X(60) VALUE                   ¦
¦ 110           'NO NAMES ON FILE MATCHING YOUR REQUEST'.                ¦
¦ 111       02  FILLER                PIC X(60) VALUE                   ¦
¦ 112           'ENTER EITHER NAME OR A REQUEST TYPE AND ACCOUNT NUMBER'.¦
¦ 113       02  FILLER                PIC X(60) VALUE                   ¦
¦ 114           'THIS ACCOUNT NUMBER ALREADY EXISTS'.                    ¦
¦ 115       02  FILLER                PIC X(60) VALUE                   ¦
¦ 116           'NO RECORD OF THIS ACCOUNT NUMBER'.                      ¦
¦ 117       02  FILLER                PIC X(47) VALUE                   ¦
¦ 118           'THIS ACCOUNT NUMBER ALREADY IN USE AT TERMINAL '.       ¦
¦ 119       02  MSG-TERM              PIC X(13).                        ¦
¦ 120       02  FILLER                PIC X(60) VALUE                   ¦
¦ 121           'PRINT REQUEST SCHEDULED'.                               ¦
¦ 122       02  FILLER                PIC X(60) VALUE                   ¦
¦ 123           'PRINTER NAME NOT RECOGNIZED'.                           ¦
¦ 124       02  FILLER                PIC X(60) VALUE                   ¦
¦ 125       'INVALID KEY PRESSED - USE ONLY "CLEAR" OR "ENTER" KEY'.     ¦
¦ 126       02  FILLER                PIC X(60) VALUE                   ¦
¦ 127          'THERE ARE MORE MATCHING NAMES. PRESS PA2 TO CONTINUE.'.  ¦
¦ 128   01  FILLER REDEFINES MSG-LIST.                                   ¦
¦ 129       02  MSG-TEXT              PIC X(60) OCCURS 15.              ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 96 through 129: These lines are more of the working storage of the program, in this case constants (messages) used by the program.

Note: All of these message constants will be physically repeated for each concurrent task, because each task gets its own copy of working storage. One way to avoid this would be to have the message constants as literals in the program.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 130   LINKAGE SECTION.                                              ¦
¦ 131   01  DFHCOMMAREA.                                             ¦
¦ 132       02  SRCH-COMM.                                           ¦
¦ 133           03 IN-COMM.                                         ¦
¦ 134               04 CTYPE PIC X.                                 ¦
¦ 135                  88 REPEAT-MAP VALUE 'R'.                     ¦
¦ 136                  88 SEARCH-CONTINUE VALUE 'S'.                ¦
¦ 137               04 FILLER PIC X(40).                            ¦
¦ 138           03 FILLER PIC X(3).                                 ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Lines 130 through 138: The structure defined here and named **DFHCOMMAREA** describes the data passed to this program by means of **COMMAREA** (See "Saving data and communicating between transactions" in topic 3.5 and "Passing control and data between programs and transactions" in topic 3.6.3.) It must have this particular name and it must be the first "01" level in the Linkage Section.

The two level 88 items are value clauses specifying the initial contents of data items **REPEAT-MAP** and **SEARCH-CONTINUE**. (The value of **SEARCH-CONTINUE** is eventually tested in line 158, for example.)

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 139  *                                                              ¦
¦ 140   PROCEDURE DIVISION.                                           ¦
¦ 141  *                                                              ¦
¦ 142  *                                                              ¦
¦ 143  *    INITIALIZE.                                               ¦
¦ 144  *    TRAP ANY UNEXPECTED ERRORS.                               ¦
¦ 145       EXEC CICS HANDLE CONDITION ERROR(OTHER-ERRORS) END-EXEC.   ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Lines 139 through 145: These statements tell CICS where control should go if unexpected errors are encountered. Specific conditions that might result from user errors and conditions that CICS regards as unusual, but that the program expects, are handled with explicit code later in the program by the **RESP** option. Examples of these are **MAPFAIL, NOTFND, ENDFILE, TERMIDERR** and **QIDERR.** The program does not attempt to recover from other unusual conditions, and therefore all of these are passed, by means of this **HANDLE CONDITION ERROR** command, to a single point in the program (**OTHER-ERRORS** at Line 408), from which control is sent to an error program. This program in turn sends a message to the user and abends the task.

Nothing happens, as the result of executing this **HANDLE CONDITION ERROR** command, that immediately affects the flow of the program or the data available to it. Instead, this command causes CICS to record information for processing exceptional conditions in this particular program, should they occur subsequently.

You'll be able to find more detailed guidance in the CICS/ESA Application Programming Guide.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 146  *                                                              ¦
¦ 147       MOVE LOW-VALUES TO ACCTMNUI, ACCTDTLI.                    ¦
¦ 148       MOVE SPACES TO SUMLNMO (1) SUMLNMO (2) SUMLNMO (3)        ¦
¦ 149                      SUMLNMO (4) SUMLNMO (5) SUMLNMO (6).       ¦
```

```
¦ 150       MOVE SPACES TO MSGMO.                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 146 through 150: Both symbolic map structures are set to nulls. We do this to the menu map because when you issue a **RECEIVE MAP** command, BMS sets the length and flag subfields for every field in the map, but it does not set the input subfields unless the corresponding map field was transmitted from the screen. (As we explained earlier, transmission occurs if either the user changes the field or the modified-data tag was set on in the program or the map.) Therefore if you do not clear the symbolic map before you receive, you cannot distinguish between input data and data left over from a previous transaction, unless you check the length subfield first.

The reason for clearing the output (detail) map is to prevent any attribute or length subfields being unintentionally overwritten, and to avoid sending unintended data to a map field not otherwise set by the program. Specifying **ACTION CLEAR** in the linkage editor clears all parts of working storage that aren't otherwise initialized to nulls, and therefore has the same effect on **ACCTMNUI** and **ACCTDTLI** as these two moves.

(Failure either to move nulls (X'00', **LOW-VALUES**) in or to specify **ACTION CLEAR** is a common cause of BMS trouble. The OS equivalent of **ACTION CLEAR** occurs automatically.)

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 151  *                                                                ¦
¦ 152  *    CHECK BASIC REQUEST TYPE.                                   ¦
¦ 153       IF EIBAID = DFHCLEAR                                        ¦
¦ 154           IF EIBCALEN = 0,                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 151 through 154: This begins the analysis of what the user wants to do in this transaction. We first test for the **CLEAR** key. In our particular application, we've defined its use to mean either:

> The user wants to escape from the application
>
> The user has finished (or given up trying to finish) a request starte in a previous transaction, and now wants a fresh menu screen to enter a new request.

If the user has pressed the **CLEAR** key, we've now got to find out which of these situations applies. That, in turn, depends on what the user did last at the terminal.

Whenever a user enters a request that cannot be completed in the course of the current transaction, this program saves information about the request in **COMMAREA** to pass to the next transaction at the same terminal. (Our reason for writing the program in this way is discussed in connection with Line 163.) In the case of input errors or of a name search whose results will not fit on a single screen, the next transaction is the same as this one, and this same program processes it. For an update request, the next transaction is **AC02,** and so this information is passed to program **ACCT02** with an **EXEC CICS RETURN** command. (See Lines 237, 365, and 396 for the commands that pass on this information.)

Therefore, we can distinguish between the two uses of the **CLEAR** key listed above by finding out whether the previous transaction from this terminal has passed data to this one through **COMMAREA.** There is a **COMMAREA** if it has a positive length (**EIBCALEN** greater than zero).

```
155               EXEC CICS SEND CONTROL FREEKB END-EXEC.
156               EXEC CICS RETURN END-EXEC.
```

Lines 155 through 156: If the user wants to exit the application, we use a **SEND CONTROL** command with the **FREEKB** option to open the keyboard. (The keyboard locks on every send operation, including **CLEAR.** Usually, the program needs to write something back to the terminal, in which case the **FREEKB** option can be included on the **SEND MAP** command. However, here we do not want to write anything, and therefore if we fail to unlock the keyboard by this other means, the user will have to use the **RESET** key before he or she can make the next entry. This isn't a disaster, but it is annoying, especially because the user will get no other notification that he or she has left control of the application and the terminal is free for the next transaction.)

```
157           ELSE GO TO NEW-MENU.
```

Line 157: If a request was in progress, however, we pass control to code at **NEW-MENU** (Line 402) which puts out a fresh menu screen (and returns to CICS with no **COMMAREA,** indicating no request in progress).

```
158       IF EIBAID = DFHPA2 AND EIBCALEN > 0 AND SEARCH-CONTINUE
```

Line 158: After testing for the **CLEAR** key, the next possibility we test for is that the previous request was for a name search on which not all the eligible names could fit on the screen, and that the user has asked to see more names. If this is the case, the user will have sent the request by pressing **PA2,** and the previous execution of the program will have saved information in **COMMAREA** for the current execution. **COMMAREA** will contain:

1. An indicator that the last request was a name search (variable **CTYPE,** set to **S** by Line 50)

2. The name limits required to control the search

3. The key for the next eligible name.

```
159           MOVE SRCH-COMM TO SRCH-CTRL, GO TO SRCH-RESUME.
```

Line 159: If all requirements for continuing a name search are met, the name limits and starting key are restored from **COMMAREA,** and the search resumes at SRCH-RESUME (Line 203).

```
160       MOVE DFHBMDAR TO SUMTTLMA.
161       IF EIBAID NOT = DFHENTER MOVE 14 TO MSG-NO
162           GO TO MENU-RESEND.
```

```
+------------------------------------------------------------------------+
```

Lines 160 through 162: The **SUMTTLMA** field will not be displayed because the attribute byte has now been set to "dark".

If the **ENTER** key has not been pressed, **MSG-NO** is set to 14 (this represents the message in **MSG-LIST.**

'INVALID KEY PRESSED - USE ONLY "CLEAR" OR "ENTER" KEY'

Control is passed to **MENU-RESEND** to redisplay the menu.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 163       IF EIBCALEN > 0 AND REPEAT-MAP, MOVE IN-COMM TO IN-AREA.     ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 163: The next step in determining what the user wants to do is to look at **COMMAREA,** to see if this transaction is a resubmission of a previous transaction on which the user made an error. As we explained in connection with Lines 152-153, **COMMAREA** will be present if a previous request was in progress. This request might have been one on which the user entered bad data, or it might have been a name search for which there were more matches than would fit on the screen. In order to distinguish between these two cases, the program saves a variable in **COMMAREA** which is initialized to S if the request was a name search and R otherwise (see Lines 32 and 50). This variable becomes **CTYPE** (Line 138) on the subsequent execution of the transaction.

If the previous transaction was *other than* a name search and **COMMAREA** is present, then we can assume that this is a resubmission after an error. We therefore restore information saved from the previous execution to an area in which we save the input, about which more in a moment. If **COMMAREA** isn't present, on the other hand, this transaction isn't a resubmission of a previous request. The input save area is left as it was initialized, in Lines 31-39, reflecting a new request. Similarly, if the previous transaction was a name search, we ignore the information in **COMMAREA,** if any. Because the **PA2** key was not pressed, the user doesn't want to continue the search. So the current entry is an entirely new request.

At this point, an explanation of the input save area, called **IN-AREA** in the program, would be helpful. In all but the very simplest applications, you have to expect that the user will sometimes key in bad data. The customary procedure, on receiving such data, is to send back a message or some other indication of what's wrong.

If you're programming in pseudoconversational mode, as we've chosen to do in our example application, the transaction that detects the error usually sends the error information back to the screen. It then returns control to CICS with the next transid pointing to itself. When the user corrects the data, the same transaction is invoked and starts all over again, this time (we hope) with good input. If the input still has errors, the cycle is repeated until it is error-free or the user quits trying.

This is the simplest method, in that the good input fields from the first entry don't have to be forwarded to the next execution of the transaction. That's because once the modified-data tag of a screen field is turned on (by the user entering data in the field), it remains on until turned off by the program. Thus, all the input fields are sent on each entry, regardless of how many times the user changes and resends the same screen.

The disadvantage to this approach is that line traffic increases because all the input fields are transmitted every time. This is *not* a

significant problem, and you need try to avoid it only if all of the following are true:

    The screen is very ful
    Line traffic is heav
    The incidence of errors and correction cycles is high

However, for the times when increased line traffic would be a significant problem, there's a second technique. In this method, the transaction checking the input moves the input fields to a save area. If it detects errors, it passes this copy of the original input along to the next execution of the transaction. **COMMAREA** is the handiest place to do this, but you can also use temporary storage. The transaction then turns off all the modified-data tags on the screen, by specifying **FRSET** when it writes the error message(s). Only the fields that have actually changed are sent. On this second time through (and any subsequent ones, if the user still makes mistakes) the transaction merges the new data with that saved from previous rounds.

We've used this second method in the **AC01** transaction, which this program (**ACCT01**) supports. For this reason, we collect all the input in **IN-AREA.** If the user makes a mistake, we pass the input from **IN-AREA,** through **COMMAREA** (in Line 396), to the next execution of the transaction. In program **ACCT02,** however, we use the more customary technique. We show both methods just to illustrate the difference; there isn't enough data in the menu screen used in **ACO1** to worry about resending it on an error cycle.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 164  *                                                                  ¦
¦ 165  *    GET INPUT AND CHECK REQUEST TYPE FURTHER.                     ¦
¦ 166       EXEC CICS RECEIVE MAP('ACCTMNU')                              ¦
¦ 167                 MAPSET('ACCTSET') RESP(RESPONSE) END-EXEC.         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 164 through 167: This statement causes CICS to rearrange the input into the symbolic map format dictated by map **ACCTMNU,** and to place this information in working storage at **ACCTMNUI.** (CICS had already read this input, as we noted earlier; it was the arrival of this input that caused the current transaction to start.)

A **MAPFAIL** condition can be raised on this command, as indeed can several other conditions. So we've specified the **RESP** option to find out, after execution, what condition has been raised on the **RECEIVE MAP.** The program can then check the value of **RESP** in the **RESPONSE** variable (defined on line 15) to see if any errors have occurred.

The **RESP** option allows processing to continue with the next COBOL statement.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 168       IF RESPONSE = DFHRESP(MAPFAIL) GO TO NO-MAP.                  ¦
¦ 169       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 168 through 169: After issuing the **RECEIVE** command with the **RESP** option, the response is checked. First the program checks for the **MAPFAIL** condition and, if this has occurred, transfers control to **NO-MAP.** Otherwise the program just checks to see if a **NORMAL** response has not been produced; if this is the case, it transfers control to **OTHER-ERRORS.**

The use of the **RESP** option on a CICS command followed by the explicit test of the key field has the same effect as **EXEC CICS HANDLE CONDITION (label)**, but improves the structure of the program as well as making it easier to understand and follow.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 170       IF REQML > 0 MOVE REQMI TO REQC.                            ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 170: If the user keyed a request code, we save it in **IN-AREA** at **REQC.** (**REQC** was initialized to a space, so we can tell later whether or not such a code has been entered by checking **REQC** to see if it still contains a space.)

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 171       IF REQMF NOT = LOW-VALUE, MOVE SPACE TO REQC.               ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 171: Next we check whether the user has erased a request code that was entered on an earlier transaction. The length field of the request subfield will be zero, meaning no input, but the old code will have been restored to **REQC** at Line 163. So we need to test the flag subfield as well as the length. If the flag is on, we need to erase the value in **REQC.** This check of the flag is an extra step associated with the second technique for handling errors described before Lines 166-167. If all the input fields come in fresh every time, as in the first approach, the length will tell you whether there is data there or not.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 172       IF ACCTML > 0 MOVE ACCTMI TO ACCTC.                         ¦
¦ 173       IF ACCTMF NOT = LOW-VALUE, MOVE SPACES TO ACCTC.            ¦
¦ 174       IF PRTRML > 0 MOVE PRTRMI TO PRTRC.                         ¦
¦ 175       IF PRTRMF NOT = LOW-VALUE, MOVE SPACES TO PRTRC.            ¦
¦ 176       IF SNAMEML > 0 MOVE SNAMEMI TO SNAMEC.                      ¦
¦ 177       IF SNAMEMF NOT = LOW-VALUE, MOVE SPACES TO SNAMEC.          ¦
¦ 178       IF FNAMEML > 0 MOVE FNAMEMI TO FNAMEC.                      ¦
¦ 179       IF FNAMEMF NOT = LOW-VALUE, MOVE SPACES TO FNAMEC.          ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 172 through 179: These statements process the other input fields in the same way as Lines 170-171 process the request code.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 180       MOVE LOW-VALUES TO ACCTMNUI.                                 ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 180: We clear the symbolic map area for the menu map to nulls again. We do this in case any new information (error messages or name search output) has to be sent using the same map. Clearing prevents information that is on the screen and not changed from being retransmitted, because BMS does not send null fields.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 181       IF IN-NAMES = SPACES GO TO CK-ANY.                           ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 181: Here we find out whether we have a name search request (by checking for the presence of some name input). If not, we skip to statement **CK-ANY** at Line 242.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 182  *                                                               ¦
¦ 183  *    NAME INQUIRY PROCESSING.                                   ¦
¦ 184  *    VALIDATE NAME INPUT.                                       ¦
¦ 185       IF FNAMEC NOT ALPHABETIC, MOVE 1 TO MSG-NO,                ¦
¦ 186           MOVE -1 TO FNAMEML, MOVE DFHBMBRY TO FNAMEMA.          ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 182 through 186: At this point, we know that the user wants a name search and we check the input name(s) for mistakes. In this program we'll indicate errors in the names, and other fields as well, as follows:

> The field(s) in error will be highlighted **MOVE DFHBMBRY TO FNAMEA** to set the bright attribute, for example).
>
> The cursor will be placed under the first field that is in erro (we'll move -1 to the length subfield for every such field, and CICS will find the first one for us).
>
> A message explaining the particular error or combination of error will be placed in the message area of the screen (**MOVE** 1 TO MSG-NO in combination with Lines 393-395). The message number is used as an index to the actual error message. Message number 1 produces the error message:
>
> NAMES MUST BE ALPHABETIC, AND SURNAME IS REQUIRED
>
> (see Line 98).
>
> If the user fails to fill in a required field, we'll place asterisk in the field as a convention to warn the user that we want him or her to fill it in:
>
> MOVE STARS TO SNAMEMO

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 187       IF SNAMEC ALPHABETIC AND SNAMEC NOT = SPACES GO TO CK-NAME. ¦
¦ 188       MOVE 1 TO MSG-NO.                                           ¦
¦ 189       MOVE -1 TO SNAMEML, MOVE DFHBMBRY TO SNAMEMA.               ¦
¦ 190   CK-NAME.                                                        ¦
¦ 191       IF MSG-NO > 0 GO TO MENU-RESEND.                            ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 187 through 191: These statements complete the validating of the names on which a search is requested. The surname is required and must be alphabetic; the first name is optional but must be alphabetic if present. At the end of these tests, we look at **MSG-NO** to see if there were any errors. It will be zero if there were none, because we initialized it that way in Line 15, and we'll continue at the next statement. Otherwise it will be the number of the error message to be put in the message area when the menu is redisplayed (at **MENU-RESEND, Line** 387).

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 192  *                                                               ¦
¦ 193  *    BUILD KEY AND LIMITING NAME VALUES FOR SEARCH.             ¦
```

```
194   SRCH-INIT.
195       MOVE SNAMEC TO BRKEY-SNAME, MAX-SNAME.
196       MOVE LOW-VALUES TO BRKEY-ACCT.
197       INSPECT MAX-SNAME REPLACING ALL SPACES BY HIGH-VALUES.
198       MOVE FNAMEC TO MIN-FNAME, MAX-FNAME.
199       INSPECT MIN-FNAME REPLACING ALL SPACES BY LOW-VALUES.
200       INSPECT MAX-FNAME REPLACING ALL SPACES BY HIGH-VALUES.
```

Lines 192 through 200 (SRCH-INIT): These statements initialize for the name search, as explained in connection with the **STARTBR** command in "Browsing a file" in topic 3.4.1.2.

MAX-SNAME is just higher in the alphabetical sequence than any surname that is eligible on the search.

MIN-FNAME and **MAX-FNAME** are the lowest and highest first names that are eligible on the search.

BRKEY is just lower than the key of the first eligible record in the index file.

```
201  *
202  *    INITIALIZE FOR SEQUENTIAL SEARCH.
203   SRCH-RESUME.
204       EXEC CICS STARTBR FILE('ACCTIX') RIDFLD(BRKEY) GTEQ
205           RESP(RESPONSE) END-EXEC.
206       IF RESPONSE = DFHRESP(NOTFND) GO TO SRCH-ANY.
207       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.
```

Lines 201 through 206 (SRCH-RESUME): At this point we've either computed all the values we need to perform a name search, or we've restored them from **COMMAREA,** where they were put by the previous execution of this transaction for this terminal (see Line 159).

We now begin the search of the file by pointing to the first eligible record in the index file with a **STARTBR** command, asking for the first record with a key equal to *or greater than* **BRKEY.**

Of the several unusual results that can occur on this command, we've concerned ourselves only with **NOTFND,** which occurs if we've constructed a starting key that is larger than the largest key in the file. This situation does not indicate a user or program error; it simply means that the user tried to search for a name not in the file and very late in the alphabet. So, if this happens, we send control to the same place that it goes after checking all the possibly eligible records in the file (namely, **SRCH-ANY** at Line 227).

```
208  *
209  *    BUILD NAME DISPLAY.
210   SRCH-LOOP.
211       EXEC CICS READNEXT FILE('ACCTIX') INTO(ACIXREC)
212           LENGTH(ACIX-LNG) RIDFLD(BRKEY) RESP(RESPONSE) END-EXEC.
213       IF RESPONSE = DFHRESP(ENDFILE) GO TO SRCH-DONE.
214       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.
```

Lines 211 through 214 (SRCH-LOOP): This command brings in the first (or next) record from the index file, starting at the point established in the **STARTBR** command. The only unusual conditions that we need to deal with on this command are **ENDFILE,** which will occur if the last name in the file isn't greater than the largest surname we allow, and anything other than a normal response to the command. So these are the only conditions we explicitly test **RESP** for. If an **ENDFILE** condition has arisen, we pass control to the same place that finding a name larger than the largest allowable surname would take us, (namely, **SRCH-DONE** at Line 225).

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 215       IF SNAMEDO IN ACIXREC > MAX-SNAME GO TO SRCH-DONE.           ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 215: If the surname in the index record is higher in the alphabetic sequence than the largest surname we allow, then we've read all the records that might be matches. We therefore go to **SRCH-DONE** (Line 225) to investigate the results of the search.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 216       IF FNAMEDO IN ACIXREC < MIN-FNAME OR                          ¦
¦ 217            FNAMEDO IN ACIXREC > MAX-FNAME, GO TO SRCH-LOOP.         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 216 through 217: If the surname is in range, we test whether the first name is also in range. If it is not, we simply loop back to Line 211 to read the next record.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 218       ADD 1 TO LINE-CNT.                                            ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 218: If both names match, we add one to our count of matches in **LINE-CNT.**

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 219       IF LINE-CNT > MAX-LINES,                                      ¦
¦ 220            MOVE MSG-TEXT (15) TO MSGMO,                             ¦
¦ 221            MOVE DFHBMBRY TO MSGMA, GO TO SRCH-DONE.                 ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 219 through 221: Next we work out if there's any room on the screen for the latest match. If there isn't (if **LINE-CNT** is greater than **MAX-LINES,** a constant that indicates how many search output lines there are on the menu screen), we know that we have to tell the user that there are more matches. We therefore move the appropriate text to the message area of the map.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 222       MOVE CORRESPONDING ACIXREC TO SUM-LINE.                       ¦
¦ 223       MOVE SUM-LINE TO SUMLNMO (LINE-CNT).                          ¦
¦ 224       GO TO SRCH-LOOP.                                              ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 222 through 224: On the other hand, if there **is** room on the screen

for the current name, we format the information in the record into a display line and move it to the next available line in the map. Then we go back to continue reading the index file at **SRCH-LOOP** (Line 211).

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 225   SRCH-DONE.                                                      ¦
¦ 226       EXEC CICS ENDBR FILE('ACCTIX') END-EXEC.                    ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Line 225 through 226 (SRCH-DONE): This is the end of the loop for reading index records, which we reach if:

1. We've no more room on the screen
2. We've read beyond the largest allowable surname
3. We've reached the end of the file.

All of the candidate records have been read at this point, so we end the browse.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 227   SRCH-ANY.                                                       ¦
¦ 228       IF LINE-CNT = 0, MOVE 7 TO MSG-NO,                          ¦
¦ 229           MOVE -1 TO SNAMEML, GO TO MENU-RESEND.                  ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 228 through 229 (SRCH-ANY): Next we check whether there were any matches at all to the name search. If not, we send a message to this effect to the user. Even though this isn't really an error, the processing is similar to error processing and so we use the code at **MENU-RESEND** (Line 393).

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 230  *                                                                ¦
¦ 231  *    SEND THE NAME SEARCH RESULTS TO TERMINAL.                   ¦
¦ 232       MOVE DFHBMASB TO MSGMA, SUMTTLMA.                           ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 230 through 232: We change some attribute bytes in the menu map in preparation for sending the results of the search to the terminal, for reasons explained in the next paragraph. Specifically, we first change the attributes of both the search output lines and the message field from their default in the map (autoskip) to unprotected. We also change the search output header line, which we want to show only in name search output, from nondisplay to autoskip. (This header and the message field are brightened at the same time, for emphasis.)

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 233       EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET')             ¦
¦ 234           FREEKB ERASE END-EXEC.                                  ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 233 through 234: Finally we send out results. **FREEKB** unlocks the keyboard for the user's next input, and appears on all of the **SEND MAP** commands in this application (see Lines 155 to 157). We specify **ERASE** to erase anything that may have been left on the screen from a previous execution of this transaction.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 235       IF LINE-CNT NOT > MAX-LINES,                                  ¦
¦ 236           EXEC CICS RETURN TRANSID('AC01') END-EXEC.                ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 235 through 236: If all the eligible names fit on the current screen, we return control to CICS, requesting that this same transaction be the next one executed. Nothing is saved in **COMMAREA,** because the current request is complete and the next one will be entirely new.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 237       ELSE EXEC CICS RETURN TRANSID('AC01') COMMAREA(SRCH-CTRL)      ¦
¦ 238               LENGTH(44) END-EXEC.                                  ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 237 through 238: If, however, there are eligible names remaining, we save all the search variables and the request type in **COMMAREA** so that the next transaction can resume the search if the user so requests.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 239  *                                                                  ¦
¦ 240  *    DISPLAY, PRINT, ADD, MODIFY AND DELETE PROCESSING.            ¦
¦ 241  *    CHECK ACCOUNT NUMBER.                                         ¦
¦ 242   CK-ANY.                                                           ¦
¦ 243       IF IN-REQ = SPACES, MOVE -1 TO SNAMEML,                       ¦
¦ 244           MOVE 8 TO MSG-NO, GO TO MENU-RESEND.                      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 239 through 244 (CK-ANY): By this point in the code, we've found out that the user doesn't want a name search (because he or she didn't fill in a name), and we begin checking for other request types. First we ensure that we got *some* input. If we didn't, we set the cursor to its normal position, set the error message accordingly, and go to send it at **MENU-RESEND** (Line 393).

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 245   CK-ACCTNO-1.                                                      ¦
¦ 246       IF ACCTC = SPACES                                             ¦
¦ 247           MOVE 5 TO MSG-NO, GO TO ACCT-ERR.                         ¦
¦ 248       IF (ACCTC < '10000' OR ACCTC > '79999' OR ACCTC NOT NUMERIC),  ¦
¦ 249           MOVE 6 TO MSG-NO, GO TO ACCT-ERR.                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 245 through 249 (CK-ACCTNO-1): Next we make sure the input is valid. All the remaining request types require an account number, which must be numeric and between **10 000** and **79 999.** We use the same diagnostic conventions for this and the remaining fields as for the name fields: highlighting (done in Line 262), asterisks if the field was omitted but is required (Line 246), cursor under the first error (Line 262), and an appropriate error message (Lines 247 and 249).

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 250   CK-ACCTNO-2.                                                      ¦
¦ 251       EXEC CICS READ FILE('ACCTFIL') RIDFLD(ACCTC) RESP(RESPONSE)   ¦
¦ 252           INTO(ACCTREC) LENGTH(ACCT-LNG) END-EXEC.                  ¦
¦                                                                         ¦
```

Lines 250 through 252: This command reads the account file record indicated by the account number in the input.

```
 253       IF RESPONSE = DFHRESP(NOTFND) GO TO NO-ACCT-RECORD.
 254       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.
 255       IF REQC = 'A',
 256           MOVE 9 TO MSG-NO, GO TO ACCT-ERR,
 257       ELSE GO TO CK-REQ.
```

Lines 253 through 257: We explicitly test **RESP** for two conditions. If there is no such record (the **NOTFND** condition), control will go to **NO-ACCT-RECORD** (Line 258), because of the command just executed in Line 253. If there is some other sort of error (any **NOT NORMAL** condition), control will go to **OTHER-ERRORS** at Line 408.

On the other hand, if we reach statement 255, we know that we've successfully read the record with the key in **ACCTC** into the area **ACCTREC.** We next test to see whether this is the result we expected. If the request was to add a record, the user has made an error, because there is already a record in the file with this number. In this case, therefore, we save the message number assigned to represent this particular error situation (to use as an index to the actual error message) and go to **ACCT-ERR** (Line 261) to diagnose an error in the account number. For other request types, however, this is the response we expect, and we continue processing at **CK-REQ** (Line 265).

```
 258   NO-ACCT-RECORD.
 259       IF REQC = 'A', GO TO CK-REQ.
```

Lines 258 through 259 (NO-ACCT-RECORD): This statement is executed only if the record that we try to read in line 251 isn't in the account file. If the user has asked to add a record, this is the only correct response to the **READ** command, and we continue processing at **CK-REQ**, line 265.

```
 260       MOVE 10 TO MSG-NO.
```

Line 260: If the user has asked to display, print, modify or delete, however, this not-found response means that the account number is wrong. We set the message number accordingly, and continue at the next line to complete diagnosing an error in the account number.

```
 261   ACCT-ERR.
 262       MOVE -1 TO ACCTML, MOVE DFHBMBRY TO ACCTMA.
```

Line 261 (ACCT-ERR): Control reaches these statements from several points earlier in the program, after an error in the account number has been detected and the appropriate message number set. The statements complete

the processing of an error in the account number, by brightening the field and positioning the cursor.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 263  *                                                                 ¦
¦ 264  *    CHECK REQUEST TYPE.                                          ¦
¦ 265   CK-REQ.                                                         ¦
¦ 266       IF REQC =  'D' OR 'P' OR 'A' OR 'M' OR 'X',                  ¦
¦ 267           IF MSG-NO = 0 GO TO CK-USE, ELSE GO TO MENU-RESEND.      ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 263 through 267 (CK-REQ): The next input field we check is the request type. If it is one of the types permitted, we look at **MSG-NO,** which tells us whether there was an error detected earlier (in the account field). If it is zero (no error), we continue checking the input at **CK-USE** (line 273); otherwise we go to **MENU-RESEND** (line 387) to send out the diagnostic information.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 268       MOVE -1 TO REQML, MOVE DFHBMBRY TO REQMA,                    ¦
¦ 269       MOVE 3 TO MSG-NO.                                            ¦
¦ 270       GO TO MENU-RESEND.                                           ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 268 through 270: Control reaches this point when we do not have a good request type. We process an error in this field in the same way as one in the account field (see explanation for lines 245-249), and then go to **MENU-RESEND** (Line 387) to send the error information.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 271  *                                                                 ¦
¦ 272  *    TEST IF ACCOUNT NUMBER IN USE, ON UPDATES ONLY.              ¦
¦ 273   CK-USE.                                                         ¦
¦ 274       IF REQC = 'P' OR 'D' GO TO BUILD-MAP.                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 273 (CK-USE): At this point we have a good request for a good account number. If the request is for an update, however, we need to make a further check to ensure that no one else is updating this record at the moment. This test isn't required for a display or print request, however, and this statement skips the check on these types of requests.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 275       MOVE ACCTC TO USE-QID2.                                      ¦
¦ 276       EXEC CICS READQ TS QUEUE(USE-QID) INTO(USE-REC)              ¦
¦ 277           ITEM(USE-ITEM) LENGTH(USE-LNG) RESP(RESPONSE) END-EXEC.  ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 275 through 277: This command begins the test to ensure that the account number is available for update. We read the temporary storage queue whose name is **AC0** followed by the account number in question (see lines 21 to 23 for the structure and initialization of this name).

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 278       IF RESPONSE = DFHRESP(QIDERR) GO TO RSRV-1.                  ¦
¦ 279       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.        ¦
```

```
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 278 through 279: We explicitly test **RESP** for two conditions. If there is no temporary storage queue with that name (the **QIDERR** condition) which, in turn, means that the number isn't in use, control will go to **RSRV-1** (Line 296), where we'll reserve the number for ourselves. If there is some other sort of error (any **NOT NORMAL** condition), control will go to **OTHER-ERRORS** at Line 408.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 280       ADD USE-LIMIT TO USE-TIME.                                   ¦
¦ 281       IF USE-TIME > 236000, ADD 1 TO USE-DATE,                     ¦
¦ 282           SUBTRACT 236000 FROM USE-TIME.                           ¦
¦ 283       IF USE-DATE > EIBDATE OR                                     ¦
¦ 284           (USE-DATE = EIBDATE AND USE-TIME NOT < EIBTIME)          ¦
¦ 285           MOVE USE-TERM TO MSG-TERM, MOVE 11 TO MSG-NO,            ¦
¦ 286           MOVE -1 TO ACCTML, MOVE DFHBMBRY TO ACCTMA,              ¦
¦ 287           GO TO MENU-RESEND.                                       ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 280 through 287: On a normal response to our **READQ TS** command (normal in the CICS sense, that is), control comes to this statement, the one following the command. This response means either that the number is in use or that a scratchpad entry for it has been left in temporary storage because of an accident on an earlier update against the same record. (See "Pseudoconversational or not?" in topic 2.7 for a discussion of how this might occur.)

As we explained earlier, we'll distinguish between the two cases by comparing the time in the scratchpad record (**USE-TIME** on **USE-DATE**) with the time that the current transaction started (**EIBTIME** on **EIBDATE**). If more than a certain amount of time has passed, we'll go ahead and claim the number for ourselves at **RSRV** in Line 290. Otherwise we tell the user that the account number is in use. In the code, we treat the latter situation as an error in the account number (Lines 285-287); the only difference is that we customize the error message by moving in the name of the terminal using the number (stored at **USE-TERM** in the temporary storage record). The allowed time is stored in the constant **USE-LIMIT,** defined in Line 28 to be 10 minutes and 00 seconds. (The code in Line 281 takes care of the possibility that the scratchpad was written just before midnight, assuming that **USE-LIMIT** is less than an hour.)

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 288   *                                                                ¦
¦ 289   *    RESERVE ACCOUNT NUMBER.                                     ¦
¦ 290    RSRV.                                                           ¦
¦ 291       MOVE EIBTRMID TO USE-TERM, MOVE EIBTIME TO USE-TIME.         ¦
¦ 292       MOVE EIBDATE TO USE-DATE.                                    ¦
¦ 293       EXEC CICS WRITEQ TS QUEUE(USE-QID) FROM(USE-REC)             ¦
¦ 294           LENGTH(12) ITEM(USE-ITEM) REWRITE END-EXEC.              ¦
¦ 295       GO TO BUILD-MAP.                                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 288 through 295 (RSRV): These statements are executed if there was an old scratchpad record for the account number whose time has expired. They reserve the account number for use by the current terminal. The scratchpad record is built from the name of the current input terminal and the time and date that the current transaction started. This record is then written to temporary storage, *replacing* the expired record. The **REWRITE** option causes replacement of the queue record whose number is in

USE-ITEM. This number, defined at Line 29, is always 1, because we've designed our scratchpad to use single-item queues. If there are any errors in executing this command, control will go to **OTHER-ERRORS,** (Line 408), as dictated by Line 143. Otherwise we pass control to **BUILD-MAP,** in Line 303, where we build the output screen for the impending update.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 296   RSRV-1.                                                         ¦
¦ 297       MOVE EIBTRMID TO USE-TERM, MOVE EIBTIME TO USE-TIME.        ¦
¦ 298       MOVE EIBDATE TO USE-DATE.                                   ¦
¦ 299       EXEC CICS WRITEQ TS QUEUE(USE-QID) FROM(USE-REC)            ¦
¦ 300           LENGTH(12) END-EXEC.                                    ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 296 through 300 (RSRV-1): These statements are executed if there was *no* scratchpad record for the account number. They serve the same purpose as Lines 290-295 (reserving the account number for the current input terminal), but a different form of the **WRITEQ TS** command is needed, because we're creating a new queue.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 301  *                                                                ¦
¦ 302  *    BUILD THE RECORD DISPLAY.                                   ¦
¦ 303   BUILD-MAP.                                                      ¦
¦ 304       IF REQC = 'X' MOVE 'DELETION' TO TITLEDO,                   ¦
¦ 305           MOVE -1 TO VFYDL, MOVE DFHBMUNP TO VFYDA,               ¦
¦ 306           MOVE 'ENTER "Y" TO CONFIRM OR "CLEAR" TO CANCEL'        ¦
¦ 307               TO MSGDO,                                           ¦
¦ 308       ELSE MOVE -1 TO SNAMEDL.                                    ¦
¦ 309       IF REQC = 'A' MOVE 'NEW RECORD' TO TITLEDO,                 ¦
¦ 310           MOVE DFHPROTN TO STATTLDA, LIMTTLDA, HISTTLDA,          ¦
¦ 311           MOVE ACCTC TO ACCTDI,                                   ¦
¦ 312           MOVE 'FILL IN AND PRESS "ENTER," OR "CLEAR" TO CANCEL'  ¦
¦ 313               TO MSGDO,                                           ¦
¦ 314           GO TO SEND-DETAIL.                                      ¦
¦ 315       IF REQC = 'M' MOVE 'RECORD CHANGE' TO TITLEDO,              ¦
¦ 316           MOVE 'MAKE CHANGES AND "ENTER" OR "CLEAR" TO CANCEL'    ¦
¦ 317               TO MSGDO,                                           ¦
¦ 318       ELSE IF REQC = 'D',                                         ¦
¦ 319               MOVE 'PRESS "CLEAR" OR "ENTER" WHEN FINISHED'       ¦
¦ 320                   TO MSGDO.                                       ¦
¦ 321       MOVE ACCTDO IN ACCTREC TO ACCTDO IN ACCTDTLO.               ¦
¦ 322       MOVE SNAMEDO IN ACCTREC TO SNAMEDO IN ACCTDTLO.             ¦
¦ 323       MOVE FNAMEDO IN ACCTREC TO FNAMEDO IN ACCTDTLO.             ¦
¦ 324       MOVE MIDO IN ACCTREC TO MIDO IN ACCTDTLO.                   ¦
¦ 325       MOVE TTLDO IN ACCTREC TO TTLDO IN ACCTDTLO.                 ¦
¦ 326       MOVE TELDO IN ACCTREC TO TELDO IN ACCTDTLO.                 ¦
¦ 327       MOVE ADDR1DO IN ACCTREC TO ADDR1DO IN ACCTDTLO.             ¦
¦ 328       MOVE ADDR2DO IN ACCTREC TO ADDR2DO IN ACCTDTLO.             ¦
¦ 329       MOVE ADDR3DO IN ACCTREC TO ADDR3DO IN ACCTDTLO.             ¦
¦ 330       MOVE AUTH1DO IN ACCTREC TO AUTH1DO IN ACCTDTLO.             ¦
¦ 331       MOVE AUTH2DO IN ACCTREC TO AUTH2DO IN ACCTDTLO.             ¦
¦ 332       MOVE AUTH3DO IN ACCTREC TO AUTH3DO IN ACCTDTLO.             ¦
¦ 333       MOVE AUTH4DO IN ACCTREC TO AUTH4DO IN ACCTDTLO.             ¦
¦ 334       MOVE CARDSDO IN ACCTREC TO CARDSDO IN ACCTDTLO.             ¦
¦ 335       MOVE IMODO IN ACCTREC TO IMODO IN ACCTDTLO.                 ¦
¦ 336       MOVE IDAYDO IN ACCTREC TO IDAYDO IN ACCTDTLO.               ¦
¦ 337       MOVE IYRDO IN ACCTREC TO IYRDO IN ACCTDTLO.                 ¦
¦ 338       MOVE RSNDO IN ACCTREC TO RSNDO IN ACCTDTLO.                 ¦
¦ 339       MOVE CCODEDO IN ACCTREC TO CCODEDO IN ACCTDTLO.             ¦
¦ 340       MOVE APPRDO IN ACCTREC TO APPRDO IN ACCTDTLO.               ¦
¦ 341       MOVE SCODE1DO IN ACCTREC TO SCODE1DO IN ACCTDTLO.           ¦
```

```
¦ 342       MOVE SCODE2DO IN ACCTREC TO SCODE2DO IN ACCTDTLO.            ¦
¦ 343       MOVE SCODE3DO IN ACCTREC TO SCODE3DO IN ACCTDTLO.            ¦
¦ 344       MOVE STATDO IN ACCTREC TO STATDO IN ACCTDTLO.                ¦
¦ 345       MOVE LIMITDO IN ACCTREC TO LIMITDO IN ACCTDTLO.              ¦
¦ 346       MOVE CORRESPONDING PAY-HIST (1) TO PAY-LINE.                 ¦
¦ 347       MOVE PAY-LINE TO HIST1DO.                                    ¦
¦ 348       MOVE CORRESPONDING PAY-HIST (2) TO PAY-LINE.                 ¦
¦ 349       MOVE PAY-LINE TO HIST2DO.                                    ¦
¦ 350       MOVE CORRESPONDING PAY-HIST (3) TO PAY-LINE.                 ¦
¦ 351       MOVE PAY-LINE TO HIST3DO.                                    ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 352       IF REQC  = 'M' GO TO SEND-DETAIL,                            ¦
¦ 353       ELSE IF REQC = 'P' GO TO PRINT-PROC.                         ¦
¦ 354       MOVE DFHBMASK TO                                             ¦
¦ 355           SNAMEDA, FNAMEDA, MIDA, TTLDA, TELDA, ADDR1DA,           ¦
¦ 356           ADDR2DA, ADDR3DA, AUTH1DA, AUTH2DA, AUTH3DA,             ¦
¦ 357           AUTH4DA, CARDSDA, IMODA, IDAYDA, IYRDA, RSNDA,           ¦
¦ 358           CCODEDA, APPRDA, SCODE1DA, SCODE2DA, SCODE3DA.           ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 304 through 348 (BUILD-MAP): At this point we're ready to build the output screen. Since we're using the same map for all types of requests, we have to make certain adjustments, depending on the type of request. Specifically, we must:

1. Put a text description of the request type in the title line. (Lines 304, 309, 315 do this for delete, add, and modify requests, respectively. The default in the map takes care of the most common case, a display request, and also applies to print requests.)

2. Arrange for the cursor to be under the proper field. For a deletion, this is the verify field (see line 305, first part). For other requests, it is the surname field (line 308).

3. For deletions, change the attribute of the verify field (**VFYDA**) from autoskip to unprotected (line 305, second part).

4. Tell the user, in the message area, what to do next after completing the screen. Lines 306 through 307 do this for deletes; Lines 312 through 313 are for adds, 314 through 315 for modifications and 318 through 320 for display requests. We do not want any such message in the output for a print request, so the message area is left empty in this case.

5. For additions, darken the title lines for the payment history at the bottom of the screen, since this part of the screen does not apply to add requests (line 310).

6. Also for additions, put the account number in the input request into the screen (line 311). This is the only field that can be filled in on an addition; we put the account number there for two reasons: to save the user the trouble, and to make sure he or she doesn't change it. (Having gone to some lengths to ensure that it was a good number and not in use at another terminal, we cannot let the user change it now.)

7. For requests other than additions, move the contents of the account file record for the requested account number into the map (lines 321 to 327).

8. For display and delete requests, protect all of the fields from the record that aren't protected by default in the map. This reminds users that they cannot change the record in display or delete operations (Lines 354 to 358).

```
359  *
360  *    SEND THE RECORD DETAIL MAP TO THE TERMINAL.
361   SEND-DETAIL.
362       EXEC CICS SEND MAP('ACCTDTL') MAPSET('ACCTSET') ERASE FREEKB
363           CURSOR END-EXEC.
```

Lines 359 through 363 (SEND-DETAIL): This command sends the output map (prepared in the preceding statements) to the (input) terminal, for all types of requests except print requests. The **ERASE** option is used, because a new map is being displayed. We specify **CURSOR** without a value, to tell CICS to put the cursor in the first field with a length value of -1.

```
364       IF REQC = 'D', EXEC CICS RETURN TRANSID('ACCT') END-EXEC,
```

Line 364: Now we return control to CICS, after sending output to the terminal. If the request was to display a record (that is, **REQC=D**), the request is complete at this point. The requested record is on the screen, and the user has been instructed to use either the **CLEAR** or **ENTER** key after inspecting the record. Since the next thing the user will want to see is a menu screen, we set the next transaction identifier to **ACCT,** which will display the menu screen, whatever key is next used to send input. We *do not* specify a **COMMAREA,** because there is no information about the current transaction that needs to be passed to the next one.

```
365       ELSE EXEC CICS RETURN TRANSID('AC02')
366               COMMAREA(IN-REQ) LENGTH(6) END-EXEC.
```

Lines 365 through 366: On the other hand, if the request was an add, modify or delete, we set the next transaction identifier to **AC02,** which does the second part of an update request, and we pass the account number and the request type to that transaction through **COMMAREA.**

```
367  *
368  *    START UP A TASK TO PRINT THE RECORD.
369   PRINT-PROC.
370       IF PRTRC = SPACES
371           MOVE 4 TO MSG-NO, GO TO TERMID-ERR1.
```

Lines 369 through 372 (PRINT-PROC): This code applies only to print requests. Control is brought here by the test in Line 353, because output to be printed cannot be sent to the input terminal. We've not checked, up to this point, whether the user has given us a good printer name. We didn't do this with the earlier validating because doing so requires

interrogation of the terminal control table (TCT). You can do this in CICS, but not with the type of commands included in the Primer. However, you get the same check automatically with the **START** command, so we've waited until now to make this test.

If the printer name is omitted or is all spaces, you know you have trouble, because spaces aren't an acceptable terminal identifier. So we check for this error first. If we find it, we fill the field with asterisks to show that it is required, and we reinforce this with an appropriate message. Then we go directly to the code that completes noting an error in this field (**TERMID-ERR1** at Line 382).

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 372       EXEC CICS START TRANSID('AC03') FROM(ACCTDTLO)              ¦
¦ 373           LENGTH(DTL-LNG) TERMID(PRTRC) RESP(RESPONSE) END-EXEC.  ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 372 through 373: Otherwise we issue the **START** command to initiate the transaction that will do the printing. The name of this transaction is **AC03,** and the data that we'll pass it consists of the detail map we built (at **ACCTDTLO**) in Lines 304 to 353. The length of this data, **DTL-LNG,** is stored at a constant defined in Line 19. The name of the terminal that must be available to the transaction before it can be started is at **PRTRC.** Because we didn't specify any **TIME** or **INTERVAL** parameter, CICS will start the transaction as soon as it can after the required terminal is free.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 374       IF RESPONSE = DFHRESP(TERMIDERR) GO TO TERMID-ERR.           ¦
¦ 375       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO OTHER-ERRORS.       ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 374 through 375: We explicitly test **RESP** for two conditions. The most probable result of this read will be **TERMIDERR,** meaning that there is no terminal identifier corresponding to the one that has been entered. In this case, control will go to **TERMID-ERR** (Line 380). If there is some other sort of error (any **NOT NORMAL** condition), control will go to **OTHER-ERRORS** at Line 408.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 376       MOVE MSG-TEXT (12) TO MSGMO.                                ¦
¦ 377       EXEC CICS SEND MAP('ACCTMNU') MAPSET ('ACCTSET') DATAONLY   ¦
¦ 378            ERASEAUP FREEKB END-EXEC.                              ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 376 through 378: If the **START** command is successful, control falls through to this statement. We need to send a message, saying that the user's print request has been scheduled. We use the **DATAONLY** option, so that the only thing we send is the message itself. And we use **ERASEAUP** to erase the print request input, so that the user is all set to enter the next request.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 379       EXEC CICS RETURN TRANSID('AC01') END-EXEC.                   ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Line 379: Finally we return control to CICS. The next transaction

identifier is set to **AC01,** because the menu is still on the screen. However, there is no **COMMAREA,** because the current request has been completed.

```
 380   TERMID-ERR.
 381       MOVE 13 TO MSG-NO.
 382   TERMID-ERR1.
 383       MOVE -1 TO PRTRML, MOVE DFHBMBRY TO PRTRMA.
```

Lines 380 through 383 (TERMID-ERR): If, on the other hand, the terminal to which we attempted to **START** a transaction isn't in the TCT, control is sent here (see Lines 145 and 370 through 371). In this case, we choose a message appropriate to the situation and flag the error in the usual fashion.

```
 384  *
 385  *    ERROR PROCESSING, FOR ALL REQUESTS.
 386  *    RESEND MENU SCREEN.
 387   MENU-RESEND.
 388       MOVE REQC TO REQMI.
 389       MOVE ACCTC TO ACCTMI.
 390       MOVE PRTRC TO PRTRMI.
 391       MOVE SNAMEC TO SNAMEMI.
 392       MOVE FNAMEC TO FNAMEMI.
 393       MOVE MSG-TEXT (MSG-NO) TO MSGMO.
```

Lines 384 through 393 (MENU-RESEND): This statement begins the code used to display input errors. It first moves the applicable error message to the menu map, using the message number that was set earlier in the program.

```
 394       EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET')
 395           CURSOR DATAONLY FRSET ERASEAUP FREEKB END-EXEC.
```

Lines 394 through 395: This command sends the changes and additions to the screen. We cleared the map to nulls (Line 147) before the editing began, and we specify **DATAONLY** here. So the only data that will be sent is:

1. The changed attribute bytes
2. Any fields we filled with asterisks
3. The message field (if any).

This is where we specify **FRSET,** so that only changed fields are sent on the next transmission, as we explained at Line 163.

```
 396       EXEC CICS RETURN TRANSID('AC01') COMMAREA(IN-AREA)
 397                LENGTH(41) END-EXEC.
```

Lines 396 through 397: After we've added to the menu display, we return control to CICS. Because the menu screen is on display, we set the next transaction identifier to this same transaction, **AC01.** And because some sort of error occurred on this request, we also save the input from the current execution in **COMMAREA** for use in the next execution of the transaction.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 398  *                                                                  ¦
¦ 399  *    PROCESSING FOR MAP FAILURES, CLEARS.                          ¦
¦ 400   NO-MAP.                                                          ¦
¦ 401       MOVE 2 TO MSG-NO, MOVE -1 TO SNAMEML, GO TO MENU-RESEND.      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 400 through 401 (NO-MAP): Control reaches this point if **MAPFAIL** occurs when we **RECEIVE** the input map. **MAPFAIL** could happen in this situation if the user pressed the **ENTER** key, or one of the PF keys, without keying anything into the screen. This is because no modified-data tags are turned on by the program or the map, so that if the user does this, there will be no fields sent (**MAPFAIL** by definition). It could also happen if one of the short-read keys that we've not already tested for is used. So the first thing we do is find out whether the user pressed **ENTER** or one of these short-read keys. If so, we send a message pointing out that he or she has to enter *some* data and use either the **ENTER** or **CLEAR** keys, once again using the error code at Lines 393 through 397.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 402   NEW-MENU.                                                         ¦
¦ 403       EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET')               ¦
¦ 404           FREEKB ERASE END-EXEC.                                    ¦
¦ 405       EXEC CICS RETURN TRANSID ('AC01') END-EXEC.                   ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 402 through 405: If the user managed to get a **MAPFAIL** in some other way, we send a message saying there has been an input error and inviting another attempt, using only the **CLEAR** and **ENTER** keys. We send this on an entirely fresh map, since we cannot be sure what's on the screen at this point. Then we return control to CICS. The next transid is **AC01,** because we've just put the menu on the screen. There is no **COMMAREA,** because we want a fresh start.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 406  *                                                                  ¦
¦ 407  *    PROCESSING FOR UNEXPECTED ERRORS.                             ¦
¦ 408   OTHER-ERRORS.                                                     ¦
¦ 409       MOVE EIBFN TO ERR-FN, MOVE EIBRCODE TO ERR-RCODE.             ¦
¦ 410       MOVE EIBFN TO ERR-COMMAND, MOVE EIBRESP TO ERR-RESP.          ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 408 through 410 (OTHER-ERRORS): This statement begins the code that is executed on any unusual response to a CICS command except **MAPFAIL, NOTFND, ENDFILE, QIDERR,** and **TERMIDERR.** CICS sends control here, rather than abending the task, because of the **ERROR(OTHER-ERRORS)** option on our **HANDLE CONDITION ERROR** command in Line 145. The first thing we do is save the type of command we were trying to execute at the time of the error (which is at **EIBFN**) and the response code we got (which is at **EIBRCODE**). We must do this before we issue any other CICS command, or this vital information will be overwritten.

```
411       EXEC CICS HANDLE CONDITION ERROR END-EXEC.
```

Line 411: Next we disable the **HANDLE CONDITION ERROR** request that brought us to this paragraph. We do so to prevent any possibility of a loop. If we did not, and we experienced some unusual condition on the **LINK** command that follows, the program would loop.

Note: We would also "unhandle" any *other* specifically handled conditions that could occur at this point, if there was any possibility of them occurring on the **LINK** command that's coming up next. (However, there are none as we've instead chosen to test **RESP** options on our CICS commands.)

```
412       EXEC CICS LINK PROGRAM('ACCT04')
413           COMMAREA(COMMAREA-FOR-ACCT04) LENGTH(14) END-EXEC.
```

Lines 412 through 413: Finally, we transfer control to program **ACCT04,** which will send the user a message about the nature of the error. Notice we're using **LINK** rather than **XCTL** -- see "Errors within the example application" in topic 3.8.3.

We pass three items of information along to this program, in the area named **COMMAREA-FOR-ACCT04** (see Lines 40-45). These items are the function and response codes just saved and the name of the current program (**ACCT01**).

```
414       GOBACK.
```

Line 414: This **GOBACK** statement is actually never executed, because control doesn't return from our error handling program, **ACCT04.** However, this logical "end of program" keeps the compiler happy.

4.3 Program ACCT02: update processing

```
001   IDENTIFICATION DIVISION.
002   PROGRAM-ID. ACCT02.
003  *REMARKS. THIS PROGRAM IS THE FIRST INVOKED BY THE 'AC02'
004  *         TRANSACTION.  IT COMPLETES REQUESTS FOR ACCOUNT FILE
005  *         UPDATES (ADDS, MODIFIES, AND DELETES), AFTER THE USER
006  *         ENTERED THE UPDATE INFORMATION.
007   ENVIRONMENT DIVISION.
008   DATA DIVISION.
009   WORKING STORAGE SECTION.
010   01  MISC.
011       02  RESPONSE              PIC S9(8) COMP.
012       02  OWN-FLAG              PIC 9.
013       02  MENU-MSGNO            PIC S9(4) COMP VALUE +1.
014       02  DTL-MSGNO             PIC S9(4) COMP VALUE +0.
015       02  ACCT-LNG              PIC S9(4) COMP VALUE +383.
016       02  ACIX-LNG              PIC S9(4) COMP VALUE +63.
017       02  DTL-LNG               PIC S9(4) COMP VALUE +751.
018       02  DUMMY                 PIC S9(4) COMP VALUE +128.
019       02  FILLER REDEFINES DUMMY.
020           04  FILLER            PIC X.
021           04  HEX80             PIC X.
022       02  STARS                 PIC X(12) VALUE '************'.
023       02  USE-QID.
024           04  USE-QID1          PIC X(3) VALUE 'AC0'.
025           04  USE-QID2          PIC X(5).
026       02  USE-REC.
027           04  USE-TERM          PIC X(4).
028           04  USE-TIME          PIC S9(7) COMP-3.
029           04  USE-DATE          PIC S9(7) COMP-3.
030       02  USE-LNG               PIC S9(4) COMP VALUE +12.
031       02  OLD-IXKEY.
032           04  IXOLD-SNAME       PIC X(12).
033           04  IXOLD-ACCT        PIC X(5).
034       02  COMMAREA-FOR-ACCT04.
035           04  ERR-PGRMID        PIC X(8) VALUE 'ACCT02'.
036           04  ERR-FN            PIC X.
037           04  ERR-RCODE         PIC X.
038           04  ERR-COMMAND       PIC XX.
039           04  ERR-RESP          PIC 99.
040       02  PAY-INIT              PIC X(36) VALUE
041               '    0.00000000    0.00000000    0.00'.
```

```
042  *    MESSAGES DISPLAYED ON MENU SCREEN
043       02  MENU-MSG-LIST.
044           04  FILLER            PIC X(60) VALUE
045               'PREVIOUS REQUEST CANCELLED AS REQUESTED'.
046           04  FILLER            PIC X(60) VALUE
047               'REQUESTED ADDITION COMPLETED'.
048           04  FILLER            PIC X(60) VALUE
049               'REQUESTED MODIFICATION COMPLETED'.
050           04  FILLER            PIC X(60) VALUE
051               'REQUESTED DELETION COMPLETED'.
052  *    MESSAGES DISPLAYED ON DETAIL SCREEN
053       02  MENU-MSG REDEFINES MENU-MSG-LIST PIC X(60) OCCURS 4.
054       02  DTL-MSG-LIST.
055           04  FILLER            PIC X(60) VALUE
056               'EITHER ENTER "Y" TO CONFIRM OR "CLEAR" TO CANCEL'.
057           04  FILLER            PIC X(60) VALUE
```

```
¦ 058           'YOUR REQUEST WAS INTERRUPTED; PLEASE CANCEL AND RETRY'. ¦
¦ 059           04  FILLER             PIC X(60) VALUE                  ¦
¦ 060           'CORRECT HIGHLIGHTED ITEMS (STARS MEAN ITEM REQUIRED)'.  ¦
¦ 061           04  FILLER             PIC X(60) VALUE                  ¦
¦ 062           'USE ONLY "ENTER" (TO PROCEED) OR "CLEAR" (TO CANCEL)'.  ¦
¦ 063           04  FILLER             PIC X(60) VALUE                  ¦
¦ 064           'MAKE SOME ENTRIES AND "ENTER" OR "CLEAR" TO CANCEL'.    ¦
¦ 065       02  DTL-MSG REDEFINES DTL-MSG-LIST PIC X(60) OCCURS 5.      ¦
¦ 066       02  MOD-LINE.                                               ¦
¦ 067           04  FILLER             PIC X(25) VALUE                  ¦
¦ 068               '==========> CHANGES TO:  '.                         ¦
¦ 069           04  MOD-NAME           PIC X(6) VALUE SPACES.           ¦
¦ 070           04  MOD-TELE           PIC X(5) VALUE SPACES.           ¦
¦ 071           04  MOD-ADDR           PIC X(6) VALUE SPACES.           ¦
¦ 072           04  MOD-AUTH           PIC X(6) VALUE SPACES.           ¦
¦ 073           04  MOD-CARD           PIC X(6) VALUE SPACES.           ¦
¦ 074           04  MOD-CODE           PIC X(5) VALUE SPACES.           ¦
¦ 075       02  UPDT-LINE.                                              ¦
¦ 076           04  FILLER             PIC X(30) VALUE                  ¦
¦ 077               '==========> UPDATED AT TERM:  '.                    ¦
¦ 078           04  UPDT-TERM          PIC X(4).                        ¦
¦ 079           04  FILLER             PIC X(6) VALUE '  AT  '.         ¦
¦ 080           04  UPDT-TIME          PIC 9(7).                        ¦
¦ 081           04  FILLER             PIC X(6) VALUE '  ON  '.         ¦
¦ 082           04  UPDT-DATE          PIC 9(7).                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 10 through 82: These lines are the working storage area of the program. Individual variables will be explained in the comments below as they are used.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 083   01  NEW-ACCTREC. COPY ACCTREC.                                   ¦
¦ 084   01  OLD-ACCTREC. COPY ACCTREC.                                   ¦
¦ 085   01  NEW-ACIXREC. COPY ACIXREC.                                   ¦
¦ 086   01  OLD-ACIXREC. COPY ACIXREC.                                   ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 83 through 86: These lines copy in the record formats for the account and index files. There's space for two records for each file--one for the old version of the record (before modification or deletion) and one for the new version (for modifications and additions).

See "The account file record format" in topic 3.4.1.1.1 and "The index file record format" in topic 3.4.1.1.2. for the source code of **ACCTREC** and **ACIXREC.**

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 087       COPY ACCTSET.                                                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 87: This line brings in a copy of the symbolic description map structure.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 088       COPY DFHAID.                                                 ¦
¦ 089       COPY DFHBMSCA.                                               ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 88 through 89: These lines bring in the definitions of the attention identifiers and attribute bytes that CICS provides for COBOL programmers (See "Finding out what key the operator pressed" in topic 3.3.6 and "Symbolic description maps (DSECT structures)" in topic 3.3.1.)

```
090   LINKAGE SECTION.
091   01  DFHCOMMAREA.
092       02  REQC                  PIC X.
093       02  ACCTC                 PIC X(5).
```

Lines 90 through 93: The structure defined here and named **DFHCOMMAREA** describes the data passed to this program by means of **COMMAREA.**

```
094  *
095   PROCEDURE DIVISION.
096  *
097   MAIN SECTION.
098  *    INITIALIZE.
099       MOVE LOW-VALUES TO ACCTDTLI.
```

Line 99: This **MOVE** statement initializes the symbolic map structure for the detail map to nulls, in preparation for receiving input (see notes for Line 143 in program **ACCT01**).

```
100       MOVE SPACES TO OLD-ACCTREC, NEW-ACCTREC,
101           OLD-ACIXREC, NEW-ACIXREC.
```

Lines 100 through 101: The areas in which new account and index file records will be built (and read) are initialized to spaces.

```
102  *    CATER FOR UNEXPECTED ERRORS
103       EXEC CICS HANDLE CONDITION ERROR(NO-GOOD) END-EXEC.
```

Line 103: This command tells CICS where control should go if unexpected errors are encountered. Specific conditions that might result from user errors and conditions that CICS regards as unusual, but that the program expects, are handled with explicit code later in the program by the **RESP** option. Examples of these are **MAPFAIL, LENGERR,** and **QIDERR.** The program does not attempt to recover from other unusual conditions, and therefore all of these are passed, by means of this **HANDLE CONDITION ERROR** command, to a single point in the program (**NO-GOOD** at Line 172), from which control is sent to an error program. This program in turn sends a message to the user and abends the task.

Nothing happens, as the result of executing this **HANDLE CONDITION ERROR** command, that immediately affects the flow of the program or the data available to it. Instead, this command causes CICS to record information

for processing exceptional conditions in this particular program, should they occur subsequently.

See the preface for some general comments about error handling in the **ACCT** application for this edition of the Primer. You'll also be able to find more detailed guidance in the CICS/ESA Application Programming Guide.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 104       IF EIBAID = DFHCLEAR THEN                                     ¦
¦ 105           PERFORM CK-OWN                                            ¦
¦ 106           IF OWN-FLAG = 1 GO TO NO-OWN ELSE                         ¦
¦ 107           GO TO RELEASE-ACCT.                                       ¦
¦ 108       IF EIBAID NOT = DFHENTER THEN                                 ¦
¦ 109           GO TO PA-KEY.                                             ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 104 through 109: If the user has pressed the **CLEAR** key, control passes to **CK-OWN** in Line 317. If the user has not pressed the **ENTER** key, control passes to **PA-KEY** in Line 168.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 110  *                                                                  ¦
¦ 111  *    GET INPUT AND BUILD NEW RECORD.                               ¦
¦ 112       EXEC CICS RECEIVE MAP('ACCTDTL') MAPSET('ACCTSET')            ¦
¦ 113                 RESP(RESPONSE) END-EXEC.                          ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 110 through 113: This command tells CICS:

1. To convert the terminal input (whose arrival caused the current transaction to be initiated) into the symbolic map format required by map **ACCTDTL**

2. To place this information in working storage at **ACCTDTLI.**

A **MAPFAIL** condition can be raised on this command, as indeed can several other conditions. So we've specified the **RESP** option to find out, after execution, what condition has been raised on the **RECEIVE MAP.** The program can then check the value of **RESP** in the **RESPONSE** variable (defined on line 11) to see if any errors have occurred.

The **RESP** option allows processing to continue with the next COBOL statement.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 114       IF RESPONSE = DFHRESP(MAPFAIL) GO TO NO-MAP.                  ¦
¦ 115       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO NO-GOOD.              ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 114 through 115: We explicitly test **RESP** for two conditions. The most probable error condition arising from this read will be **MAPFAIL,** in which case control will go to **NO-MAP** (Line 163). If there is some other sort of error (any **NOT NORMAL** condition), control will go to **NO-GOOD** at Line 172.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 116       IF REQC NOT = 'A',                                            ¦
¦ 117           EXEC CICS READ FILE('ACCTFIL') INTO(OLD-ACCTREC)          ¦
```

```
¦ 118               RIDFLD(ACCTC) UPDATE LENGTH(ACCT-LNG) END-EXEC.      ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 116 through 118: Next we test to find out what kind of request we're processing. The request code, you'll remember, has been passed from the previous transaction, **AC01,** to this one in **COMMAREA** in the variable **REQC.** If the request is to modify or delete a record, as opposed to adding one, we read the target record from the account file. (The key for this record was also passed through **COMMAREA,** in the variable **ACCTC**.) The record is placed in the structure **OLD-ACCTREC.** Notice that we specify **UPDATE,** since that is what we're about to do. We must also specify a maximum length, which we do with the constant in **ACCT-LNG.**

Unlike program **ACCT01,** this program does not expect, and has no code to handle, any of the many irregularities possible on a file read. So, if anything unusual happens during this read, CICS will pass control to paragraph **NO-GOOD** at Line 172 (because of the **HANDLE CONDITION ERROR** command at Line 103).

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 119           MOVE OLD-ACCTREC TO NEW-ACCTREC,                         ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 119: After a successful read, a new version of the account record is initialized to the contents of the old record.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 120           MOVE SNAMEDO IN OLD-ACCTREC TO IXOLD-SNAME,              ¦
¦ 121           MOVE ACCTC TO IXOLD-ACCT.                                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 120 through 121: We also build the key of the index file record associated with this account record, defined in Lines 31 to 33, for use later in the program.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 122       IF REQC = 'X',                                               ¦
¦ 123           IF VFYDI = 'Y', GO TO NO-EDIT                            ¦
¦ 124           ELSE MOVE -1 TO VFYDL, MOVE DFHUNIMD TO VFYDA,           ¦
¦ 125               MOVE 1 TO DTL-MSGNO,                                 ¦
¦ 126               GO TO INPUT-REDISPLAY.                               ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 122 through 126: These statements do the simple verification checking we need for delete requests.

If the user has confirmed the requested deletion by putting "Y" in the "verify" field, then the only test requirement is met, and the program goes on to ensure that there has been no lapse in control of the account number in question (in other words, the user still "owns" that account record) at **CK-OWN** (Line 317). However, if this field contains anything else, the program assumes an error ("Y" in the "verify" field or a **CLEAR** are the only acceptable responses in this situation).

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 127  *                                                                 ¦
¦ 128       PERFORM EDIT.                                                ¦
```

```
 129       IF DTL-MSGNO = 3 OR DTL-MSGNO = 5
 130           GO TO INPUT-REDISPLAY.
```

Lines 128 through 130 (PERFORM EDIT): The same diagnostic conventions are used in this program as in **ACCT01** (see discussion of Lines 172-173 in that program). **DTL-MSGNO** is the number of the error message that is sent to the screen (which is displaying the detail map).

```
 131  *
 132   NO-EDIT.
 133       PERFORM CK-OWN.
 134       IF OWN-FLAG = 1 GO TO NO-OWN.
```

Lines 132 through 134 (NO-EDIT): We go to **CK-OWN** (Lines 320-321) to check the ownership of the account number, otherwise if we do not have exclusive control of the account number we're trying to update, we go to **NO-OWN** (Line 160).

```
 135  *
 136       PERFORM UPDTE.
```

Line 136 (PERFORM UPDTE): This is where the program carries out the **PERFORM UPDTE** operation on the files.

```
 137  *
 138  *    RELEASE OWNERSHIP OF ACCOUNT NUMBER.
 139   RELEASE-ACCT.
 140       EXEC CICS DELETEQ TS QUEUE(USE-QID) END-EXEC.
```

Line 140 (RELEASE-ACCT): At this point, the files have been updated, and we can release exclusive control of the account number. We do this by deleting the corresponding scratchpad record.

```
 141  *
 142  *    SEND MENU MAP BACK TO TERMINAL.
 143   MENU-REFRESH.
 144       MOVE LOW-VALUES TO ACCTMNUO.
 145       MOVE MENU-MSG (MENU-MSGNO) TO MSGMO.
 146       EXEC CICS SEND MAP('ACCTMNU') MAPSET('ACCTSET') ERASE FREEKB
 147           END-EXEC.
```

Lines 143 through 147 (MENU-REFRESH): The final step in processing an update is to redisplay the menu screen, with a message in the message area confirming that the requested update has been completed. Line 145 moves in the appropriate message, based on the message number set up in Line 412, 421, or 442; and the next two lines send the screen. We send an entirely new map and use the **ERASE** option, since another map (the detail

map) is currently on the screen.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 148       EXEC CICS RETURN TRANSID('AC01') END-EXEC.                   ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 148: Then we return control to CICS. The next transid is set to **AC01** because the menu map is on the screen, ready for a new request to be entered.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 149  *                                                                 ¦
¦ 150  *    FOR INPUT ERRORS, RESEND DETAIL MAP.                         ¦
¦ 151   INPUT-REDISPLAY.                                                ¦
¦ 152       MOVE DTL-MSG (DTL-MSGNO) TO MSGDO.                           ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 152 (INPUT-REDISPLAY): This statement begins a block of code that is executed if the program detects any errors in the input. First, the appropriate error message is moved to the message area of the map.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 153       IF DTL-MSGNO = 2 OR 4 OR 5, MOVE -1 TO SNAMEDL.              ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 153: Next, if the error isn't one that would place the cursor elsewhere, the cursor is placed under the surname field. (The errors that place the cursor elsewhere are field errors and failures to confirm deletions.)

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 154       EXEC CICS SEND MAP('ACCTDTL') MAPSET('ACCTSET') DATAONLY     ¦
¦ 155           CURSOR FREEKB END-EXEC.                                  ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 154 through 155: Then the error information is sent. As in the corresponding code in program **ACCT01** (Lines 377-378 there), we use the **DATAONLY** option to send as little as possible to the screen. We also specify **CURSOR** without a value, so that CICS will place the cursor under the first field with a -1 in the length subfield. This will be the first field in error if there has been a field error.

We don't specify **FRSET** as we did in **ACCT01,** however. If we did so, we'd have to save the input from this execution of the **AC02** transaction, because only the fields changed after this **SEND MAP** command would be returned on the next **RECEIVE** command, as we explained in connection with Line 155 of program **ACCT01.**

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 156       EXEC CICS RETURN TRANSID('AC02') COMMAREA(DFHCOMMAREA)       ¦
¦ 157           LENGTH(6) END-EXEC.                                      ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 156 through 157: Having written the error information to the screen, we return control to CICS. We request that this same transaction

be the next one executed (to process the user's corrected input). We also have to save the information passed by transaction **AC01** for this execution of **AC02** for when we execute **AC02** again. So we forward the **COMMAREA** passed *to* the current execution of this transaction to its next execution.

```
158  *
159  *    PROCESSING FOR RECOVERABLE ERRORS.
160   NO-OWN.
161       IF EIBAID = DFHCLEAR OR MENU-MSGNO = 5 GO TO MENU-REFRESH.
162       MOVE 2 TO DTL-MSGNO, GO TO INPUT-REDISPLAY.
```

Lines 161 through 162 (NO-OWN): This is the code that is executed if we find that we do not have exclusive control of the account number we're trying to update. It may be that the user has cancelled a request, or a **MAPFAIL** of unexplained origin has occurred (so that we're not sure about the condition of the screen). In either case, we simply refresh the menu screen at Line 143 (**MENU-REFRESH**) with the applicable message (the message number is set in Line 13 or Line 167).

Otherwise, however, we must treat the situation as an error. We tell the user what has happened in the message area and indicate that he or she must cancel and start again (see the discussion in connection with Line 321).

```
163   NO-MAP.
164  *    IF MAPFAIL OCCURRED THEN REDISPLAY WITH APPROPRIATE MESSAGE.
165       IF REQC = 'X' MOVE 1 TO DTL-MSGNO, MOVE -1 TO VFYDL
166          ELSE MOVE 5 TO DTL-MSGNO.
167       GO TO INPUT-REDISPLAY.
```

Lines 164 through 166 (NO-MAP): This statement begins the code that is executed if a **MAPFAIL** condition occurs when we receive the input map. If the **ENTER** key was used, we assume that the user didn't enter any data into the detail map. We send a message saying that at least some entry is necessary, using the code for other types of errors at **INPUT-REDISPLAY** (Line 152).

If the **ENTER** key was not used, and if we know from Lines 104 and 108 that neither **CLEAR** nor one of the **PA** keys was used either, then the cause of the **MAPFAIL** is more obscure. In this unlikely event, we proceed as if the user had cleared the screen, except that we use a different message on the menu screen when we display it. We therefore go to **CK-OWN** (Lines 320-321) to release the account number.

```
168   PA-KEY.
169       MOVE 4 TO DTL-MSGNO, GO TO INPUT-REDISPLAY.
```

Line 169 (PA-KEY): This line is executed if a **PA** key is used to send the input. We handle the situation in the same way as the **ENTER** key without data, except for a different error message (see Lines 164-165).

```
170  *                                                                  
171  *    PROCESSING FOR UNRECOVERABLE ERRORS.                          
172   NO-GOOD.                                                         
173       MOVE EIBFN TO ERR-FN, MOVE EIBRCODE TO ERR-RCODE.            
174       MOVE EIBFN TO ERR-COMMAND, MOVE EIBRESP TO ERR-RESP.         
175       EXEC CICS HANDLE CONDITION ERROR END-EXEC.                   
176       EXEC CICS LINK PROGRAM('ACCT04')                             
177           COMMAREA(COMMAREA-FOR-ACCT04) LENGTH(14) END-EXEC.       
```

Lines 173 through 177 (NO-GOOD): These statements are executed whenever there is an unusual response to a CICS command, as shown by each **IF RESPONSE NOT = DFHRESP(NORMAL)** test. The code is identical to the corresponding code in Lines 409-413 of program **ACCT01.**

```
178       GOBACK.                                                    
```

Line 178: This **GOBACK** performs the same function that the **GOBACK** ending **ACCT01** does. It satisfies the COBOL compiler requirement for a logical end of program.

```
179   MAIN-EXIT.                                                     
180       EXIT.                                                      
```

Lines 179 through 180: This marks the end of the main-exit procedure, and control is returned to CICS.

```
181   EDIT SECTION.                                                  
182   EDIT-START.                                                    
183       IF SNAMEDL > 0 MOVE SNAMEDI TO SNAMEDO IN NEW-ACCTREC.     
184       IF FNAMEDL > 0 MOVE FNAMEDI TO FNAMEDO IN NEW-ACCTREC.     
185       IF MIDL > 0 MOVE MIDI TO MIDO IN NEW-ACCTREC.              
186       IF TTLDL > 0 MOVE TTLDI TO TTLDO IN NEW-ACCTREC.           
187       IF TELDL > 0 MOVE TELDI TO TELDO IN NEW-ACCTREC.           
188       IF ADDR1DL > 0 MOVE ADDR1DI TO ADDR1DO IN NEW-ACCTREC.     
189       IF ADDR2DL > 0 MOVE ADDR2DI TO ADDR2DO IN NEW-ACCTREC.     
190       IF ADDR3DL > 0 MOVE ADDR3DI TO ADDR3DO IN NEW-ACCTREC.     
191       IF AUTH1DL > 0 MOVE AUTH1DI TO AUTH1DO IN NEW-ACCTREC.     
192       IF AUTH2DL > 0 MOVE AUTH2DI TO AUTH2DO IN NEW-ACCTREC.     
193       IF AUTH3DL > 0 MOVE AUTH3DI TO AUTH3DO IN NEW-ACCTREC.     
194       IF AUTH4DL > 0 MOVE AUTH4DI TO AUTH4DO IN NEW-ACCTREC.     
195       IF CARDSDL > 0 MOVE CARDSDI TO CARDSDO IN NEW-ACCTREC.     
196       IF IMODL > 0 MOVE IMODI TO IMODO IN NEW-ACCTREC.           
197       IF IDAYDL > 0 MOVE IDAYDI TO IDAYDO IN NEW-ACCTREC.        
198       IF IYRDL > 0 MOVE IYRDI TO IYRDO IN NEW-ACCTREC.           
199       IF RSNDL > 0 MOVE RSNDI TO RSNDO IN NEW-ACCTREC.           
200       IF CCODEDL > 0 MOVE CCODEDI TO CCODEDO IN NEW-ACCTREC.     
201       IF APPRDL > 0 MOVE APPRDI TO APPRDO IN NEW-ACCTREC.        
202       IF SCODE1DL > 0 MOVE SCODE1DI TO SCODE1DO IN NEW-ACCTREC.  
203       IF SCODE2DL > 0 MOVE SCODE2DI TO SCODE2DO IN NEW-ACCTREC.  
204       IF SCODE3DL > 0 MOVE SCODE3DI TO SCODE3DO IN NEW-ACCTREC.  
```

Lines 181 through 204: For all add and modify requests, the fields that the user filled in or changed (that is, that have a length subfield greater than zero) are moved in to replace the corresponding fields in the new version of the account record.

```
205       IF REQC = 'A' GO TO EDIT-0.
206       IF SNAMEDF = HEX80 MOVE SPACES TO SNAMEDO IN NEW-ACCTREC.
207       IF FNAMEDF = HEX80 MOVE SPACES TO FNAMEDO IN NEW-ACCTREC.
208       IF MIDF = HEX80 MOVE SPACES TO MIDO IN NEW-ACCTREC.
209       IF TTLDF = HEX80 MOVE SPACES TO TTLDO IN NEW-ACCTREC.
210       IF TELDF = HEX80 MOVE SPACES TO TELDO IN NEW-ACCTREC.
211       IF ADDR1DF = HEX80 MOVE SPACES TO ADDR1DO IN NEW-ACCTREC.
212       IF ADDR2DF = HEX80 MOVE SPACES TO ADDR2DO IN NEW-ACCTREC.
213       IF ADDR3DF = HEX80 MOVE SPACES TO ADDR3DO IN NEW-ACCTREC.
214       IF AUTH1DF = HEX80 MOVE SPACES TO AUTH1DO IN NEW-ACCTREC.
215       IF AUTH2DF = HEX80 MOVE SPACES TO AUTH2DO IN NEW-ACCTREC.
216       IF AUTH3DF = HEX80 MOVE SPACES TO AUTH3DO IN NEW-ACCTREC.
217       IF AUTH4DF = HEX80 MOVE SPACES TO AUTH4DO IN NEW-ACCTREC.
218       IF CARDSDF = HEX80 MOVE SPACE TO CARDSDO IN NEW-ACCTREC.
219       IF IMODF = HEX80 MOVE ZERO TO IMODO IN NEW-ACCTREC.
220       IF IDAYDF = HEX80 MOVE ZERO TO IDAYDO IN NEW-ACCTREC.
221       IF IYRDF = HEX80 MOVE ZERO TO IYRDO IN NEW-ACCTREC.
222       IF RSNDF = HEX80 MOVE SPACE TO RSNDO IN NEW-ACCTREC.
223       IF CCODEDF = HEX80 MOVE SPACES TO CCODEDO IN NEW-ACCTREC.
224       IF APPRDF = HEX80 MOVE SPACES TO APPRDO IN NEW-ACCTREC.
225       IF SCODE1DF = HEX80 MOVE SPACES TO SCODE1DO IN NEW-ACCTREC.
226       IF SCODE2DF = HEX80 MOVE SPACES TO SCODE2DO IN NEW-ACCTREC.
227       IF SCODE3DF = HEX80 MOVE SPACES TO SCODE3DO IN NEW-ACCTREC.
```

Lines 205 through 227: For modifications only, we test to see if the user erased the information in the old version of the record. (Erasure on the screen sets the flag subfield to a value of hexadecimal 80.) Any field erased on the screen is set to spaces in the new version of the record.

```
228       IF OLD-ACCTREC = NEW-ACCTREC,
229           MOVE 5 TO DTL-MSGNO,
230           GO TO EDIT-99.
```

Lines 228 through 230: Also for modifications, after the new record is built, it is compared to the old record. If there is no difference, the program treats the situation as an error, and goes to **INPUT-REDISPLAY** (Line 152) to send the appropriate message.

```
231  *    EDIT INPUT.
232   EDIT-0.
233       MOVE LOW-VALUES TO ACCTDTLI.
```

Line 233 (EDIT-0): The detail map structure is cleared to nulls, in preparation for (maybe) having to use the area again to send error information to the user. As in program **ACCT01,** we'll send only the information that needs to be changed on the screen. For some fields this will mean the entire field (as when we move asterisks into a field); for others it may mean only the attribute byte (when we just highlight the

field).

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 234       IF SNAMEDO IN NEW-ACCTREC = SPACES,                       ¦
¦ 235           MOVE STARS TO SNAMEDI,                                ¦
¦ 236       ELSE IF SNAMEDO IN NEW-ACCTREC ALPHABETIC GO TO EDIT-1.   ¦
¦ 237       MOVE DFHUNIMD TO SNAMEDA, MOVE -1 TO SNAMEDL.             ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Lines 234 through 237: These statements begin the validation of each individual field in the account record which the user may have entered on the screen. The checks in this program are very simple. In a real application, however, they are often so complex that they can form the great bulk of the program's code. For this reason, they are sometimes relegated to a separate program, which is executed by either a **LINK** command or an **XCTL** command, depending on whether control must return to the original program or not.

The first field we check is the surname, which is required and must be alphabetic. We use the same diagnostic conventions as in program **ACCT01.** If a field that is required has been left blank, we fill it with asterisks to remind the user that it is required. If there's any error in the field, we set the length subfield to -1, so that CICS will place the cursor under the first such field. We also highlight the field, to call it to the user's attention.

Notice that we use a different value for highlighting in this program from that used in program **ACCT01.** In this program, we want an attribute byte that means bright, unprotected and modified-data tag on, whereas in **ACCT01** we wanted bright, unprotected and modified-data tag off. This is because here we want to be sure that anything the user keyed on any cycle of the current request gets transmitted every time the screen is sent. In **ACCT01,** by contrast, we wanted just the fields changed on the most recent cycle to be sent. (Refer to the discussion of Line 161 in program **ACCT01** for more information about this.)

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 238   EDIT-1.                                                       ¦
¦ 239       IF FNAMEDO IN NEW-ACCTREC = SPACES,                       ¦
¦ 240           MOVE STARS TO FNAMEDI,                                ¦
¦ 241       ELSE IF FNAMEDO IN NEW-ACCTREC ALPHABETIC, GO TO EDIT-2.  ¦
¦ 242       MOVE DFHUNIMD TO FNAMEDA, MOVE -1 TO FNAMEDL.             ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Lines 239 through 242 (edit-1): These statements check the first name, which is required and must be alphabetic.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 243   EDIT-2.                                                       ¦
¦ 244       IF MIDO IN NEW-ACCTREC NOT ALPHABETIC,                    ¦
¦ 245           MOVE DFHUNIMD TO MIDA, MOVE -1 TO MIDL.               ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Lines 243 through 245 (EDIT-2): The middle initial isn't required, but must be alphabetic if present.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 246       IF TTLDO IN NEW-ACCTREC NOT ALPHABETIC,                   ¦
```

```
¦ 247           MOVE DFHUNIMD TO TTLDA, MOVE -1 TO TTLDL.                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 246 through 247: The title part of a name (Jr, Sr, and so on) must also be alphabetic if present, but isn't required.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 248       IF (TELDO IN NEW-ACCTREC NOT = SPACES AND                    ¦
¦ 249               TELDO IN NEW-ACCTREC NOT NUMERIC),                   ¦
¦ 250           MOVE DFHUNIMD TO TELDA, MOVE -1 TO TELDL.                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 248 through 250: The telephone number isn't required, but if present, must be all numeric.

We should mention here that there is a temptation to assume that if a field (like the telephone number here) has the numeric attribute in the map, the corresponding input subfield is automatically numeric in the COBOL sense. Unfortunately, this isn't necessarily true.

First, the user may not have entered anything into it, and it may contain nulls. Second, if the user did enter something, it may contain periods and minus signs, which the 3278 considers to be numeric. (On some keyboards, still other keys are allowed.) Such items may not be in the right place in the field to make sense, or they may not make sense at all for the field in question (this one, for example). Finally, some terminals don't have the numeric lock feature, which allows the 3270 system to distinguish between unprotected fields and numeric fields. So we still have to check such fields for proper values.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 251       IF ADDR1DO IN NEW-ACCTREC = SPACES,                          ¦
¦ 252           MOVE STARS TO ADDR1DI,                                   ¦
¦ 253           MOVE DFHBMBRY TO ADDR1DA, MOVE -1 TO ADDR1DL.            ¦
¦ 254       IF ADDR2DO IN NEW-ACCTREC = SPACES,                          ¦
¦ 255           MOVE STARS TO ADDR2DI,                                   ¦
¦ 256           MOVE DFHBMBRY TO ADDR2DA, MOVE -1 TO ADDR2DL.            ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 251 through 256: An address consisting of at least two lines is required, but there are no other checks on the (up to) four lines of address information.

Notice that we use **DFHBMBRY** (MDT off) instead of **DFHUNIMD** (MDT on) for these two fields. You see, if we used **DFHUNIMD,** and the user didn't replace the asterisks (**STARS**) that we put there (to show the field was needed) then we'd get the asterisks sent back. And since all we're requiring in the address is *something*, these asterisks would pass our edit check. This isn't a problem with other fields, such as the name, because asterisks will fail the more demanding edit checks on these other fields.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 257       IF CARDSDO IN NEW-ACCTREC = SPACES,                          ¦
¦ 258           MOVE STARS TO CARDSDI,                                   ¦
¦ 259       ELSE IF (CARDSDO IN NEW-ACCTREC > '0' AND                    ¦
¦ 260               CARDSDO IN NEW-ACCTREC NOT > '9'), GO TO EDIT-3.     ¦
¦ 261       MOVE DFHUNIMD TO CARDSDA, MOVE -1 TO CARDSDL.                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 257 through 261: The number of cards issued is needed and must be between 1 and 9.

```
 262   EDIT-3.
 263       IF IMODO IN NEW-ACCTREC = SPACES,
 264           MOVE STARS TO IMODI,
 265       ELSE IF IMODO IN NEW-ACCTREC NUMERIC AND
 266           IMODO IN NEW-ACCTREC > '00' AND
 267           IMODO IN NEW-ACCTREC < '13', GO TO EDIT-4.
 268       MOVE DFHUNIMD TO IMODA, MOVE -1 TO IMODL.
 269   EDIT-4.
 270       IF IDAYDO IN NEW-ACCTREC = SPACES,
 271           MOVE STARS TO IDAYDI,
 272       ELSE IF IDAYDO IN NEW-ACCTREC NUMERIC AND
 273               IDAYDO IN NEW-ACCTREC > '00' AND
 274               IDAYDO IN NEW-ACCTREC < '32',
 275               GO TO EDIT-5.
 276       MOVE DFHUNIMD TO IDAYDA, MOVE -1 TO IDAYDL.
 277   EDIT-5.
 278       IF IYRDO IN NEW-ACCTREC = SPACES,
 279           MOVE STARS TO IYRDI,
 280       ELSE IF IYRDO IN NEW-ACCTREC NUMERIC AND
 281           IYRDO IN NEW-ACCTREC > '75', GO TO EDIT-6.
 282       MOVE DFHUNIMD TO IYRDA, MOVE -1 TO IYRDL.
```

Lines 263 through 282 (EDIT-3, EDIT-4, EDIT-5): The date of issue, which consists of a month, day and year, is required. The program checks that the month is between 1 and 12, that the day is between 1 and 31, and that the year is at least 75. This is clearly not a complete date check, because it lets in things like June 31, to say nothing of leap-year problems. You'll probably want a much tighter check in any real application. (Thorough date checks can be so complex that you may want to use a separate routine.)

You can use a separate CICS program and **LINK** to it if you don't have too many dates to check. If you do, and you need a subroutine, it's more efficient to use a standard COBOL subroutine and link it into the program with the linkage editor. (See "Subroutines revisited" in topic 3.6.2.5.)

```
 283   EDIT-6.
 284       IF RSNDO IN NEW-ACCTREC = SPACES,
 285           MOVE STARS TO RSNDI,
 286       ELSE IF (RSNDO IN NEW-ACCTREC = 'N' OR
 287               RSNDO IN NEW-ACCTREC = 'L' OR
 288               RSNDO IN NEW-ACCTREC = 'S' OR
 289               RSNDO IN NEW-ACCTREC = 'R'), GO TO EDIT-7.
 290       MOVE DFHUNIMD TO RSNDA, MOVE -1 TO RSNDL.
```

Lines 284 through 290 (EDIT-6): The **RSNDO** (reason for issue) code must be filled in and must be either N (for new), L (lost), S (stolen), or R (reissued--for some other reason).

```
 291   EDIT-7.
 292       IF CCODEDO IN NEW-ACCTREC = SPACES,
```

```
¦ 293           MOVE STARS TO CCODEDI,                                  ¦
¦ 294           MOVE -1 TO CCODEDL, MOVE DFHBMBRY TO CCODEDA.           ¦
¦ 295       IF APPRDO IN NEW-ACCTREC = SPACES,                          ¦
¦ 296           MOVE STARS TO APPRDI,                                   ¦
¦ 297           MOVE -1 TO APPRDL, MOVE DFHBMBRY TO APPRDA.             ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 292 through 297 (EDIT-7): The "card code" and the initials of the approver must both be present, and not spaces.

See Lines 251 through 256 for an explanation of the use of **DFHMBRY.**

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 298       IF ACCTDTLI NOT = LOW-VALUES,                               ¦
¦ 299           MOVE 3 TO DTL-MSGNO, GO TO EDIT-99.                     ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 298 through 299: All the checking is complete at this point, so we test to see if there were any errors by looking at the map area. If it is all nulls, as we initialized it at **EDIT-0** (Line 233), then there were no errors. Otherwise there were, and we send the error information at **INPUT-REDISPLAY** (Line 152). We use the same general message for all field-checking errors (namely that the highlighted fields contain errors), rather than sending field-specific messages like the ones in **ACCT01.**

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 300       IF REQC = 'A' MOVE ACCTC TO ACCTDO IN NEW-ACCTREC,          ¦
¦ 301           MOVE 'N ' TO STATDO IN NEW-ACCTREC,                     ¦
¦ 302           MOVE ' 1000.00' TO LIMITDO IN NEW-ACCTREC,              ¦
¦ 303           MOVE PAY-INIT TO PAY-HIST IN NEW-ACCTREC (1),           ¦
¦ 304               PAY-HIST IN NEW-ACCTREC (2),                        ¦
¦ 305               PAY-HIST IN NEW-ACCTREC (3).                        ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 300 through 305: These statements are for add requests only, and initialize those fields in the new record that are maintained in the batch system rather than online. The account status field is set to a value meaning "new account," the credit limit is set to an arbitrary $1000, and the payment history fields are all set to zeros.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 306       MOVE ACCTDO IN NEW-ACCTREC TO ACCTDO IN NEW-ACIXREC.        ¦
¦ 307       MOVE SNAMEDO IN NEW-ACCTREC TO SNAMEDO IN NEW-ACIXREC.      ¦
¦ 308       MOVE FNAMEDO IN NEW-ACCTREC TO FNAMEDO IN NEW-ACIXREC.      ¦
¦ 309       MOVE MIDO IN NEW-ACCTREC TO MIDO IN NEW-ACIXREC.            ¦
¦ 310       MOVE TTLDO IN NEW-ACCTREC TO TTLDO IN NEW-ACIXREC.          ¦
¦ 311       MOVE ADDR1DO IN NEW-ACCTREC TO ADDR1DO IN NEW-ACIXREC.      ¦
¦ 312       MOVE STATDO IN NEW-ACCTREC TO STATDO IN NEW-ACIXREC.        ¦
¦ 313       MOVE LIMITDO IN NEW-ACCTREC TO LIMITDO IN NEW-ACIXREC.      ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 306 through 313: Here we build the index record corresponding to this account record. The index record will be needed on all add requests and some modification requests.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 314   EDIT-99.                                                        ¦
```

```
¦ 315       EXIT.                                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Lines 314 through 315: This marks the end of the editing procedures referred to in lines 228-230 and 298-299, and control is returned to CICS.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦ 316  *                                                                   ¦
¦ 317   CK-OWN SECTION.                                                    ¦
¦ 318  *    CHECK OWNERSHIP OF ACCOUNT NUMBER.                             ¦
¦ 319   CK-01.                                                            ¦
¦ 320       MOVE 0 TO OWN-FLAG.                                            ¦
¦ 321       MOVE ACCTC TO USE-QID2.                                        ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Lines 319 through 321 (CK-OWN): The lines that begin here ensure that the account number that is about to be updated is still exclusively controlled by the terminal that entered this transaction.

We need to include this code because we've allowed for a time limit on the ownership of an account number, and we don't yet know how long it has been since this particular user requested control of the number. (See "Pseudoconversational or not?" in topic 2.7 for a discussion of how the terminal that originated the update might lose control of an account number.)

It doesn't really matter if the time has expired, so long as our terminal has had uninterrupted control of the number. However, if someone else has made or is making a change to this record in the meantime, we cannot let this user proceed with the update. At best, the current transaction will fail later because of a duplicate or missing record. And because this user is working with an old copy of the record, we could lose the changes made subsequently by another user.

This situation will occur rarely, and only if the user leaves the operation half-done for a long period of time, or if some sort of system error occurs. When it does occur, we'll just send the user a message saying that the request has been interrupted and that it must be resubmitted. We'll not erase the screen, because the user may wish to look at it further, even though the only thing he or she can do next is to cancel the transaction by pressing **CLEAR.**

The first step in this rechecking process is to build the name of the temporary storage queue that contains the scratchpad information for this account number.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦ 322       EXEC CICS READQ TS QUEUE(USE-QID) INTO(USE-REC)                ¦
¦ 323           LENGTH(USE-LNG) ITEM(1) RESP(RESPONSE) END-EXEC.           ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Lines 322 through 323: Next, we read the scratchpad entry for this account number. We specify the first item in the queue, since that is where we saved the information. (We might not get the first item in the queue if some other terminal had attempted to use our number, because program **ACCT01** would have read the first item on behalf of that terminal, and CICS would be ready to send the next item to the next transaction that read the queue.) We must also specify a maximum length for the item, initialized to 12 in Line 30.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 324       IF RESPONSE = DFHRESP(LENGERR)                              ¦
¦ 325          OR RESPONSE = DFHRESP(QIDERR) THEN                       ¦
¦ 326           MOVE 1 TO OWN-FLAG                                      ¦
¦ 327           GO TO CK-EXIT.                                          ¦
¦ 328       IF RESPONSE NOT = DFHRESP(NORMAL) GO TO NO-GOOD.            ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 324 through 328: We explicitly test **RESP** for three conditions. The most probable error conditions arising from this command will be **LENGERR** (length error) and **QIDERR** (queue name error), in which case control will go to the **CK-EXIT** label with the **OWN-FLAG** value set to one. We thus return to Line 134 (the line after the **PERFORM** verb that brought us here in the first place), where we promptly examine the value of **OWN-FLAG.**

If there is some other sort of error (any **NOT NORMAL** condition), control will go to **NO-GOOD** at Line 173, and thus, ultimately, to the error-handling program, **ACCT04.**

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 329       IF USE-TERM NOT = EIBTRMID OR USE-LNG NOT = 12              ¦
¦ 330           MOVE 1 TO OWN-FLAG                                      ¦
¦ 331           GO TO CK-EXIT.                                          ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 329 through 331: Control comes here if we successfully read the scratchpad. We then ensure that the name of the terminal that "owns" the number is the same as the name of the input terminal. We also ensure that the length of the entry is correct, as a further check on the validity of the entry. If either condition fails, control will go to the **CK-EXIT** label with the **OWN-FLAG** value set to one.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 332       IF EIBAID = DFHCLEAR OR MENU-MSGNO = 5 GO TO CK-EXIT.       ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Line 325: At this point, we've established that the input terminal still has control over the account number. Now we have to recheck how we got here. Three paths in the code might have brought us here:

1. A request to cancel (that is, use of the **CLEAR** key)

2. A **MAPFAIL** that was not caused by a short-read key (see **NO-MAP, Lines** 164-167)

3. A normal sequence (no errors, no cancellation).

For either of the first two conditions, we want to release control of the number, so that some other task can use it. We do this at **RELEASE-ACCT, Line** 140.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 333  *                                                               ¦
¦ 334  *    WRITE HARDCOPY LOG RECORDS.                                ¦
¦ 335       MOVE LOW-VALUES TO ACCTDTLO.                               ¦
¦ 336       MOVE DFHBMDAR TO HISTTLDA, STATTLDA, STATDA, LIMTTLDA,     ¦
¦ 337           LIMITDA.                                               ¦
¦                                                                       ¦
```

```
+-------------------------------------------------------------------------+
```

Lines 335 through 337: Otherwise, we proceed with the update. The first thing we do is to write an entry to the hard-copy log that we're required to produce. We'll format this log as follows:

For additions, we'll print the new record, using the same format that we use on the screen (the "detail" map).

For modifications, we'll print both the old version of the record and the new one, again using the map format. In the message area of the old record we'll note the areas that were changed (name, address, and so on), to make it easy for the supervisor to check.

For deletions, we'll print the old record.

For all types:

1. We'll note the contents of the screen in the title line of the map: **NEW RECORD** for additions, **BEFORE CHANGE** and **AFTER CHANGE** for the two images printed on a modification, and **DELETION** on a delete.

2. We'll show the time and date of the update and the name of the terminal at which it was entered. We'll put this information in the message area (for modifications, it will be in the "new" record image).

These two statements simply get things ready for this process. The map is cleared to nulls, and certain fields in the map that should not appear in the log output are made nondisplay. (These are the title fields for the information from the record that is maintained by the batch system, and not subject to online update.)

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 338       IF REQC = 'A' MOVE 'NEW RECORD' TO TITLEDO, GO TO LOG-1.      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 338: Next, we check the request type. If the update's an addition, the proper title is placed in the map and control goes to **LOG-1** at Line 398, where the rest of the output will be built.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 339       MOVE ACCTDO IN OLD-ACCTREC TO ACCTDO IN ACCTDTLO.             ¦
¦ 340       MOVE SNAMEDO IN OLD-ACCTREC TO SNAMEDO IN ACCTDTLO.           ¦
¦ 341       MOVE FNAMEDO IN OLD-ACCTREC TO FNAMEDO IN ACCTDTLO.           ¦
¦ 342       MOVE MIDO IN OLD-ACCTREC TO MIDO IN ACCTDTLO.                 ¦
¦ 343       MOVE TTLDO IN OLD-ACCTREC TO TTLDO IN ACCTDTLO.               ¦
¦ 344       MOVE TELDO IN OLD-ACCTREC TO TELDO IN ACCTDTLO.               ¦
¦ 345       MOVE ADDR1DO IN OLD-ACCTREC TO ADDR1DO IN ACCTDTLO.           ¦
¦ 346       MOVE ADDR2DO IN OLD-ACCTREC TO ADDR2DO IN ACCTDTLO.           ¦
¦ 347       MOVE ADDR3DO IN OLD-ACCTREC TO ADDR3DO IN ACCTDTLO.           ¦
¦ 348       MOVE AUTH1DO IN OLD-ACCTREC TO AUTH1DO IN ACCTDTLO.           ¦
¦ 349       MOVE AUTH2DO IN OLD-ACCTREC TO AUTH2DO IN ACCTDTLO.           ¦
¦ 350       MOVE AUTH3DO IN OLD-ACCTREC TO AUTH3DO IN ACCTDTLO.           ¦
¦ 351       MOVE AUTH4DO IN OLD-ACCTREC TO AUTH4DO IN ACCTDTLO.           ¦
¦ 352       MOVE CARDSDO IN OLD-ACCTREC TO  CARDSDO IN ACCTDTLO.          ¦
¦ 353       MOVE IMODO IN OLD-ACCTREC TO IMODO IN ACCTDTLO.               ¦
¦ 354       MOVE IDAYDO IN OLD-ACCTREC TO IDAYDO IN ACCTDTLO.             ¦
¦ 355       MOVE IYRDO IN OLD-ACCTREC TO IYRDO IN ACCTDTLO.               ¦
¦ 356       MOVE RSNDO IN OLD-ACCTREC TO RSNDO IN ACCTDTLO.               ¦
¦ 357       MOVE CCODEDO IN OLD-ACCTREC TO CCODEDO IN ACCTDTLO.           ¦
```

```
¦ 358       MOVE APPRDO IN OLD-ACCTREC TO APPRDO IN ACCTDTLO.            ¦
¦ 359       MOVE SCODE1DO IN OLD-ACCTREC TO SCODE1DO IN ACCTDTLO.        ¦
¦ 360       MOVE SCODE2DO IN OLD-ACCTREC TO SCODE2DO IN ACCTDTLO.        ¦
¦ 361       MOVE SCODE3DO IN OLD-ACCTREC TO SCODE3DO IN ACCTDTLO.        ¦
¦ 362       MOVE STATDO IN OLD-ACCTREC TO STATDO IN ACCTDTLO.            ¦
¦ 363       MOVE LIMITDO IN OLD-ACCTREC TO LIMITDO IN ACCTDTLO.          ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 339 through 363: For deletions and modifications, we build an image of the old version of the record in the map area.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 364       IF REQC = 'X' MOVE 'DELETION' TO TITLEDO, GO TO LOG-2.       ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Line 364: For deletions, the title **DELETION** is added and then control goes to Line 424 (LOG-2), where the update particulars are added.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 365       MOVE 'BEFORE CHANGE' TO TITLEDO.                             ¦
¦ 366       IF SNAMEDO IN OLD-ACCTREC NOT = SNAMEDO IN NEW-ACCTREC OR    ¦
¦ 367           FNAMEDO IN OLD-ACCTREC NOT = FNAMEDO IN NEW-ACCTREC      ¦
¦ 368           OR MIDO IN OLD-ACCTREC NOT = MIDO IN NEW-ACCTREC OR      ¦
¦ 369           TTLDO IN OLD-ACCTREC NOT = TTLDO IN NEW-ACCTREC          ¦
¦ 370           MOVE 'NAME' TO MOD-NAME.                                 ¦
¦ 371       IF TELDO IN OLD-ACCTREC NOT = TELDO IN NEW-ACCTREC           ¦
¦ 372           MOVE 'TEL' TO MOD-TELE.                                  ¦
¦ 373       IF ADDR1DO IN OLD-ACCTREC NOT = ADDR1DO IN NEW-ACCTREC OR    ¦
¦ 374           ADDR2DO IN OLD-ACCTREC NOT = ADDR2DO IN NEW-ACCTREC OR   ¦
¦ 375           ADDR3DO IN OLD-ACCTREC NOT = ADDR3DO IN NEW-ACCTREC      ¦
¦ 376           MOVE 'ADDR' TO MOD-ADDR.                                 ¦
¦ 377       IF AUTH1DO IN OLD-ACCTREC NOT = AUTH1DO IN NEW-ACCTREC OR    ¦
¦ 378           AUTH2DO IN OLD-ACCTREC NOT = AUTH2DO IN NEW-ACCTREC OR   ¦
¦ 379           AUTH3DO IN OLD-ACCTREC NOT = AUTH3DO IN NEW-ACCTREC OR   ¦
¦ 380           AUTH4DO IN OLD-ACCTREC NOT = AUTH4DO IN NEW-ACCTREC      ¦
¦ 381           MOVE 'AUTH' TO MOD-AUTH.                                 ¦
¦ 382       IF CARDSDO IN OLD-ACCTREC NOT = CARDSDO IN NEW-ACCTREC OR    ¦
¦ 383           IMODO IN OLD-ACCTREC NOT = IMODO IN NEW-ACCTREC OR       ¦
¦ 384           IDAYDO IN OLD-ACCTREC NOT = IDAYDO IN NEW-ACCTREC OR     ¦
¦ 385           IYRDO IN OLD-ACCTREC NOT = IYRDO IN NEW-ACCTREC OR       ¦
¦ 386           RSNDO IN OLD-ACCTREC NOT = RSNDO IN NEW-ACCTREC OR       ¦
¦ 387           CCODEDO IN OLD-ACCTREC NOT = CCODEDO IN NEW-ACCTREC OR   ¦
¦ 388           APPRDO IN OLD-ACCTREC NOT = APPRDO IN NEW-ACCTREC        ¦
¦ 389           MOVE 'CARD' TO MOD-CARD.                                 ¦
¦ 390       IF SCODE1DO IN OLD-ACCTREC NOT = SCODE1DO IN NEW-ACCTREC OR  ¦
¦ 391           SCODE2DO IN OLD-ACCTREC NOT = SCODE2DO IN NEW-ACCTREC OR ¦
¦ 392           SCODE3DO IN OLD-ACCTREC NOT = SCODE3DO IN NEW-ACCTREC    ¦
¦ 393           MOVE 'CODES' TO MOD-CODE.                                ¦
¦ 394       MOVE MOD-LINE TO MSGDO.                                      ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 365 through 394: For modifications only, **BEFORE CHANGE** is placed in the title line. Then the fields in the old record are compared to those in the new record in logical groups, so that we can note the areas of the record that got changed. The changes are listed in the message area of the map, using an abbreviated form of the group name. For this purpose, the fields are grouped as follows: name (with the four component fields of surname, first name, middle initial, and title), telephone, address (the four address lines), authorized users (four), card issue information (seven fields), and special codes (three).

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 395       EXEC CICS WRITEQ TS QUEUE('ACCTLOG') FROM(ACCTDTLO)          ¦
¦ 396           LENGTH(DTL-LNG) END-EXEC.                               ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 395 through 396: The next process in handling modifications is to write the old-record image, which we've just finished building, to the temporary storage queue that represents the log. We named this **ACCTLOG.** The length specified is the length of the symbolic map description in working storage (available from the compiler in the DMAP output).

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 397       MOVE 'AFTER CHANGE' TO TITLEDO.                               ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 397: Finally, the title line in the map area is changed from **BEFORE CHANGE** to **AFTER CHANGE** in preparation for building an image of the new record to put into the log.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 398   LOG-1.                                                           ¦
¦ 399       MOVE ACCTDO IN NEW-ACCTREC TO ACCTDO IN ACCTDTLO.            ¦
¦ 400       MOVE SNAMEDO IN NEW-ACCTREC TO SNAMEDO IN ACCTDTLO.          ¦
¦ 401       MOVE FNAMEDO IN NEW-ACCTREC TO FNAMEDO IN ACCTDTLO.          ¦
¦ 402       MOVE MIDO IN NEW-ACCTREC TO MIDO IN ACCTDTLO.                ¦
¦ 403       MOVE TTLDO IN NEW-ACCTREC TO TTLDO IN ACCTDTLO.              ¦
¦ 404       MOVE TELDO IN NEW-ACCTREC TO TELDO IN ACCTDTLO.              ¦
¦ 405       MOVE ADDR1DO IN NEW-ACCTREC TO ADDR1DO IN ACCTDTLO.          ¦
¦ 406       MOVE ADDR2DO IN NEW-ACCTREC TO ADDR2DO IN ACCTDTLO.          ¦
¦ 407       MOVE ADDR3DO IN NEW-ACCTREC TO ADDR3DO IN ACCTDTLO.          ¦
¦ 408       MOVE AUTH1DO IN NEW-ACCTREC TO AUTH1DO IN ACCTDTLO.          ¦
¦ 409       MOVE AUTH2DO IN NEW-ACCTREC TO AUTH2DO IN ACCTDTLO.          ¦
¦ 410       MOVE AUTH3DO IN NEW-ACCTREC TO AUTH3DO IN ACCTDTLO.          ¦
¦ 411       MOVE AUTH4DO IN NEW-ACCTREC TO AUTH4DO IN ACCTDTLO.          ¦
¦ 412       MOVE CARDSDO IN NEW-ACCTREC TO  CARDSDO IN ACCTDTLO.         ¦
¦ 413       MOVE IMODO IN NEW-ACCTREC TO IMODO IN ACCTDTLO.              ¦
¦ 414       MOVE IDAYDO IN NEW-ACCTREC TO IDAYDO IN ACCTDTLO.            ¦
¦ 415       MOVE IYRDO IN NEW-ACCTREC TO IYRDO IN ACCTDTLO.              ¦
¦ 416       MOVE RSNDO IN NEW-ACCTREC TO RSNDO IN ACCTDTLO.              ¦
¦ 417       MOVE CCODEDO IN NEW-ACCTREC TO CCODEDO IN ACCTDTLO.          ¦
¦ 418       MOVE APPRDO IN NEW-ACCTREC TO APPRDO IN ACCTDTLO.            ¦
¦ 419       MOVE SCODE1DO IN NEW-ACCTREC TO SCODE1DO IN ACCTDTLO.        ¦
¦ 420       MOVE SCODE2DO IN NEW-ACCTREC TO SCODE2DO IN ACCTDTLO.        ¦
¦ 421       MOVE SCODE3DO IN NEW-ACCTREC TO SCODE3DO IN ACCTDTLO.        ¦
¦ 422       MOVE STATDO IN NEW-ACCTREC TO STATDO IN ACCTDTLO.            ¦
¦ 423       MOVE LIMITDO IN NEW-ACCTREC TO LIMITDO IN ACCTDTLO.          ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 398 through 423 (LOG-1): For modifications and additions, an image of the new version of the record is completed in the map area, from the fields in the new file record.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 424   LOG-2.                                                           ¦
¦ 424       MOVE EIBTRMID TO UPDT-TERM, MOVE EIBTIME TO UPDT-TIME,       ¦
¦ 426       MOVE EIBDATE TO UPDT-DATE, MOVE UPDT-LINE TO MSGDO.          ¦
¦                                                                         ¦
```

```
+------------------------------------------------------------------------+
```

Lines 424 through 426 (LOG-2): At this point, we're ready to write to the log, either:

The second ("after") image, for modifications, o
The only image, for additions and deletions

We add information about the update to the message area of the map: specifically, the time and date of the update and the terminal at which the entry was made.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 427       EXEC CICS WRITEQ TS QUEUE('ACCTLOG') FROM(ACCTDTLO)          ¦
¦ 428           LENGTH(DTL-LNG) END-EXEC.                                ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 427 through 428: The last step in the logging process is to write this image to temporary storage; this command is identical to that in Lines 395-396.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 429   CK-EXIT.                                                        ¦
¦ 430       EXIT.                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 429 through 430: This marks the end of the checking procedures referred to in lines 324 through 332, and control is returned to CICS.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 431   UPDTE SECTION.                                                  ¦
¦ 432  *    UPDATE THE FILES FOR ADD REQUESTS.                           ¦
¦ 433       IF REQC = 'A' GO TO UPDT-ADD.                                ¦
¦ 434       IF REQC = 'X' GO TO UPDT-DELETE.                             ¦
¦ 435       IF REQC = 'M' GO TO UPDT-MODIFY.                             ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 433 through 435: These statements begin the updating of the files. Additions, modifications and deletions are handled separately in the code, starting at Lines 436, 445, and 466 respectively.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 436   UPDT-ADD.                                                       ¦
¦ 437       MOVE 2 TO MENU-MSGNO.                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 436 through 437 (UPDT-ADD): The first step in adding a new record is to move a confirmation message to the message area of the menu map that will be displayed when the update is finished.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 438       EXEC CICS WRITE FILE('ACCTFIL') FROM(NEW-ACCTREC)            ¦
¦ 439           RIDFLD(ACCTC) LENGTH(ACCT-LNG) END-EXEC.                 ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Lines 438 through 439: Next, the new record is added to the account file.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 440       EXEC CICS WRITE FILE('ACCTIX') FROM(NEW-ACIXREC)            ¦
¦ 441           RIDFLD(SNAMEDO IN NEW-ACIXREC) LENGTH(ACIX-LNG) END-EXEC.¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 440 through 441: Then the index record corresponding to the new account file record is added to the index file.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 442       GO TO UPDT-EXIT.                                            ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Line 442: The updates for additions are complete at this point and control goes to **RELEASE-ACCT** (Line 140), where we give up exclusive control of the new account number.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 443  *                                                                ¦
¦ 444  *    UPDATE THE FILES FOR MODIFY REQUESTS.                       ¦
¦ 445   UPDT-MODIFY.                                                    ¦
¦ 446       MOVE 3 TO MENU-MSGNO.                                       ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 443 through 446 (UPDT-MODIFY): The next 18 lines update the files for modifications. As with an addition, the first step is to move the appropriate confirmation message to the message area of the menu map, ready for the next output display.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 447       EXEC CICS REWRITE FILE('ACCTFIL') FROM(NEW-ACCTREC)         ¦
¦ 448           LENGTH (ACCT-LNG) END-EXEC.                             ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 447 through 448: Then we replace the old version of the account record with the new version in the account file.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 449       IF SNAMEDO IN NEW-ACCTREC NOT = SNAMEDO IN OLD-ACCTREC      ¦
¦ 450           EXEC CICS DELETE FILE('ACCTIX') RIDFLD(OLD-IXKEY)       ¦
¦ 451               END-EXEC.                                           ¦
¦ 452           EXEC CICS WRITE FILE('ACCTIX') FROM (NEW-ACIXREC)       ¦
¦ 453               RIDFLD (SNAMEDO IN NEW-ACIXREC) LENGTH(ACIX-LNG)    ¦
¦ 454               END-EXEC.                                           ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 449 through 454: Next we find what has to be done to the corresponding record in the index file. If the surname has changed, then the key of the index record has also changed, and we must delete the old index record and add a new one. Lines 449-450 do the deletion, using the key generated in Lines 120-121. The addition follows immediately, in a command that's the same as the one we used in Lines 440-441 for processing an addition. The index record was built at Lines 306-313 in preparation for this step.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 455       ELSE IF FNAMEDO IN NEW-ACCTREC NOT = FNAMEDO IN OLD-ACCTREC  ¦
¦ 456           OR MIDO IN NEW-ACCTREC NOT = MIDO IN OLD-ACCTREC OR      ¦
¦ 457           TTLDO IN NEW-ACCTREC NOT = TTLDO IN OLD-ACCTREC OR       ¦
¦ 458           ADDR1DO IN NEW-ACCTREC NOT = ADDR1DO IN OLD-ACCTREC      ¦
¦ 459           EXEC CICS READ FILE('ACCTIX') INTO (OLD-ACIXREC)         ¦
¦ 460               RIDFLD(OLD-IXKEY) LENGTH(ACIX-LNG) UPDATE END-EXEC.  ¦
¦ 461           EXEC CICS REWRITE FILE('ACCTIX') FROM(NEW-ACIXREC)       ¦
¦ 462               LENGTH(ACIX-LNG) END-EXEC.                           ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Lines 455 through 462: Even if the surname hasn't changed, we may still need to update the index file, because one of the fields in the index record may have been changed in the modification. So we compare all the fields that appear in the index record in the new and old versions of the account record. If any field has changed, we rewrite the index record (in Lines 460-461). Before we do this, however, we have to read this record (Lines 458-459), because CICS doesn't let you rewrite without first reading the same record for update.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 463       GO TO UPDT-EXIT.                                          ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Line 463: The file updates for modifications are complete at this point, and control goes to **RELEASE-ACCT,** just as it does for additions.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
¦ 464  *                                                             ¦
¦ 465  *    UPDATE THE FILES FOR DELETE REQUESTS.                    ¦
¦ 466   UPDT-DELETE.                                                 ¦
¦ 467       MOVE 4 TO MENU-MSGNO.                                     ¦
¦ 468       EXEC CICS DELETE FILE('ACCTFIL') END-EXEC.                ¦
¦ 469       EXEC CICS DELETE FILE('ACCTIX') RIDFLD(OLD-IXKEY)         ¦
¦ 470           END-EXEC.                                             ¦
¦                                                                     ¦
+---------------------------------------------------------------------+
```

Lines 466 through 470 (UPDT-DELETE): As with the other types of updates, processing a deletion begins by moving the appropriate confirmation message to the menu map. Then the account record and the associated index record are deleted from the account and index files, respectively. Note that we specify a key for the index file (the **RIDFLD(OLD-IXKEY)** option), but not for the account file. This is because we've read the account record for update (in Lines 117-118), but we haven't read the index record. In contrast to the situation with the **REWRITE** command, CICS allows you to use the **DELETE** command without first doing a read-for-update.

This was our bug! Our original version of Line 468 read:

EXEC CICS DELETE FILE('ACCTFIL') ***RIDFLD(ACCTC)*** END-EXEC.

You'll see the problems this gave us when you move on to "Testing and diagnosis" in topic 5.0 where we show you a session with the Execution Diagnostic Facility, EDF.

```
+---------------------------------------------------------------------+
¦                                                                     ¦
```

```
¦ 471   UPDT-EXIT.                                                        ¦
¦ 472       EXIT.                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Lines 471 to 472: This marks the end of the updating procedures referred to in lines 442 and 463, and control is returned to CICS.

4.4 Program ACCT03: requests for printing

```
001   IDENTIFICATION DIVISION.
002   PROGRAM-ID. ACCT03.
003  *REMARKS. THIS PROGRAM IS THE FIRST INVOKED BY TRANSACTIONS
004  *         'AC03', 'ACLG', 'AC05', 'ACEL' AND 'AC06'. 'AC03'
005  *         COMPLETES A REQUEST FOR PRINTING OF A CUSTOMER
006  *         RECORD, WHICH WAS PROCESSED INITIALLY BY TRANSACTION
007  *         'AC01'.  'ACLG,' WHICH IS A USER REQUEST TO PRINT THE
008  *         LOG MERELY REQUESTS 'AC05' BE STARTED WHEN THE LOG
009  *         PRINTER ('L86O') IS AVAILABLE. 'AC05' TRANSFERS THE
010  *         LOG DATA FROM TEMPORARY STORAGE TO THE PRINTER.
011  *         'ACEL,' WHICH IS A USER REQUEST TO PRINT THE ERROR
012  *         LOG MERELY REQUESTS 'AC06' BE STARTED WHEN THE LOG
013  *         PRINTER ('L86O') IS AVAILABLE. 'AC06' TRANSFERS THE
014  *         ERROR LOG DATA FROM TEMPORARY STORAGE TO THE PRINTER.
015   ENVIRONMENT DIVISION.
016   DATA DIVISION.
017   WORKING STORAGE SECTION.
018   01  MISC.
019       02  RESPONSE               PIC S9(8) COMP.
020   01  COMMAREA-FOR-ACCT04.
021       02  ERR-PGM                PIC X(8) VALUE 'ACCT03'.
022       02  ERR-FN                 PIC X.
023       02  ERR-RCODE              PIC X.
024       02  ERR-COMMAND            PIC XX.
025       02  ERR-RESP               PIC 99.
026   01  TS-LNG                     PIC S9(4) COMP VALUE +751.
027   01  TS-ELNG                    PIC S9(4) COMP VALUE +156.
```

Lines 17 through 27: These lines are the working storage area of the program. Individual variables will be explained in the comments below as they are used.

```
028       COPY ACCTSET.
```

Line 28: This line brings in a copy of the symbolic description map structure.

```
029  *
030   PROCEDURE DIVISION.
031  *
032  *    INITIALIZE.
033   INIT.
034  *    CATER FOR UNEXPECTED ERRORS.
035       EXEC CICS HANDLE CONDITION ERROR(NO-GOOD) END-EXEC.
```

Lines 29 through 35 (INIT): This command corresponds in function to the **HANDLE CONDITION ERROR** commands early in programs **ACCT01** (Line 143) and **ACCT02** (Line 103). It tells CICS where to transfer control if there's an unusual response to any other CICS command.

```
036  *                                                              
037  *    TEST FOR TRANSACTION TYPE.                                
038       IF EIBTRNID = 'AC03' GO TO AC03.                         
039       IF EIBTRNID = 'ACLG' GO TO ACLG.                         
040       IF EIBTRNID = 'AC05' GO TO AC05.                         
041       IF EIBTRNID = 'ACEL' GO TO ACEL.                         
042       IF EIBTRNID = 'AC06' GO TO AC06.                         
```

Lines 36 through 42: This program performs five independent functions, each of which is invoked by a different transaction code. We could have used separate programs, but each would be so short that we've chosen instead to combine them. In these statements we look at the transaction identifier to find out which function we want on this execution of the program.

```
043  *                                                              
044  *    PROCESS TRANSACTION 'AC03'.                               
045   AC03.                                                        
046       EXEC CICS RETRIEVE INTO(ACCTDTLI) LENGTH(TS-LNG) END-EXEC.
```

Lines 44 through 46 (AC03): The three lines beginning here complete the processing of a request to print a particular account file record. Processing for this type of request begins in transaction **AC01,** which reads the record from the file, assembles the information into symbolic map form, and then invokes this transaction to finish the work. (See the discussion at Lines 372-373 of program **ACCT01**.)

The first step here, therefore, is to get access to the symbolic map data prepared in transaction **AC01.** We use a **RETRIEVE** command to do this, bringing the data into working storage, where we've defined the same symbolic map structure. We have to specify the maximum length of data to be retrieved in this command, and for this we use a variable, defined at Line 26, and initialized to the length of the map. (We found this length from the **DMAP** section of the COBOL compiler output.)

```
047       EXEC CICS SEND MAP('ACCTDTL') MAPSET('ACCTSET') PRINT    
048           ERASE END-EXEC.                                      
```

Lines 47 through 48: Because the **START** command that invoked this transaction named a terminal, we know that we can write to that terminal directly, just as we send output back to the input terminal in other transactions. This command writes the map prepared in **AC01** to this terminal, which is the printer that the user named in the original print request. We specify the **PRINT** option here to emphasize that it is required in either the command or the map if the terminal to which you are writing is a printer. (Strictly speaking, we could omit it here because we did include it in the map definition.) We also use **ERASE,** to remove any information in the device buffer left over from the previous print operation.

```
049       GO TO RTRN.                                              
```

Line 49: The printing is now complete and we return to CICS at Line 103 (**RTRN**).

We should note here that some of the work done in the **AC01** part of processing a print request--reading the file and arranging the data into map format--could have been done in this one instead. There are two reasons for doing it where we did:

1. We cannot check the input properly without reading the file record (to ensure that it really exists). We have to do this in transaction **AC01,** because afterward there's no way to send an error message back to the user (**AC03** doesn't have access to the input terminal).

2. There may be a delay between the **AC01** and **AC03** halves of the request, while CICS waits for the requested printer to become free. During this time the record may be modified, so that the user will see a copy that includes different information from what existed at the time of his or her request.

   It's true that this information will be more current than what was requested, but it is probably not what the user wants. Worse, the record could get deleted in this interval. Then the **AC03** part of the request would fail, and there would be no way to tell the user what had happened (short of starting still another transaction to send a message back to the original input terminal, in the hope that the same user would still be there).

```
050  *
051  *    PROCESS TRANSACTION 'ACLG'.
052   ACLG.
053       EXEC CICS START TRANSID('AC05') TERMID('L86O') END-EXEC.
```

Lines 50 through 53 (ACLG): The six lines beginning here perform the second function of this program, which is to process the first part of a request to print the log. Like a request to print an individual record, a request to print the log has to be handled as two transactions, because two different terminals are involved: the one that enters the request, and the printer.

This code processes the request from an input terminal. The first step is to ask CICS to start up the second transaction (**AC05**) as soon as the printer used for the log is free (this printer happens to be named **L86O**). This **START** command does this; there's no data for this transaction to forward to the next one, and so we have no **FROM** option.

```
054       MOVE LOW-VALUES TO ACCTMSGO.
055       MOVE 'PRINTING OF LOG HAS BEEN SCHEDULED' TO MSGO.
056       EXEC CICS SEND MAP('ACCTMSG') MAPSET('ACCTSET')
057           FREEKB END-EXEC.
```

Lines 54 through 57: As usual, we clear the map to nulls (**LOW-VALUES**) before filling it in.

Then we send a message back to the requesting terminal, confirming that the requested work has been scheduled. In contrast to all the other types of requests that make up this application, a request to print the log

isn't entered through the menu screen. So it isn't appropriate to use the message area of the menu screen, and we need a separate map to send this message. This map is **ACCTMSG,** which is simply a one-line map consisting of an area for a message.

```
058       GO TO RTRN.
```

Line 58: Then we return to CICS at Line 103.

```
059  *
060  *    PROCESS TRANSACTION 'AC05'.
061   AC05.
062       EXEC CICS READQ TS QUEUE('ACCTLOG') INTO (ACCTDTLI)
063           LENGTH(TS-LNG) NEXT RESP(RESPONSE) END-EXEC.
064       IF RESPONSE = DFHRESP(QIDERR)
065           GO TO RTRN.
066       IF RESPONSE = DFHRESP(ITEMERR)
067           GO TO LOG-END.
068       IF RESPONSE NOT = DFHRESP(NORMAL)
069           GO TO NO-GOOD.
070       EXEC CICS SEND MAP('ACCTDTL') MAPSET('ACCTSET') PRINT ERASE
071           END-EXEC.
072       GO TO AC05.
```

Lines 59 through 72 (AC05): The code for handling the second half of a request to print the transaction log is the next function of this program. It begins with a two-command loop, in which the first command (Lines 62-63) reads the next item from the temporary storage queue (**ACCTLOG**) that represents the log. The second command sends this item, which is a map already formatted by program **ACCT01** (at Lines 304-353), to the printer. These two commands are repeated until all of the entries in the log have been printed.

Since this is the only transaction that reads this temporary storage queue, we can omit the **ITEM** number option and code the **NEXT** option in reading it. Then the first time **READQ** is executed, the first item on the log is fetched; the next time, the second item is fetched, and so on until the end of the queue. (The end of the queue results in the **ITEMERR** condition, at which control goes to **LOG-END**--as explained in the next paragraph.)

We explicitly test **RESP** for three error conditions: if a **QIDERR** has occurred (meaning that there is no queue corresponding to the one that has been entered), control will go to **RTRN** (Line 103). If an **ITEMERR** condition has occurred (meaning that we're trying to read an item that isn't there), this transfers control to **LOG-END** at Line 74. If there is some other sort of error (any **NOT NORMAL** condition), control will go to **NO-GOOD** at Line 107.

```
073   LOG-END.
074       EXEC CICS DELETEQ TS QUEUE('ACCTLOG') END-EXEC.
075       GO TO RTRN.
```

Lines 73 through 75 (LOG-END): Having printed the log, we now delete the temporary storage queue in which it was collected, to free up temporary storage.

Note: The next *write* to this deleted queue will cause CICS to create a new queue, with the same name, and the accumulation of change log records will begin all over again.

```
076  *
077  *    PROCESS TRANSACTION 'ACEL'.
078   ACEL.
079       EXEC CICS START TRANSID('AC06') TERMID('L86O') END-EXEC.
080       MOVE LOW-VALUES TO ACCTMSGO.
081       MOVE 'PRINTING OF ERROR LOG HAS BEEN SCHEDULED' TO MSGO.
082       EXEC CICS SEND MAP('ACCTMSG') MAPSET('ACCTSET')
083           FREEKB END-EXEC.
```

Lines 76 through 83 (ACEL): We now issue the **START** command to initiate the third transaction that will do the printing of the error log. The name of this transaction is **AC06.** Like transaction **ACLG,** a request to print the error log has to be handled as two transactions, because two different terminals are involved: the one that enters the request, and the printer. Because we didn't specify any **TIME** or **INTERVAL** parameter, CICS will start the transaction as soon as it can after the required terminal is free.

As usual, we clear the map to nulls (**LOW-VALUES**) before filling it in.

Then we send a message back to the requesting terminal, confirming that the requested work has been scheduled. Like the request to print the transaction log, a request to print the error log isn't entered through the menu screen. So it isn't appropriate to use the message area of the menu screen, and we need a separate map to send this message. This map is **ACCTMSG,** which is simply a one-line map consisting of an area for a message.

```
084       GO TO RTRN.
```

Line 84: Then we return to CICS at Line 103.

```
085  *
086  *    PROCESS TRANSACTION 'AC06'.
087   AC06.
088       EXEC CICS READQ TS QUEUE('ACERLOG') INTO (ACCTERRI)
089           LENGTH(TS-ELNG) NEXT RESP(RESPONSE) END-EXEC.
090       IF RESPONSE = DFHRESP(QIDERR)
091           GO TO RTRN.
092       IF RESPONSE = DFHRESP(ITEMERR)
093           GO TO ELOG-END.
094       IF RESPONSE NOT = DFHRESP(NORMAL)
095           GO TO NO-GOOD.
096       EXEC CICS SEND MAP('ACCTERR') MAPSET('ACCTSET') PRINT ERASE
097           END-EXEC.
098       GO TO AC06.
```

Lines 85 through 98 (AC06): This is very similar to **AC05,** described in lines 59 through 72. It begins with a two-command loop, in which the first command (Lines 88-89) reads the next item from the temporary storage queue (**ACERLOG**) that represents the error log. The second command sends this item, which is a map already formatted by program **ACCT01** (at Lines 304-353), to the printer. These two commands are repeated until all of the entries in the error log have been printed.

Since this is the only transaction that reads this temporary storage queue, we can omit the **ITEM** number option and code the **NEXT** option in reading it. Then the first time **READQ** is executed, the first item on the log is fetched; the next time, the second item is fetched, and so on until the end of the queue. (The end of the queue results in the **ITEMERR** condition, at which control goes to **ELOG-END**.)

```
099   ELOG-END.
100        EXEC CICS DELETEQ TS QUEUE('ACERLOG') END-EXEC.
```

Lines 99 through 100 (ELOG-END): Having printed the error log, we now delete the temporary storage queue in which it was collected, to free up temporary storage.

Note: The next *write* to this deleted queue will cause CICS to create a new queue, with the same name, and the accumulation of error log records will begin all over again.

```
101  *
102  *    RETURN TO CICS.
103   RTRN.
104        EXEC CICS RETURN END-EXEC.
```

Lines 101 through 104 (RTRN): This command returns control to CICS and is shared by all the functions in the program. Notice that no next transid is set. The concept of next transid doesn't apply, of course, to terminals that don't get input.

This explains its absence for the first and third transactions, because the terminal associated with the transaction is a printer. In the case of the **ACLG** code, it is because the request is entered directly, and not through the menu. Once again, since we're not controlling the contents of the screen, we don't know what transaction the user will want next.

```
105  *
106  *    PROCESS UNRECOVERABLE ERRORS.
107   NO-GOOD.
108        MOVE EIBFN TO ERR-FN, MOVE EIBRCODE TO ERR-RCODE.
109        MOVE EIBFN TO ERR-COMMAND, MOVE EIBRESP TO ERR-RESP.
110        EXEC CICS HANDLE CONDITION ERROR END-EXEC.
111        EXEC CICS LINK PROGRAM('ACCT04')
112            COMMAREA(COMMAREA-FOR-ACCT04) LENGTH(14) END-EXEC.
```

Lines 105 through 112 (NO-GOOD): This code handles unrecoverable errors on CICS commands and is identical to the corresponding code in Lines 410-413 of program **ACCT01.**

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ 113       GOBACK.                                                       ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Line 113: This **GOBACK** has the same function as those that terminate the other programs.

4.5 Program ACCT04: error processing

```
001   IDENTIFICATION DIVISION.
002   PROGRAM-ID. ACCT04.
003  *REMARKS. THIS PROGRAM IS A GENERAL PURPOSE ERROR ROUTINE.
004  *         CONTROL IS TRANSFERRED TO IT BY OTHER PROGRAMS IN THE
005  *         ONLINE ACCOUNT FILE APPLICATION WHEN AN UNRECOVERABLE
006  *         ERROR HAS OCCURRED.
007  *         IT SENDS A MESSAGE TO INPUT TERMINAL DESCRIBING THE
008  *         TYPE OF ERROR AND ASKS THE OPERATOR TO REPORT IT.
009  *         THEN IT ABENDS, SO THAT ANY UPDATES MADE IN THE
010  *         UNCOMPLETED TRANSACTION ARE BACKED OUT AND SO THAT AN
011  *         ABEND DUMP IS AVAILABLE.
012   ENVIRONMENT DIVISION.
013   DATA DIVISION.
014   WORKING STORAGE SECTION.
015       COPY ACCTSET.
016    01  RESPTAB.
017        02  RESP01             PIC X(12)     VALUE 'ERROR'.
018        02  RESP02             PIC X(12)     VALUE 'RDATT'.
019        02  RESP03             PIC X(12)     VALUE 'WRBRK'.
020        02  RESP04             PIC X(12)     VALUE 'EOF'.
021        02  RESP05             PIC X(12)     VALUE 'EODS'.
022        02  RESP06             PIC X(12)     VALUE 'EOC'.
023        02  RESP07             PIC X(12)     VALUE 'INBFMH'.
024        02  RESP08             PIC X(12)     VALUE 'ENDINPT'.
025        02  RESP09             PIC X(12)     VALUE 'NONVAL'.
026        02  RESP10             PIC X(12)     VALUE 'NOSTART'.
027        02  RESP11             PIC X(12)     VALUE 'TERMIDERR'.
028        02  RESP12             PIC X(12)     VALUE 'FILENOTFOUND'.
029        02  RESP13             PIC X(12)     VALUE 'NOTFND'.
030        02  RESP14             PIC X(12)     VALUE 'DUPREC'.
031        02  RESP15             PIC X(12)     VALUE 'DUPKEY'.
032        02  RESP16             PIC X(12)     VALUE 'INVREQ'.
033        02  RESP17             PIC X(12)     VALUE 'IOERR'.
034        02  RESP18             PIC X(12)     VALUE 'NOSPACE'.
035        02  RESP19             PIC X(12)     VALUE 'NOTOPEN'.
036        02  RESP20             PIC X(12)     VALUE 'ENDFILE'.
037        02  RESP21             PIC X(12)     VALUE 'ILLOGIC'.
038        02  RESP22             PIC X(12)     VALUE 'LENGERR'.
039        02  RESP23             PIC X(12)     VALUE 'QZERO'.
040        02  RESP24             PIC X(12)     VALUE 'SIGNAL'.
041        02  RESP25             PIC X(12)     VALUE 'QBUSY'.
042        02  RESP26             PIC X(12)     VALUE 'ITEMERR'.
043        02  RESP27             PIC X(12)     VALUE 'PGMIDERR'.
044        02  RESP28             PIC X(12)     VALUE 'TRANSIDERR'.
045        02  RESP29             PIC X(12)     VALUE 'ENDDATA'.
046        02  RESP30             PIC X(12)     VALUE 'INVTSREQ'.
047        02  RESP31             PIC X(12)     VALUE 'EXPIRED'.
048        02  RESP32             PIC X(12)     VALUE 'RETPAGE'.
049        02  RESP33             PIC X(12)     VALUE 'RTEFAIL'.
```

```
050        02  RESP34             PIC X(12)     VALUE 'RTESOME'.
051        02  RESP35             PIC X(12)     VALUE 'TSIOERR'.
052        02  RESP36             PIC X(12)     VALUE 'MAPFAIL'.
053        02  RESP37             PIC X(12)     VALUE 'INVERRTERM'.
054        02  RESP38             PIC X(12)     VALUE 'INVMPSZ'.
055        02  RESP39             PIC X(12)     VALUE 'IGREQID'.
056        02  RESP40             PIC X(12)     VALUE 'OVERFLOW'.
057        02  RESP41             PIC X(12)     VALUE 'INVLDC'.
```

```
 058        02  RESP42              PIC X(12)     VALUE 'NOSTG'.
 059        02  RESP43              PIC X(12)     VALUE 'JIDERR'.
 060        02  RESP44              PIC X(12)     VALUE 'QIDERR'.
 061        02  RESP45              PIC X(12)     VALUE 'NOJBUFSP'.
 062        02  RESP46              PIC X(12)     VALUE 'DSSTAT'.
 063        02  RESP47              PIC X(12)     VALUE 'SELNERR'.
 064        02  RESP48              PIC X(12)     VALUE 'FUNCERR'.
 065        02  RESP49              PIC X(12)     VALUE 'UNEXPIN'.
 066        02  RESP50              PIC X(12)     VALUE 'NOPASSBKRD'.
 067        02  RESP51              PIC X(12)     VALUE 'NOPASSBKWR'.
 068        02  RESP52              PIC X(12)     VALUE '*NOT VALID*'.
 069        02  RESP53              PIC X(12)     VALUE 'SYSIDERR'.
 070        02  RESP54              PIC X(12)     VALUE 'ISCINVREQ'.
 071        02  RESP55              PIC X(12)     VALUE 'ENQBUSY'.
 072        02  RESP56              PIC X(12)     VALUE 'ENVDEFERR'.
 073        02  RESP57              PIC X(12)     VALUE 'IGREQCD'.
 074        02  RESP58              PIC X(12)     VALUE 'SESSIONERR'.
 075        02  RESP59              PIC X(12)     VALUE 'SYSBUSY'.
 076        02  RESP60              PIC X(12)     VALUE 'SESSBUSY'.
 077        02  RESP61              PIC X(12)     VALUE 'NOTALLOC'.
 078        02  RESP62              PIC X(12)     VALUE 'CBIDERR'.
 079        02  RESP63              PIC X(12)     VALUE 'INVEXITREQ'.
 080        02  RESP64              PIC X(12)     VALUE 'INVPARTNSET'.
 081        02  RESP65              PIC X(12)     VALUE 'INVPARTN'.
 082        02  RESP66              PIC X(12)     VALUE 'PARTNFAIL'.
 083        02  RESP67              PIC X(12)     VALUE '*NOT VALID*'.
 084        02  RESP68              PIC X(12)     VALUE '*NOT VALID*'.
 085        02  RESP69              PIC X(12)     VALUE 'USERIDERR'.
 086        02  RESP70              PIC X(12)     VALUE 'NOTAUTH'.
 087        02  RESP71              PIC X(12)     VALUE 'VOLIDERR'.
 088        02  RESP72              PIC X(12)     VALUE 'SUPPRESSED'.
 089        02  RESP73              PIC X(12)     VALUE '*NOT VALID*'.
 090        02  RESP74              PIC X(12)     VALUE '*NOT VALID*'.
 091        02  RESP75              PIC X(12)     VALUE 'RESIDERR'.
 092        02  RESP76              PIC X(12)     VALUE '*NOT VALID*'.
 093        02  RESP77              PIC X(12)     VALUE '*NOT VALID*'.
 094        02  RESP78              PIC X(12)     VALUE '*NOT VALID*'.
 095        02  RESP79              PIC X(12)     VALUE '*NOT VALID*'.
 096        02  RESP80              PIC X(12)     VALUE 'NOSPOOL'.
 097        02  RESP81              PIC X(12)     VALUE 'TERMERR'.
 098        02  RESP82              PIC X(12)     VALUE 'ROLLEDBACK'.
 099        02  RESP83              PIC X(12)     VALUE 'END'.
 100        02  RESP84              PIC X(12)     VALUE 'DISABLED'.
 101        02  RESP85              PIC X(12)     VALUE 'ALLOCERR'.
 102        02  RESP86              PIC X(12)     VALUE 'STRELERR'.
```

```
 103        02  RESP87              PIC X(12)     VALUE 'OPENERR'.
 104        02  RESP88              PIC X(12)     VALUE 'SPOLBUSY'.
 105        02  RESP89              PIC X(12)     VALUE 'SPOLERR'.
 106        02  RESP90              PIC X(12)     VALUE 'NODEIDER'.
 107        02  RESP91              PIC X(12)     VALUE 'TASKIDERR'.
 108        02  RESP92              PIC X(12)     VALUE 'TCIDERR'.
 109        02  RESP93              PIC X(12)     VALUE 'DSNNOTFOUND'.
 110        02  RESP-NOT-FOUND      PIC X(12)     VALUE '*NOT VALID*'.
 111    01  FILLER                  REDEFINES RESPTAB.
 112        02  RESPVAL             OCCURS 94
 113                                PIC X(12).
 114    01  COMMAND-LIST.
 115        02  HEX-0202        PIC XX VALUE '  '.
 116        02  HEX-0204        PIC XX VALUE '  '.
 117        02  HEX-0206        PIC XX VALUE '  '.
```

```
¦ 118        02  HEX-0208        PIC XX VALUE '  '.                      ¦
¦ 119        02  HEX-020A        PIC XX VALUE '  '.                      ¦
¦ 120        02  HEX-020C        PIC XX VALUE '  '.                      ¦
¦ 121        02  HEX-020E        PIC XX VALUE '  '.                      ¦
¦ 122        02  HEX-0210        PIC XX VALUE '  '.                      ¦
¦ 123        02  HEX-0402        PIC XX VALUE '  '.                      ¦
¦ 124        02  HEX-0404        PIC XX VALUE '  '.                      ¦
¦ 125        02  HEX-0406        PIC XX VALUE '  '.                      ¦
¦ 126        02  HEX-0408        PIC XX VALUE '  '.                      ¦
¦ 127        02  HEX-040A        PIC XX VALUE '  '.                      ¦
¦ 128        02  HEX-040C        PIC XX VALUE '  '.                      ¦
¦ 129        02  HEX-040E        PIC XX VALUE '  '.                      ¦
¦ 130        02  HEX-0410        PIC XX VALUE '  '.                      ¦
¦ 131        02  HEX-0412        PIC XX VALUE '  '.                      ¦
¦ 132        02  HEX-0414        PIC XX VALUE '  '.                      ¦
¦ 133        02  HEX-0416        PIC XX VALUE '  '.                      ¦
¦ 134        02  HEX-0418        PIC XX VALUE '  '.                      ¦
¦ 135        02  HEX-041A        PIC XX VALUE '  '.                      ¦
¦ 136        02  HEX-041C        PIC XX VALUE '  '.                      ¦
¦ 137        02  HEX-041E        PIC XX VALUE '  '.                      ¦
¦ 138        02  HEX-0420        PIC XX VALUE '  '.                      ¦
¦ 139        02  HEX-0422        PIC XX VALUE '  '.                      ¦
¦ 140        02  HEX-0424        PIC XX VALUE '  '.                      ¦
¦ 141        02  HEX-0426        PIC XX VALUE '  '.                      ¦
¦ 142        02  HEX-0428        PIC XX VALUE '  '.                      ¦
¦ 143        02  HEX-042A        PIC XX VALUE '  '.                      ¦
¦ 144        02  HEX-042C        PIC XX VALUE '  '.                      ¦
¦ 145        02  HEX-042E        PIC XX VALUE '  '.                      ¦
¦ 146        02  HEX-0430        PIC XX VALUE '  '.                      ¦
¦ 147        02  HEX-0432        PIC XX VALUE '  '.                      ¦
¦ 148        02  HEX-0434        PIC XX VALUE '  '.                      ¦
¦ 149        02  HEX-0436        PIC XX VALUE '  '.                      ¦
¦ 150        02  HEX-0438        PIC XX VALUE '  '.                      ¦
¦ 151        02  HEX-043A        PIC XX VALUE '  '.                      ¦
¦ 152        02  HEX-043C        PIC XX VALUE '  '.                      ¦
¦ 153        02  HEX-0602        PIC XX VALUE '  '.                      ¦
¦ 154        02  HEX-0604        PIC XX VALUE '  '.                      ¦
¦ 155        02  HEX-0606        PIC XX VALUE '  '.                      ¦
¦                                                                        ¦
+------------------------------------------------------------------------+

+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 156        02  HEX-0608        PIC XX VALUE '  '.                      ¦
¦ 157        02  HEX-060A        PIC XX VALUE '  '.                      ¦
¦ 158        02  HEX-060C        PIC XX VALUE '  '.                      ¦
¦ 159        02  HEX-060E        PIC XX VALUE '  '.                      ¦
¦ 160        02  HEX-0610        PIC XX VALUE '  '.                      ¦
¦ 161        02  HEX-0612        PIC XX VALUE '  '.                      ¦
¦ 162        02  HEX-0614        PIC XX VALUE '  '.                      ¦
¦ 163        02  HEX-0802        PIC XX VALUE '  '.                      ¦
¦ 164        02  HEX-0804        PIC XX VALUE '  '.                      ¦
¦ 165        02  HEX-0806        PIC XX VALUE '  '.                      ¦
¦ 166        02  HEX-0A02        PIC XX VALUE '  '.                      ¦
¦ 167        02  HEX-0A04        PIC XX VALUE '  '.                      ¦
¦ 168        02  HEX-0A06        PIC XX VALUE '  '.                      ¦
¦ 169        02  HEX-0C02        PIC XX VALUE '  '.                      ¦
¦ 170        02  HEX-0C04        PIC XX VALUE '  '.                      ¦
¦ 171        02  HEX-0E02        PIC XX VALUE '  '.                      ¦
¦ 172        02  HEX-0E04        PIC XX VALUE '  '.                      ¦
¦ 173        02  HEX-0E06        PIC XX VALUE '  '.                      ¦
¦ 174        02  HEX-0E08        PIC XX VALUE '  '.                      ¦
¦ 175        02  HEX-0E0A        PIC XX VALUE '  '.                      ¦
¦ 176        02  HEX-0E0C        PIC XX VALUE '  '.                      ¦
¦ 177        02  HEX-0E0E        PIC XX VALUE '  '.                      ¦
```

```
¦ 178        02  HEX-1002        PIC XX VALUE '  '.                        ¦
¦ 179        02  HEX-1004        PIC XX VALUE '  '.                        ¦
¦ 180        02  HEX-1006        PIC XX VALUE '  '.                        ¦
¦ 181        02  HEX-1008        PIC XX VALUE '  '.                        ¦
¦ 182        02  HEX-100A        PIC XX VALUE '  '.                        ¦
¦ 183        02  HEX-100C        PIC XX VALUE '  '.                        ¦
¦ 184        02  HEX-1202        PIC XX VALUE '  '.                        ¦
¦ 185        02  HEX-1204        PIC XX VALUE '  '.                        ¦
¦ 186        02  HEX-1206        PIC XX VALUE '  '.                        ¦
¦ 187        02  HEX-1208        PIC XX VALUE '  '.                        ¦
¦ 188        02  HEX-1402        PIC XX VALUE '  '.                        ¦
¦ 189        02  HEX-1404        PIC XX VALUE '  '.                        ¦
¦ 190        02  HEX-1602        PIC XX VALUE '  '.                        ¦
¦ 191        02  HEX-1604        PIC XX VALUE '  '.                        ¦
¦ 192        02  HEX-1802        PIC XX VALUE '  '.                        ¦
¦ 193        02  HEX-1804        PIC XX VALUE '  '.                        ¦
¦ 194        02  HEX-1806        PIC XX VALUE '  '.                        ¦
¦ 195        02  HEX-1808        PIC XX VALUE '  '.                        ¦
¦ 196        02  HEX-180A        PIC XX VALUE '  '.                        ¦
¦ 197        02  HEX-180C        PIC XX VALUE '  '.                        ¦
¦ 198        02  HEX-180E        PIC XX VALUE '  '.                        ¦
¦ 199        02  HEX-1810        PIC XX VALUE '  '.                        ¦
¦ 200        02  HEX-1812        PIC XX VALUE '  '.                        ¦
¦ 201        02  HEX-1A02        PIC XX VALUE '  '.                        ¦
¦ 202        02  HEX-1A04        PIC XX VALUE '  '.                        ¦
¦ 203        02  HEX-1C02        PIC XX VALUE '  '.                        ¦
¦ 204        02  HEX-1E02        PIC XX VALUE '  '.                        ¦
¦ 204        02  HEX-1E04        PIC XX VALUE '  '.                        ¦
¦ 205        02  HEX-1E06        PIC XX VALUE '  '.                        ¦
¦ 207        02  HEX-1E08        PIC XX VALUE '  '.                        ¦
¦ 208        02  HEX-1E0A        PIC XX VALUE '  '.                        ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+

+--------------------------------------------------------------------------+
¦                                                                          ¦
¦ 209        02  HEX-1E0C        PIC XX VALUE '  '.                        ¦
¦ 210        02  HEX-1E0E        PIC XX VALUE '  '.                        ¦
¦ 211        02  HEX-1E10        PIC XX VALUE '  '.                        ¦
¦ 212        02  HEX-1E12        PIC XX VALUE '  '.                        ¦
¦ 213        02  HEX-1E14        PIC XX VALUE '  '.                        ¦
¦ 214        02  HEX-2002        PIC XX VALUE '  '.                        ¦
¦ 215        02  HEX-2202        PIC XX VALUE '  '.                        ¦
¦ 216        02  HEX-2204        PIC XX VALUE '  '.                        ¦
¦ 217        02  HEX-2206        PIC XX VALUE '  '.                        ¦
¦ 218        02  HEX-4802        PIC XX VALUE '  '.                        ¦
¦ 219        02  HEX-4804        PIC XX VALUE '  '.                        ¦
¦ 220        02  HEX-4A02        PIC XX VALUE '  '.                        ¦
¦ 221        02  HEX-4A04        PIC XX VALUE '  '.                        ¦
¦ 222        02  HEX-4C02        PIC XX VALUE '  '.                        ¦
¦ 223        02  HEX-4C04        PIC XX VALUE '  '.                        ¦
¦ 224        02  HEX-4E02        PIC XX VALUE '  '.                        ¦
¦ 225        02  HEX-4E04        PIC XX VALUE '  '.                        ¦
¦ 226        02  HEX-5002        PIC XX VALUE '  '.                        ¦
¦ 227        02  HEX-5004        PIC XX VALUE '  '.                        ¦
¦ 228        02  HEX-5202        PIC XX VALUE '  '.                        ¦
¦ 229        02  HEX-5204        PIC XX VALUE '  '.                        ¦
¦ 230        02  HEX-5206        PIC XX VALUE '  '.                        ¦
¦ 231        02  HEX-5402        PIC XX VALUE '  '.                        ¦
¦ 232        02  HEX-5404        PIC XX VALUE '  '.                        ¦
¦ 233        02  HEX-5602        PIC XX VALUE '  '.                        ¦
¦ 234        02  HEX-5604        PIC XX VALUE '  '.                        ¦
¦ 235        02  HEX-5606        PIC XX VALUE '  '.                        ¦
¦ 236        02  HEX-5610        PIC XX VALUE '  '.                        ¦
¦ 237        02  HEX-5802        PIC XX VALUE '  '.                        ¦
```

```
¦ 238        02  HEX-5804        PIC XX VALUE '  '.                      ¦
¦ 239        02  HEX-5A02        PIC XX VALUE '  '.                      ¦
¦ 240        02  HEX-5A04        PIC XX VALUE '  '.                      ¦
¦ 241        02  HEX-5C02        PIC XX VALUE '  '.                      ¦
¦ 242        02  HEX-5C04        PIC XX VALUE '  '.                      ¦
¦ 243        02  HEX-5E02        PIC XX VALUE '  '.                      ¦
¦ 244        02  HEX-5E04        PIC XX VALUE '  '.                      ¦
¦ 245        02  HEX-5E06        PIC XX VALUE '  '.                      ¦
¦ 246        02  HEX-5E12        PIC XX VALUE '  '.                      ¦
¦ 247        02  HEX-5E14        PIC XX VALUE '  '.                      ¦
¦ 248        02  HEX-6002        PIC XX VALUE '  '.                      ¦
¦ 249        02  HEX-6004        PIC XX VALUE '  '.                      ¦
¦ 250        02  HEX-6202        PIC XX VALUE '  '.                      ¦
¦ 251        02  HEX-6204        PIC XX VALUE '  '.                      ¦
¦ 252        02  HEX-6402        PIC XX VALUE '  '.                      ¦
¦ 253        02  HEX-6602        PIC XX VALUE '  '.                      ¦
¦ 254        02  HEX-6604        PIC XX VALUE '  '.                      ¦
¦ 255        02  HEX-6612        PIC XX VALUE '  '.                      ¦
¦ 256        02  HEX-6614        PIC XX VALUE '  '.                      ¦
¦ 257        02  HEX-6622        PIC XX VALUE '  '.                      ¦
¦ 258        02  HEX-6624        PIC XX VALUE '  '.                      ¦
¦ 269        02  HEX-6802        PIC XX VALUE '  '.                      ¦
¦ 260        02  HEX-6804        PIC XX VALUE '  '.                      ¦
¦ 261        02  HEX-6812        PIC XX VALUE '  '.                      ¦
¦                                                                        ¦
+------------------------------------------------------------------------+

+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 262        02  HEX-6814        PIC XX VALUE '  '.                      ¦
¦ 263        02  HEX-6A02        PIC XX VALUE '  '.                      ¦
¦ 264        02  HEX-6C02        PIC XX VALUE '  '.                      ¦
¦ 265        02  HEX-6C12        PIC XX VALUE '  '.                      ¦
¦ 266        02  HEX-6E02        PIC XX VALUE '  '.                      ¦
¦ 267        02  HEX-6E04        PIC XX VALUE '  '.                      ¦
¦ 268        02  HEX-7002        PIC XX VALUE '  '.                      ¦
¦ 269        02  HEX-7004        PIC XX VALUE '  '.                      ¦
¦ 270        02  HEX-7006        PIC XX VALUE '  '.                      ¦
¦ 271        02  HEX-7008        PIC XX VALUE '  '.                      ¦
¦ 272        02  HEX-7012        PIC XX VALUE '  '.                      ¦
¦ 273        02  HEX-7014        PIC XX VALUE '  '.                      ¦
¦ 274        02  HEX-7202        PIC XX VALUE '  '.                      ¦
¦ 275        02  HEX-7402        PIC XX VALUE '  '.                      ¦
¦ 276        02  HEX-7404        PIC XX VALUE '  '.                      ¦
¦ 277        02  HEX-7602        PIC XX VALUE '  '.                      ¦
¦ 278        02  HEX-7802        PIC XX VALUE '  '.                      ¦
¦ 279        02  HEX-7804        PIC XX VALUE '  '.                      ¦
¦ 280        02  HEX-7812        PIC XX VALUE '  '.                      ¦
¦ 281        02  HEX-7814        PIC XX VALUE '  '.                      ¦
¦ 282        02  HEX-7822        PIC XX VALUE '  '.                      ¦
¦ 283        02  HEX-7824        PIC XX VALUE '  '.                      ¦
¦ 284        02  HEX-7A02        PIC XX VALUE '  '.                      ¦
¦ 285        02  HEX-7A04        PIC XX VALUE '  '.                      ¦
¦ 286        02  HEX-7E02        PIC XX VALUE '  '.                      ¦
¦ 287        02  HEX-7E04        PIC XX VALUE '  '.                      ¦
¦ 288        02  HEX-MISC        PIC XX VALUE '  '.                      ¦
¦ 289    01  FILLER              REDEFINES COMMAND-LIST.                 ¦
¦ 290        02  HEX-COMMAND     PIC X(2) OCCURS 174.                    ¦
¦ 291    01  COMMAND-NAMES.                                              ¦
¦ 292        02  NAME-0202       PIC X(20) VALUE 'ADDRESS'.              ¦
¦ 293        02  NAME-0204       PIC X(20) VALUE 'HANDLE CONDITION'.     ¦
¦ 294        02  NAME-0206       PIC X(20) VALUE 'HANDLE AID'.           ¦
¦ 295        02  NAME-0208       PIC X(20) VALUE 'ASSIGN'.               ¦
¦ 296        02  NAME-020A       PIC X(20) VALUE 'IGNORE CONDITION'.     ¦
¦ 297        02  NAME-020C       PIC X(20) VALUE 'PUSH'.                 ¦
```

```
 298        02  NAME-020E       PIC X(20) VALUE 'POP'.
 299        02  NAME-0210       PIC X(20) VALUE 'ADDRESS SET'.
 300        02  NAME-0402       PIC X(20) VALUE 'RECEIVE'.
 301        02  NAME-0404       PIC X(20) VALUE 'SEND'.
 302        02  NAME-0406       PIC X(20) VALUE 'CONVERSE'.
 303        02  NAME-0408       PIC X(20) VALUE 'ISSUE EODS'.
 304        02  NAME-040A       PIC X(20) VALUE 'ISSUE COPY'.
 305        02  NAME-040C       PIC X(20) VALUE 'WAIT TERMINAL'.
 306        02  NAME-040E       PIC X(20) VALUE 'ISSUE LOAD'.
 307        02  NAME-0410       PIC X(20) VALUE 'WAIT SIGNAL'.
 308        02  NAME-0412       PIC X(20) VALUE 'ISSUE RESET'.
 309        02  NAME-0414       PIC X(20) VALUE 'ISSUE DISCONNECT'.
 310        02  NAME-0416       PIC X(20) VALUE 'ISSUE ENDOUTPUT'.
 311        02  NAME-0418       PIC X(20) VALUE 'ISSUE ERASEUP'.
 312        02  NAME-041A       PIC X(20) VALUE 'ISSUE ENDFILE'.
 313        02  NAME-041C       PIC X(20) VALUE 'ISSUE PRINT'.
 314        02  NAME-041E       PIC X(20) VALUE 'ISSUE SIGNAL'.
```

```
 315        02  NAME-0420       PIC X(20) VALUE 'ALLOCATE'.
 316        02  NAME-0422       PIC X(20) VALUE 'FREE'.
 317        02  NAME-0424       PIC X(20) VALUE 'POINT'.
 318        02  NAME-0426       PIC X(20) VALUE 'BUILD ATTACH'.
 319        02  NAME-0428       PIC X(20) VALUE 'EXTRACT ATTACH'.
 320        02  NAME-042A       PIC X(20) VALUE 'EXTRACT TCT'.
 321        02  NAME-042C       PIC X(20) VALUE 'WAIT CONVID'.
 322        02  NAME-042E       PIC X(20) VALUE 'EXTRACT PROCESS'.
 323        02  NAME-0430       PIC X(20) VALUE 'ISSUE ABEND'.
 324        02  NAME-0432       PIC X(20) VALUE 'CONNECT PROCESS'.
 325        02  NAME-0434       PIC X(20) VALUE 'ISSUE CONFIRMATION'.
 326        02  NAME-0436       PIC X(20) VALUE 'ISSUE ERROR'.
 327        02  NAME-0438       PIC X(20) VALUE 'ISSUE PREPARE'.
 328        02  NAME-043A       PIC X(20) VALUE 'ISSUE PASS'.
 329        02  NAME-043C       PIC X(20) VALUE 'EXTRACT LOGONMSG'.
 330        02  NAME-0602       PIC X(20) VALUE 'READ'.
 331        02  NAME-0604       PIC X(20) VALUE 'WRITE'.
 332        02  NAME-0606       PIC X(20) VALUE 'REWRITE'.
 333        02  NAME-0608       PIC X(20) VALUE 'DELETE'.
 334        02  NAME-060A       PIC X(20) VALUE 'UNLOCK'.
 335        02  NAME-060C       PIC X(20) VALUE 'STARTBR'.
 336        02  NAME-060E       PIC X(20) VALUE 'READNEXT'.
 337        02  NAME-0610       PIC X(20) VALUE 'READPREV'.
 338        02  NAME-0612       PIC X(20) VALUE 'ENDBR'.
 339        02  NAME-0614       PIC X(20) VALUE 'RESETBR'.
 340        02  NAME-0802       PIC X(20) VALUE 'WRITEQ TD'.
 341        02  NAME-0804       PIC X(20) VALUE 'READQ TD'.
 342        02  NAME-0806       PIC X(20) VALUE 'DELETEQ TD'.
 343        02  NAME-0A02       PIC X(20) VALUE 'WRITEQ TS'.
 344        02  NAME-0A04       PIC X(20) VALUE 'READQ TS'.
 345        02  NAME-0A06       PIC X(20) VALUE 'DELETEQ TS'.
 346        02  NAME-0C02       PIC X(20) VALUE 'GETMAIN'.
 347        02  NAME-0C04       PIC X(20) VALUE 'FREEMAIN'.
 348        02  NAME-0E02       PIC X(20) VALUE 'LINK'.
 349        02  NAME-0E04       PIC X(20) VALUE 'XCTL'.
 350        02  NAME-0E06       PIC X(20) VALUE 'LOAD'.
 351        02  NAME-0E08       PIC X(20) VALUE 'RETURN'.
 352        02  NAME-0E0A       PIC X(20) VALUE 'RELEASE'.
 353        02  NAME-0E0C       PIC X(20) VALUE 'ABEND'.
 354        02  NAME-0E0E       PIC X(20) VALUE 'HANDLE ABEND'.
 355        02  NAME-1002       PIC X(20) VALUE 'ASKTIME'.
 356        02  NAME-1004       PIC X(20) VALUE 'DELAY'.
 357        02  NAME-1006       PIC X(20) VALUE 'POST'.
```

```
¦ 358        02  NAME-1008       PIC X(20) VALUE 'START'.                ¦
¦ 359        02  NAME-100A       PIC X(20) VALUE 'RETRIEVE'.             ¦
¦ 360        02  NAME-100C       PIC X(20) VALUE 'CANCEL'.               ¦
¦ 361        02  NAME-1202       PIC X(20) VALUE 'WAIT EVENT'.           ¦
¦ 362        02  NAME-1204       PIC X(20) VALUE 'ENQ'.                  ¦
¦ 363        02  NAME-1206       PIC X(20) VALUE 'DEQ'.                  ¦
¦ 364        02  NAME-1208       PIC X(20) VALUE 'SUSPEND'.              ¦
¦ 365        02  NAME-1402       PIC X(20) VALUE 'JOURNAL'.              ¦
¦ 366        02  NAME-1404       PIC X(20) VALUE 'WAIT JOURNAL'.         ¦
¦                                                                        ¦
+------------------------------------------------------------------------+

+------------------------------------------------------------------------+
¦                                                                        ¦
¦ 367        02  NAME-1602       PIC X(20) VALUE 'SYNCPOINT'.            ¦
¦ 368        02  NAME-1604       PIC X(20) VALUE 'RESYNC'.               ¦
¦ 369        02  NAME-1802       PIC X(20) VALUE 'RECEIVE MAP'.          ¦
¦ 370        02  NAME-1804       PIC X(20) VALUE 'SEND MAP'.             ¦
¦ 371        02  NAME-1806       PIC X(20) VALUE 'SEND TEXT'.            ¦
¦ 372        02  NAME-1808       PIC X(20) VALUE 'SEND PAGE'.            ¦
¦ 373        02  NAME-180A       PIC X(20) VALUE 'PURGE MESSAGE'.        ¦
¦ 374        02  NAME-180C       PIC X(20) VALUE 'ROUTE'.                ¦
¦ 375        02  NAME-180E       PIC X(20) VALUE 'RECEIVE PARTN'.        ¦
¦ 376        02  NAME-1810       PIC X(20) VALUE 'SEND PARTNSET'.        ¦
¦ 377        02  NAME-1812       PIC X(20) VALUE 'SEND CONTROL'.         ¦
¦ 378        02  NAME-1A02       PIC X(20) VALUE 'TRACE ON/OFF'.         ¦
¦ 379        02  NAME-1A04       PIC X(20) VALUE 'ENTER TRACEID'.        ¦
¦ 380        02  NAME-1C02       PIC X(20) VALUE 'DUMP'.                 ¦
¦ 381        02  NAME-1E02       PIC X(20) VALUE 'ISSUE ADD'.            ¦
¦ 382        02  NAME-1E04       PIC X(20) VALUE 'ISSUE ERASE'.          ¦
¦ 383        02  NAME-1E06       PIC X(20) VALUE 'ISSUE REPLACE'.        ¦
¦ 384        02  NAME-1E08       PIC X(20) VALUE 'ISSUE ABORT'.          ¦
¦ 385        02  NAME-1E0A       PIC X(20) VALUE 'ISSUE QUERY'.          ¦
¦ 386        02  NAME-1E0C       PIC X(20) VALUE 'ISSUE END'.            ¦
¦ 387        02  NAME-1E0E       PIC X(20) VALUE 'ISSUE RECEIVE'.        ¦
¦ 388        02  NAME-1E10       PIC X(20) VALUE 'ISSUE NOTE'.           ¦
¦ 389        02  NAME-1E12       PIC X(20) VALUE 'ISSUE WAIT'.           ¦
¦ 390        02  NAME-1E14       PIC X(20) VALUE 'ISSUE SEND'.           ¦
¦ 391        02  NAME-2002       PIC X(20) VALUE 'BIF DEEDIT'.           ¦
¦ 392        02  NAME-2202       PIC X(20) VALUE 'ENABLE'.               ¦
¦ 393        02  NAME-2204       PIC X(20) VALUE 'DISABLE'.              ¦
¦ 394        02  NAME-2206       PIC X(20) VALUE 'EXTRACT EXIT'.         ¦
¦ 395        02  NAME-4802       PIC X(20) VALUE 'ENTER TRACENUM'.       ¦
¦ 396        02  NAME-4804       PIC X(20) VALUE 'MONITOR POINT'.        ¦
¦ 397        02  NAME-4A02       PIC X(20) VALUE 'ASKTIME ABSTIME'.      ¦
¦ 398        02  NAME-4A04       PIC X(20) VALUE 'FORMATTIME'.           ¦
¦ 399        02  NAME-4C02       PIC X(20) VALUE 'INQUIRE FILE'.         ¦
¦ 400        02  NAME-4C04       PIC X(20) VALUE 'SET FILE'.             ¦
¦ 401        02  NAME-4E02       PIC X(20) VALUE 'INQUIRE PROGRAM'.      ¦
¦ 402        02  NAME-4E04       PIC X(20) VALUE 'SET PROGRAM'.          ¦
¦ 403        02  NAME-5002       PIC X(20) VALUE 'INQUIRE TRANSACTION'.  ¦
¦ 404        02  NAME-5004       PIC X(20) VALUE 'SET TRANSACTION'.      ¦
¦ 405        02  NAME-5202       PIC X(20) VALUE 'INQUIRE TERMINAL'.     ¦
¦ 406        02  NAME-5204       PIC X(20) VALUE 'SET TERMINAL'.         ¦
¦ 407        02  NAME-5206       PIC X(20) VALUE 'INQUIRE NETNAME'.      ¦
¦ 408        02  NAME-5402       PIC X(20) VALUE 'INQUIRE SYSTEM'.       ¦
¦ 409        02  NAME-5404       PIC X(20) VALUE 'SET SYSTEM'.           ¦
¦ 410        02  NAME-5602       PIC X(20) VALUE 'SPOOLOPEN'.            ¦
¦ 411        02  NAME-5604       PIC X(20) VALUE 'SPOOLREAD'.            ¦
¦ 412        02  NAME-5606       PIC X(20) VALUE 'SPOOLWRITE'.           ¦
¦ 413        02  NAME-5610       PIC X(20) VALUE 'SPOOLCLOSE'.           ¦
¦ 414        02  NAME-5802       PIC X(20) VALUE 'INQUIRE CONNECTION'.   ¦
¦ 415        02  NAME-5804       PIC X(20) VALUE 'SET CONNECTION'.       ¦
¦ 416        02  NAME-5A02       PIC X(20) VALUE 'INQUIRE MODENAME'.     ¦
¦ 417        02  NAME-5A04       PIC X(20) VALUE 'SET MODENAME'.         ¦
```

```
 418        02  NAME-5C02       PIC X(20) VALUE 'INQUIRE TDQUEUE'.
 419        02  NAME-5C04       PIC X(20) VALUE 'SET TDQUEUE'.
```

```
 420        02  NAME-5E02       PIC X(20) VALUE 'INQUIRE TASK'.
 421        02  NAME-5E04       PIC X(20) VALUE 'SET TASK'.
 422        02  NAME-5E06       PIC X(20) VALUE 'CHANGE TASK'.
 423        02  NAME-5E12       PIC X(20) VALUE 'INQUIRE TCLASS'.
 424        02  NAME-5E14       PIC X(20) VALUE 'SET TCLASS'.
 425        02  NAME-6002       PIC X(20) VALUE 'INQUIRE JOURNALNUM'.
 426        02  NAME-6004       PIC X(20) VALUE 'SET JOURNALNUM'.
 427        02  NAME-6202       PIC X(20) VALUE 'INQUIRE VOLUME'.
 428        02  NAME-6204       PIC X(20) VALUE 'SET VOLUME'.
 429        02  NAME-6402       PIC X(20) VALUE 'PERFORM SECURITY'.
 430        02  NAME-6602       PIC X(20) VALUE 'INQUIRE DUMPDS'.
 431        02  NAME-6604       PIC X(20) VALUE 'SET DUMPDS'.
 432        02  NAME-6612       PIC X(20) VALUE 'INQUIRE TRANDUMPCODE'.
 433        02  NAME-6614       PIC X(20) VALUE 'SET TRANDUMPCODE'.
 434        02  NAME-6622       PIC X(20) VALUE 'INQUIRE SYSDUMPCODE'.
 435        02  NAME-6624       PIC X(20) VALUE 'SET SYSDUMPCODE'.
 436        02  NAME-6802       PIC X(20) VALUE 'INQUIRE VTAM'.
 437        02  NAME-6804       PIC X(20) VALUE 'SET VTAM'.
 438        02  NAME-6812       PIC X(20) VALUE 'INQUIRE AUTOINSTALL'.
 439        02  NAME-6814       PIC X(20) VALUE 'SET AUTOINSTALL'.
 440        02  NAME-6A02       PIC X(20) VALUE 'QUERY SECURITY'.
 441        02  NAME-6C02       PIC X(20) VALUE 'WRITE OPERATOR'.
 442        02  NAME-6C12       PIC X(20) VALUE 'CICSMESSAGE'.
 443        02  NAME-6E02       PIC X(20) VALUE 'INQUIRE IRC'.
 444        02  NAME-6E04       PIC X(20) VALUE 'SET IRC'.
 445        02  NAME-7002       PIC X(20) VALUE 'INQUIRE STATISTICS'.
 446        02  NAME-7004       PIC X(20) VALUE 'SET STATISTICS'.
 447        02  NAME-7006       PIC X(20) VALUE 'PERFORM STATISTICS'.
 448        02  NAME-7008       PIC X(20) VALUE 'COLLECT STATISTICS'.
 449        02  NAME-7012       PIC X(20) VALUE 'INQUIRE MONITOR'.
 450        02  NAME-7014       PIC X(20) VALUE 'SET MONITOR'.
 451        02  NAME-7202       PIC X(20) VALUE 'PERFORM RESETTIME'.
 452        02  NAME-7402       PIC X(20) VALUE 'SIGNON'.
 453        02  NAME-7404       PIC X(20) VALUE 'SIGNOFF'.
 454        02  NAME-7602       PIC X(20) VALUE 'PERFORM SHUTDOWN'.
 455        02  NAME-7802       PIC X(20) VALUE 'INQUIRE TRACEDEST'.
 456        02  NAME-7804       PIC X(20) VALUE 'SET TRACEDEST'.
 457        02  NAME-7812       PIC X(20) VALUE 'INQUIRE TRACEFLAG'.
 458        02  NAME-7814       PIC X(20) VALUE 'SET TRACEFLAG'.
 459        02  NAME-7822       PIC X(20) VALUE 'INQUIRE TRACETYPE'.
 460        02  NAME-7824       PIC X(20) VALUE 'SET TRACETYPE'.
 461        02  NAME-7A02       PIC X(20) VALUE 'INQUIRE DSNAME'.
 462        02  NAME-7A04       PIC X(20) VALUE 'SET DSNAME'.
 463        02  NAME-7E02       PIC X(20) VALUE 'DUMP TRANSACTION'.
 464        02  NAME-7E04       PIC X(20) VALUE 'DUMP SYSTEM'.
 465        02  NAME-0001       PIC X(20) VALUE 'UNKNOWN COMMAND'.
 466    01  FILLER              REDEFINES COMMAND-NAMES.
 467        02  COMMAND-NAME    PIC X(20) OCCURS 174.
```

```
 468   01  MISC.
 469       02  I                  PIC S9(4) COMP.
 470       02  IXR                PIC S9(4) COMP VALUE +33.
 471       02  IXC                PIC S9(4) COMP VALUE +174.
 472       02  ERR-LNG            PIC S9(4) COMP VALUE +156.
```

```
 473       02  DSN-MSG.
 474           04  FILLER          PIC X(13) VALUE 'THE FILE IS: '.
 475           04  DSN             PIC X(8).
 476           04  FILLER          PIC X VALUE '.'.
 477       02  HEX-LIST.
 478           04  HEX-0601        PIC S9(4) COMP VALUE +1537.
 479           04  HEX-0602        PIC S9(4) COMP VALUE +1538.
 480           04  HEX-0608        PIC S9(4) COMP VALUE +1544.
 481           04  HEX-060C        PIC S9(4) COMP VALUE +1548.
 482           04  HEX-060D        PIC S9(4) COMP VALUE +1549.
 483           04  HEX-060F        PIC S9(4) COMP VALUE +1551.
 484           04  HEX-0680        PIC S9(4) COMP VALUE +1664.
 485           04  HEX-0681        PIC S9(4) COMP VALUE +1665.
 486           04  HEX-0682        PIC S9(4) COMP VALUE +1666.
 487           04  HEX-0683        PIC S9(4) COMP VALUE +1667.
 488           04  HEX-06E1        PIC S9(4) COMP VALUE +1761.
 489           04  HEX-0A01        PIC S9(4) COMP VALUE +2561.
 490           04  HEX-0A02        PIC S9(4) COMP VALUE +2562.
 491           04  HEX-0A04        PIC S9(4) COMP VALUE +2564.
 492           04  HEX-0A08        PIC S9(4) COMP VALUE +2568.
 493           04  HEX-0A20        PIC S9(4) COMP VALUE +2592.
 494           04  HEX-0AE1        PIC S9(4) COMP VALUE +2785.
 495           04  HEX-0E01        PIC S9(4) COMP VALUE +3585.
 496           04  HEX-0EE1        PIC S9(4) COMP VALUE +3809.
 497           04  HEX-1001        PIC S9(4) COMP VALUE +4097.
 498           04  HEX-1004        PIC S9(4) COMP VALUE +4100.
 499           04  HEX-1011        PIC S9(4) COMP VALUE +4113.
 500           04  HEX-1012        PIC S9(4) COMP VALUE +4114.
 501           04  HEX-1014        PIC S9(4) COMP VALUE +4116.
 502           04  HEX-1081        PIC S9(4) COMP VALUE +4225.
 503           04  HEX-10E1        PIC S9(4) COMP VALUE +4321.
 504           04  HEX-10E9        PIC S9(4) COMP VALUE +4329.
 505           04  HEX-10FF        PIC S9(4) COMP VALUE +4351.
 506           04  HEX-1801        PIC S9(4) COMP VALUE +6145.
 507           04  HEX-1804        PIC S9(4) COMP VALUE +6148.
 508           04  HEX-1808        PIC S9(4) COMP VALUE +6152.
 509           04  HEX-18E1        PIC S9(4) COMP VALUE +6369.
 510           04  HEX-MISC        PIC S9(4) COMP VALUE +0001.
 511       02  HEX-CODE REDEFINES HEX-LIST PIC X(2) OCCURS 33.
 512       02  ERR-LIST.
 513           04  MSG-0601        PIC X(60) VALUE
 514   *           FILE CONTROL - FILENOTFOUND
 515               'A PROGRAM OR FCT TABLE ERROR (INVALID FILE NAME).'.
 516           04  MSG-0602        PIC X(60) VALUE
 517   *           FILE CONTROL - ILLOGIC
 518               'A PROGRAM OR FILE ERROR (VSAM ILLOGIC).'.
 519           04  MSG-0608        PIC X(60) VALUE
 520   *           FILE CONTROL - INVREQ
```

```
 521           'A PROGRAM OR FCT TABLE ERROR (INVALID FILE REQUEST).'.
 522           04  MSG-060C        PIC X(60) VALUE
 523   *           FILE CONTROL - NOTOPEN
 524               'A FILE BEING CLOSED THAT MUST BE OPEN.'.
 525           04  MSG-060D        PIC X(60) VALUE
 526   *           FILE CONTROL - DISABLED
 527               'A FILE BEING DISABLED.'.
 528           04  MSG-060F        PIC X(60) VALUE
 529   *           FILE CONTROL - ENDFILE
 530               'A PROGRAM OR FILE ERROR (UNEXPECTED END-OF-FILE).'.
 531           04  MSG-0680        PIC X(60) VALUE
 532   *           FILE CONTROL - IOERR
```

```
533              'A FILE INPUT/OUTPUT ERROR.'.
534           04  MSG-0681        PIC X(60) VALUE
535  *            FILE CONTROL - NOTFND
536              'A PROGRAM OR FILE ERROR (RECORD NOT FOUND).'.
537           04  MSG-0682        PIC X(60) VALUE
538  *            FILE CONTROL - DUPREC
539              'A PROGRAM OR FILE ERROR (DUPLICATE RECORD).'.
540           04  MSG-0683        PIC X(60) VALUE
541  *            FILE CONTROL - NOSPACE
542              'INADEQUATE SPACE IN A FILE.'.
543           04  MSG-06E1        PIC X(60) VALUE
544  *            FILE CONTROL - LENGERR
545          'A PROGRAM OR FILE ERROR (LENGTH ERROR, FILE CONTROL).'.
546           04  MSG-0A01        PIC X(60) VALUE
547  *            TEMPORARY STORAGE CONTROL - ITEMERR
548          'A PROGRAM OR TEMPORARY STORAGE ERROR (ITEM ERROR).'.
549           04  MSG-0A02        PIC X(60) VALUE
550  *            TEMPORARY STORAGE CONTROL - QIDERR
551          'A PROGRAM OR TEMPORARY STORAGE ERROR (UNKNOWN QUEUE).'.
552           04  MSG-0A04        PIC X(60) VALUE
553  *            TEMPORARY STORAGE CONTROL - IOERR
554              'AN INPUT/OUTPUT ERROR IN TEMPORARY STORAGE.'.
555           04  MSG-0A08        PIC X(60) VALUE
556  *            TEMPORARY STORAGE CONTROL - NOSPACE
557              'NO SPACE IN TEMPORARY STORAGE.'.
558           04  MSG-0A20        PIC X(60) VALUE
559  *            TEMPORARY STORAGE CONTROL - INVREQ
560          'A PROGRAM OR SYSTEM ERROR (INVALID REQUEST IN TS).'.
561           04  MSG-0AE1        PIC X(60) VALUE
562  *            TEMPORARY STORAGE CONTROL - LENGERR
563          'A PROGRAM OR TEMPORARY STORAGE ERROR (TS LENGTH ERROR)'.
564           04  MSG-0E01        PIC X(60) VALUE
565  *            PROGRAM CONTROL - PGMIDERR
566              'A PROGRAM IS NOT DEFINED TO CICS.'.
567           04  MSG-0EE0        PIC X(60) VALUE
568  *            PROGRAM CONTROL - INVREQ
569              'A PROGRAM ERROR (INVALID PROGRAM REQUEST).'.
570           04  MSG-1001        PIC X(60) VALUE
571  *            INTERVAL CONTROL - ENDDATA
572              'A PROGRAM ERROR (END OF DATA, USING IC).'.
573           04  MSG-1004        PIC X(60) VALUE
```

```
574  *            INTERVAL CONTROL - IOERR
575          'AN INPUT/OUTPUT ERROR IN TEMPORARY STORAGE (USING IC).'.
576           04  MSG-1011        PIC X(60) VALUE
577  *            INTERVAL CONTROL - TRANSIDERR
578              'A TRANSACTION IS NOT DEFINED TO CICS'.'
579           04  MSG-1012        PIC X(60) VALUE
580  *            INTERVAL CONTROL - TERMIDERR
581              'A PROGRAM OR TCT TABLE ERROR (TERMIDERR USING IC).'.
582           04  MSG-1014        PIC X(60) VALUE
583  *            INTERVAL CONTROL - INVTSREQ
584              'A PROGRAM OR SYSTEM ERROR (INVTSREQ USING IC).'.
585           04  MSG-1081        PIC X(60) VALUE
586  *            INTERVAL CONTROL - NOTFND
587              'A PROGRAM OR SYSTEM ERROR (NOT FOUND USING IC).'.
588           04  MSG-10E1        PIC X(60) VALUE
589  *            INTERVAL CONTROL - LENGERR
590          'A PROGRAM OR TEMP STORAGE ERROR (IC LENGTH ERROR).'.
591           04  MSG-10E9        PIC X(60) VALUE
592  *            INTERVAL CONTROL - ENVDEFERR
```

```
¦ 593              'A PROGRAM ERROR (ENVDEFERR USING IC).'             ¦
¦ 594          04  MSG-10FF        PIC X(60) VALUE                     ¦
¦ 595  *           INTERVAL CONTROL - INVREQ                           ¦
¦ 596              'A PROGRAM ERROR (INVALID REQUEST USING IC).'       ¦
¦ 597          04  MSG-1801        PIC X(60) VALUE                     ¦
¦ 598  *           BASIC MAPPING SUPPORT - INVREQ                      ¦
¦ 599              'A PROGRAM ERROR (BMS INVALID REQUEST).'            ¦
¦ 600          04  MSG-1804        PIC X(60) VALUE                     ¦
¦ 601  *           BASIC MAPPING SUPPORT - MAPFAIL                     ¦
¦ 602              'A PROGRAM ERROR (BMS MAPFAIL).'.                   ¦
¦ 603          04  MSG-1808        PIC X(60) VALUE                     ¦
¦ 604  *           BASIC MAPPING SUPPORT - INVMPSZ                     ¦
¦ 605              'A PROGRAM ERROR (INVALID MAP SIZE).'.              ¦
¦ 606          04  MSG-18E1        PIC X(60) VALUE                     ¦
¦ 607  *           BASIC MAPPING SUPPORT - LENGERR                     ¦
¦ 608              'A PROGRAM ERROR (BMS LENGTH ERROR).'.              ¦
¦ 609          04  MSG-MISC        PIC X(60) VALUE                     ¦
¦ 610  *           UNKNOWN ERROR                                       ¦
¦ 611              'AN UNKNOWN TYPE OF ERROR.'.                        ¦
¦ 612      02  ERR-MSG REDEFINES ERR-LIST PIC X(60) OCCURS 33.        ¦
¦                                                                      ¦
+----------------------------------------------------------------------+
```

Lines 14 through 612: These lines are the **WORKING STORAGE** of the program. We explain individual variables as we use them in the comments that follow. Most of them, of course, are response names, command values and names, our **HEX-LIST** of error codes, and our error messages.

```
+----------------------------------------------------------------------+
¦                                                                      ¦
¦ 613   LINKAGE SECTION.                                              ¦
¦ 614   01  DFHCOMMAREA.                                              ¦
¦ 615       02  ERR-PGRMID          PIC X(8).                         ¦
¦ 616       02  ERR-CODE.                                             ¦
¦ 617           04  ERR-FN          PIC X.                            ¦
¦ 618           04  ERR-RCODE       PIC X.                            ¦
¦ 619       02  ERR-COMMAND         PIC XX.                           ¦
¦ 620       02  ERR-RESP            PIC 99.                           ¦
¦                                                                      ¦
+----------------------------------------------------------------------+
```

Lines 613 through 620: The structure defined here and named **DFHCOMMAREA** describes the data passed to this program by means of **COMMAREA.**

```
+----------------------------------------------------------------------+
¦                                                                      ¦
¦ 621   PROCEDURE DIVISION.                                           ¦
¦ 622       MOVE LOW-VALUES TO ACCTERRO.                               ¦
¦                                                                      ¦
+----------------------------------------------------------------------+
```

Lines 621 through 622: We initialize the symbolic map structure to nulls (**LOW-VALUES**) as usual, ready for building the output map.

```
+----------------------------------------------------------------------+
¦                                                                      ¦
¦ 623       MOVE EIBTRNID TO TRANEO.                                   ¦
¦                                                                      ¦
+----------------------------------------------------------------------+
```

Line 623: Next, we move the code that identifies the failed transaction into the output map. This identifier is in the **EIB** at **EIBTRNID.** Unlike **EIBFN** and **EIBRCODE,** which change every time a command is executed, **EIBTRNID** remains the same throughout the course of the transaction, and so it will still be intact.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 624       MOVE ERR-PGRMID TO PGMEO.                                   ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Line 624: We move the name of the program in which the error was detected to the output map. Like the function and the response codes, this item of information was passed in the **COMMAREA** from the program that linked to this one.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 625       PERFORM REASON-LOOKUP THROUGH REASON-END                     ¦
¦ 626           VARYING I FROM 1 BY 1 UNTIL I NOT < IXR.                ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 625 through 626 and 640 (REASON-LOOKUP): This loop finds the entry in the table named **HEX-LIST** that matches the error that has occurred. An error is defined by the type of command that failed (file commands, temporary storage commands, and so on) in combination with a specific unusual result (such as a length error, or record not found). At the time of the error, CICS stores the type of command in the first byte of **EIBFN** (the second byte indicates the particular command of a command type).

The response is saved in **EIBRCODE,** which is a six-byte field, the first byte of which indicates the type of unusual response. You may remember that these two values were saved by the program that linked to this one, and that they were passed along in the **COMMAREA.** Program **ACCT01,** for example, defines them at Lines 42-43, saves them at Line 401, and passes them to this program in Lines 412-413. They are defined at Lines 617-618 of this program in the **COMMAREA** passed to it.

HEX-LIST consists of all the combinations of these two values that might occur on the commands that are used in this application (and included in the Primer). Since both items are encoded one-byte hexadecimal values, our table consists of two-byte combinations of hexadecimal values. And since COBOL does not allow hexadecimal expressions, we've converted each combination to its decimal equivalent in order to define the table. (You can accomplish the same thing with **CHARACTER** definitions and multiple punches, but multiple punches are very tricky if you are developing programs online.)

The names in the table still reflect the hexadecimal values, however. **HEX-0601** (Line 478) means a command (function) code of X'06' in combination with a response code of X'01'; the conversion to decimal of X'0601' is 1537. A function code of 06 happens to be a file command, and a response code of 01 for that function means **FILENOTFOUND** (file name error). You'll find all the function codes and response codes listed in the CICS/ESA Application Programming Reference.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 627       MOVE ERR-MSG (IXR) TO RSNEO.                                ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Line 627: Once the proper combination of command and response has been found in **HEX-LIST,** we move the text message that describes that situation to the map that will notify the user of the error. The right message is in the corresponding position of a second table, **ERR-LIST,** as the matching entry is in **HEX-LIST.** These messages are also named to reflect the error condition to which they apply; that is, **MSG-0601** (Line 513) corresponds to

HEX-0601 (Line 478), and so on.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 628       IF IXR < 12 MOVE EIBDS TO DSN,                              ¦
¦ 629           MOVE DSN-MSG TO FILEEO.                                 ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 628 through 629: If the command that failed was a file command, there is one additional piece of information that we want to convey to the user, and that's the name of the file on which the error occurred. These two lines do that. The file errors are the first ten in the table, and the name of the file most recently used is at **EIBDS**. (This value is also unchanged since the error occurred, because no file commands have been executed since then.)

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 630       PERFORM COMMAND-LOOKUP THROUGH COMMAND-END                   ¦
¦ 631           VARYING I FROM 1 BY 1 UNTIL I NOT < IXC.                ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 630 through 631 and 643 (COMMAND-LOOKUP): This is a similar operation to the one described in lines 625 through 626 above for the **REASON-LOOKUP** procedure.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 632       MOVE COMMAND-NAME (IXC) TO CMDEO.                           ¦
¦ 633       IF ERR-RESP < 94 MOVE RESPVAL (ERR-RESP) TO RESPEO          ¦
¦ 634           ELSE MOVE RESPVAL (94) TO RESPEO.                       ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 632 through 634: This time, we get a text version of the appropriate CICS command name, as indexed by the **ERR-RESP** value. Again, we move the text message to our user's error map.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 635       EXEC CICS SEND MAP('ACCTERR') MAPSET('ACCTSET') ERASE FREEKB ¦
¦ 636       WAIT END-EXEC.                                              ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 635 through 636: Having put all the particulars into the error map, we now send it to the user.

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 637       EXEC CICS WRITEQ TS QUEUE('ACERLOG') FROM(ACCTERRO)         ¦
¦ 638           LENGTH(ERR-LNG) END-EXEC.                               ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Lines 637 through 638: Here, we also write the error log entry to the temporary storage queue **ACERLOG.**

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ 639       EXEC CICS ABEND ABCODE('EACC') NODUMP END-EXEC.             ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Line 639: Finally, we terminate the transaction with an **ABEND** command. This produces a dump (identified by the **ABCODE** of "EACC"), returns control to CICS, and causes CICS to *back out* any changes this transaction made to a protected resource. (See"Pseudoconversational or not?" in topic 2.7 and "Recovery requirements" in topic 2.4.2 for more on protected resources.)

In addition, CICS sends a message to the input terminal saying that an abend has occurred. This message is written at the current cursor position without erasing the contents of the screen.

We've not set the TCT parameter that would override the positioning of the CICS message, although a common choice is to have such messages appear at the top of the screen.

Notice that control does *not* return to the application after an **ABEND** command.

```
640   REASON-LOOKUP.
641       IF HEX-CODE (I) = ERR-CODE MOVE I TO IXR.
642   REASON-END.  EXIT.
```

Lines 640 through 642 (CODE-LOOKUP): The **REASON-LOOKUP** procedure is explained at Lines 625-626.

```
643   COMMAND-LOOKUP.
644       IF HEX-COMMAND (I) = ERR-COMMAND MOVE I TO IXC.
645   COMMAND-END.  EXIT.
```

Lines 643 through 645 (COMMAND-LOOKUP): The **COMMAND-LOOKUP** procedure is explained at Lines 630-631.

```
646   DUMMY-END.
647       GOBACK.
```

Lines 646 through 647 (DUMMY-END): This **GOBACK** provides the logical end of program required by the compiler, as do the **GOBACK** commands terminating the other programs.

5.0 Testing and diagnosis

This part of the Primer describes:

- Types of problem
- The CICS Execution Diagnostic Facility (EDF)
- The temporary storage browse transaction
- CICS abend codes.

Subtopics

5.1 Testing
5.2 Finding the problem

5.1 Testing

This topic discusses the process of testing application code and finding the causes of problems. When you bring up an application under CICS, problems can occur at any of three levels. They may be confined to the application, and affect only that one application. On the other hand, they may affect the whole of CICS. In the worst case, they affect the entire operating system.

We'll discuss how to go about finding problems in application code, describe some of the tools that CICS provides to help in this process, and show an example of a common error using our example application. Even using the subset of CICS facilities described in this Primer, however, we can't confine the discussion to a convenient subset of mistakes -- there's no such thing. Debugging is a complex subject and very sensitive to the particular application, so that it isn't possible to discuss exhaustively even the level of errors that might affect only one application.

Problems that affect the whole CICS system are generally even more difficult, as are operating-system problems, so we'll be leaving these entirely to other sources of information.

Subtopics

5.1.1 Preparing to test

You have to do two main tasks before you can attempt to test and debug an application:

- You need to prepare the application and the system table entries
- You need to prepare the system for debugging

Subtopics
5.1.1.1 Preparing the application and system table entries
5.1.1.2 Preparing the system for debugging

5.1.1.1 Preparing the application and system table entries

1. Translate, compile and link-edit each program. Make sure that there are no error messages on any of these three steps for any program before you begin testing.

2. Use the DEBUG option on your Translator step, so that you can use Translator statement numbers with Execution Diagnostic Facility (EDF) displays.

3. Use the COBOL compiler options CLIST and DMAP so that you can relate storage locations in dumps and Execution Diagnostic Facility (EDF) displays to the original COBOL source statements, and find your variables in Working-Storage.

4. Use the **resource definition online** (RDO), (5) DEFINE TRANSACTION command for each transaction in the application.

5. Use the RDO DEFINE PROGRAM command for each program used in the application.

6. Use the RDO DEFINE MAPSET command for each mapset in the application.

7. If you are using RDO, be sure to INSTALL the new definitions.

8. Put an entry in the CSD or the FCT for each file used.

9. Build at least a test version of each of the files required.

10. Put job control DLBL, EXTENT and ASSGN cards (or the equivalent OS DD cards) in the startup job stream for each file used in the application.

11. Prepare some test data.

(5) RDO enables you to add CICS system definition file (CSD) entries for a new application program to a running CICS system.

5.1.1.2 Preparing the system for debugging

1. Make sure that EDF is included in your system. Include RDO group DFHEDF in the list you specify in the GRPLIST parameter of the SIT.

2. Turn the trace on and allow a generous trace table (at least 200 entries, better 500). Specify in the SIT:

```
TRP=(YES,ON) or TRP=(xx,ON)
and TRT=nnn where nnn>200
```

3. Request that dumps be provided, for both the transaction and the system, for all abnormal terminations. Specify in the SIT:

```
DCP=YES or DCP=xx and FDP=(xx,FORMAT) or FDP=(xx,FULL)
```

4. Be prepared to print the dumps. Have a DFHDUP job stream or procedure ready, and have the CICS dump data set(s) defined in your startup procedure. (For further guidance on using DFHDUP, see the &opgc..)

5. Enable CICS to detect loops, by setting the ICVR parameter in the SIT to a number greater than zero. Something between 5 and 10 seconds (ICVR=5000 to ICVR=10000) is usually a workable value.

6. Turn off storage recovery (SIT parameter SVD=NO), so that CICS won't try to recover after one of its storage areas is over-written. Then you will know as soon as CICS does that you've made this pernicious error. For production, storage recovery should be on. For testing, unless a great many people are testing at once, it is better left off.

7. Generate shutdown statistics.

5.1.2 Types of problem

Once you start to test, the first few problems you meet will probably be what we call **startup problems**. Most of these will be in that category we described in "Handling errors and exceptional conditions" in topic 2.9.2 as category 4 "system-application mismatches." They will produce abends that can be investigated like any others. However, there may also be system initialization problems, terminal problems, and so on. While we won't try to address these directly here, "Reference materials" in topic 5.2.3 lists sources of information for help in these areas.

When you reach the point where you can begin executing your code, you will find the problems you meet can be grouped by symptom into four general types. This classification is useful, because you need to take a slightly different approach to each type. Also, it is the same problem-classification scheme used by IBM programming support representatives (PSRs). So if you require assistance, it will help you in identifying your problem to IBM. The four types are:

Abend
Loop
Wait
Incorrect output

We'll discuss the identifying symptoms first, and later (in "Finding the problem" in topic 5.2) suggest approaches for solution.

Subtopics
5.1.2.1 Abends
5.1.2.2 Loops
5.1.2.3 Waits
5.1.2.4 Incorrect output

5.1.2.1 Abends

Abends are readily identified by the presence of that same unwelcome word in a message from CICS. When a transaction terminates abnormally, CICS sends this message both to the terminal associated with the transaction and to the transient data message destination CSMT. (At most CICS installations, this message destination is directed to a printer used by the master terminal operator, to provide a second immediate notification of the unhappy event.)

5.1.2.2 Loops

Loops come in two varieties. If you have a loop containing no CICS commands, CICS generally detects this condition and terminates your transaction with an AICA abend. It will fail to do so only if you have disabled this facility (by setting ICVR = 0 in the SIT or by setting it to such a large value that the effect is the same).

If the loop contains a CICS command, however, CICS may not detect it. The problem symptom is that the transaction never ends. It usually produces less than all of the expected output and leaves the keyboard locked, too.

5.1.2.3 Waits

The symptoms of a transaction in the **wait** state are the same as those described for a loop containing a CICS command: the transaction never ends and may not produce all of its outputs. If your transaction behaves like this, you can tell whether you have a loop or a wait by using the CEMT transaction. Display the task:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦   CEMT INQUIRE TASK FACILITY(tttt)                                     ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

("tttt" is the name of the terminal from which the transaction was entered.) If the task still exists and is active, wait a minute and repeat the inquiry. If the same task is still there, the program is probably in a loop that contains a CICS command.

If the task is not active but suspended, repeat the display once or twice. If the task remains suspended, it's probably waiting for some event that's never going to happen. There is a third possibility when you display your task, of course. It may not be there at all! This disappearing transaction syndrome is really a form of "incorrect output" (as described below), but it's usually tracked down using the techniques used for loops.

When you have a transaction that seems to be stuck in a loop or a wait, cancel it with the CEMT command:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦   CEMT SET TASK FACILITY(tttt) FORCEPURGE                              ¦
¦                                                                        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

This will produce an AMTx abend, and a dump that you can use to help determine where the loop or wait is.

A word of caution about canceling tasks, however. Some perfectly normal tasks spend a lot of time in a suspended state. A transaction that writes multiple messages to a printer, for example, is suspended most of its lifetime, waiting for the printer to print the last message it sent. And, with FORCEPURGE, CICS cannot assure system integrity, so use it with care. It's OK while debugging, but avoid it in a production system.

5.1.2.4 Incorrect output

The last category of problem covers those situations in which the transaction appears to run successfully but produces the wrong results. It includes simple wrong answers, missing or extra records in files, screens filled with what appear to be random characters, and no output at all, where a transaction just shuffles off quietly without any indication that it ever existed.

5.1.3 Tools for debugging

Before trying to describe approaches to solving these four classes of problems (which we tackle in the next topic), we need to describe three important tools that CICS provides for debugging applications. These are:

The Execution Diagnostic Facility (EDF

The temporary storage browse (CEBR) facilit

The transaction dump

Subtopics
5.1.3.1 Execution diagnostic facility (EDF)
5.1.3.2 Temporary storage browse facility (CEBR)

5.1.3.1 Execution diagnostic facility (EDF)

You'll find a complete EDF session reproduced in "A session with EDF" in topic 5.1.3.1.7; refer to it whenever you need to. (Please note that it shows the example application misbehaving due to the presence of a deliberate bug...)

EDF allows you to observe the execution of your transaction under the control of another transaction, CEDF. When you execute your transaction in this debugging mode, EDF intercepts your program(s) at the following points:

1. Transaction initiation (just before the first program gets control)

2. Just before the execution of each CICS command

3. Just after the execution of each CICS command (except ABEND, XCTL and RETURN)

4. At the termination of each program

5. At normal task termination

6. When an abend occurs

7. At abnormal task termination.

At these points, EDF interrupts execution of the program and sends a display back to the terminal. This display indicates which of these interception points has been reached and shows information appropriate to the situation.

Subtopics
5.1.3.1.1 Other information displayed
5.1.3.1.2 Useful techniques with EDF
5.1.3.1.3 Invoking EDF
5.1.3.1.4 EDF displays
5.1.3.1.5 EDF options
5.1.3.1.6 Modifying execution with EDF
5.1.3.1.7 A session with EDF

5.1.3.1.1 Other information displayed

At any one of these points, you can also display a variety of other information by selecting one of the function-key options listed on the screen. The choices include:

1. The EIB. The values are displayed in symbolic form, as listed in the complete list in the CICS/ESA Application Programming Reference.

2. Working-Storage for the program being executed. The display shows the information in both hexadecimal and character form.

3. The option of showing up to ten previous EDF displays, including all the argument values, responses, and so on.

4. The contents of any temporary storage queue.

5. The contents (in hexadecimal) of any address location within the CICS region.

5.1.3.1.2 Useful techniques with EDF

Once you have an idea about what is wrong with a program, you can test your theory by intervening in its execution:

1. Before a command is executed, you can modify any argument value (but not the command options) or you can suppress execution of the command altogether.

2. After a command is executed, you can modify the response code (and some of the argument values). This allows you to test branches of the program that are hard to reach using ordinary test data (what happens on an input/output error, for instance). It also allows you to bypass the effects of an error to see if doing so eliminates a problem.

3. At any time except just before execution of a command, you can turn off the debug mode and let the transaction proceed without any further intervention from EDF.

4. Alternatively, at any time you can suppress the displays associated with EDF until some specific condition is reached. When it is, the displays resume again. This is particularly useful when you are debugging a fairly long or repetitive transaction, because you might have to go through a lot of displays before you get to the point where the trouble is, making the process very slow. If you know that the transaction runs properly up to a certain point, you can specify that point as the condition for resuming displays, and suppress them up until then. Once the stop condition is reached, you still have access to the previous ten displays, even though they were not actually sent to the screen when originally created.

   You can express this stop condition in several different ways:

   When a specific type of command is encountered, such as READQ TS

   When a specific exceptional condition arises, such as NOTFND

   When any exceptional condition at all (that CICS classifies as an error) arises

   When the command at a specific offset is encountered

   When the command at a specific translator line number is encountered (if the DEBUG option of the Translator has been used)

   When any abend occurs

   When the task terminates.

5. At any point at all, you can change the contents of working storage for your program, and you can change most of the fields in the EIB as well.

5.1.3.1.3 Invoking EDF

You can run EDF using either one terminal or two.

For two terminals: You use the first terminal for the EDF displays and for sending input to EDF; and you use the second terminal for sending input to, and receiving output from, the transaction under test.

You start by entering, at the first terminal, the transaction:

```
CEDF tttt
```

where "tttt" is the name of the other terminal to be used in the EDF session. This second terminal must be in transceive (ATI/TTI) status. This is the most common status for display terminals, but you can check its status with CEMT:

```
CEMT INQUIRE TERMINAL(tttt)
```

and change it if it isn't already ATI/TTI:

```
CEMT SET TERMINAL(tttt) ATI TTI
```

Then enter the transaction to be tested on this second terminal.

If you want to use EDF to monitor a transaction that's already running, you can do so from another terminal. If, for example, you believe a transaction at a certain terminal to be looping, you can go to another terminal and enter a CEDF transaction naming the first terminal. EDF picks up control at the next CICS command executed, and you can then observe the sequence of commands that are causing the loop.

For one terminal: When you use EDF with just one terminal, the EDF inputs and outputs are interleaved with those from the transaction. This sounds complicated, but works quite easily in practice. The only noticeable peculiarity is that when a SEND command is followed by a RECEIVE command, the display sent by the SEND command appears twice: once when the SEND is executed, and again when the RECEIVE is executed. It isn't necessary to respond to the first display, but if you do, EDF preserves anything that was entered from the first display to the second.

To start a one-terminal session with EDF, just enter the transaction identifier "CEDF." Then enter the input that invokes the transaction you want to test.

Note: EDF makes a special provision for testing pseudoconversational transactions from a single terminal. If the terminal came out of debug

mode between the several tasks that make up a pseudoconversational transaction, it would be very hard to do any debugging after the first task. So, when a task terminates, EDF asks the operator whether debug mode is to continue to the next task. If you are debugging a pseudoconversational task, reply "yes".

5.1.3.1.4 EDF displays

EDF displays consist of a header and the screen "body." The header shows:

The identifier of the transaction being execute
The name of the program being execute
The internal task number assigned by CICS to the transactio
A display numbe
Under "STATUS," the reason for the interception by EDF

The body of the screen contains information which varies with the type of interception point, as follows:

1. **At transaction initiation**, it shows the EIB.

2. **When a command is about to be executed**, it shows the command in source language form, including the keywords, options and argument values. The command is identified by giving the name of the transaction, the name of the program being executed, and the hexadecimal offset of the command in the program. If the Translator DEBUG option has been used, the line number in the *translator* source listing will also be displayed.

3. **After the command has been executed**, the same display as for item 2 appears, along with the results (response code), in source language.

4. **Whenever an abend occurs, and at termination time for a transaction ending abnormally**, the display includes:

   The EIB

   The abend code

   For an ASRA abend, the program status word (PSW) value at the time of the interrupt

   The offset within the program of this PSW, provided it is within the program being executed.

5.1.3.1.5 EDF options

The last section of an EDF display contains a menu of things you can do at that point. The choices are listed below. Not all choices are available at each interception point; the menu shows which ones are available for the current display. To select an option, press the indicated PF key. If your terminal doesn't have PF keys, place the cursor under the option you want and press the ENTER key instead.

abend user task
Selecting this option causes the transaction being monitored to be abended, just as if an ABEND command had been issued in the program. When you make this choice, EDF asks you to enter an abend code (the ABCODE parameter of the command) to request the abend again, and then to press ENTER again, as confirmation that you really want to do this.

browse temporary storage
This option produces a display of the temporary storage queue CEBRxxxx, where xxxx is the name of the terminal from which the monitored transaction is being run. You can then use CEBR commands, discussed in "Temporary storage browse facility (CEBR)" in topic 5.1.3.2, to display or modify other temporary storage queues.

continue
If you've made changes to the screen, EDF redisplays the screen with the changes incorporated. (See "Modifying execution with EDF" in topic 5.1.3.1.6.) Otherwise, it allows the transaction to continue running until the next interrupt point.

current display
If you've modified a screen, this option causes EDF to redisplay the screen with the changes incorporated. Otherwise, it causes EDF to display the screen it showed at the last interrupt point, before you requested other displays.

EIB display
This option displays the EIB contents in symbolic form. If there is a COMMAREA at this time, its contents are also displayed.

end EDF session
This choice terminates EDF control of the transaction. The transaction resumes execution from that point, but no longer runs in debug mode.

previous display
Selecting this option causes the previous display (from the previous command, unless you've requested that other displays be remembered) to be sent to the screen. The number of the display from the current interrupt point is always 00. As you call up previous displays, the display number is decreased by 1 to -01 for the first previous display, -02 for the one before that, and so on down to the oldest display, -10.

next display
This option is the reverse of previous display. When you've gone back to a previous display, this option causes the next one forward to be shown. The display number is increased by 1.

registers at abend
This option is provided only when an ASRA abend occurs. It produces a display of the PSW and the registers at the time of the abend.

scroll forward, scroll back
These options apply to an EIB or command display that will not all fit on one screen. When this happens, a plus sign (+) appears before the

first option or field in the display, to show that there are more screens. Choosing scroll forward brings up the next screen in the display. When the screen on view isn't the first one of the display, there is a minus sign (-) before the first option or field, and you can view previous screens in the display by selecting scroll back.

scroll forward full, scroll back full

These two options have the same function for displays of Working-Storage as the scroll forward and scroll back options for EIB displays. Scroll forward full gives a Working-Storage display one full screen forward, showing addresses higher in storage than those on the previous screen. Scroll back full shows the addresses lower in storage than those on the previous screen.

scroll forward half, scroll back half

Scroll forward half is similar to scroll forward full, except that the display of working storage is advanced by only half a screen. This means that the addresses on the bottom half of the previous screen still appear on the top half of the new screen, followed by the next half-screen of higher addresses. Scroll back half is the backward counterpart of scroll forward half.

suppress displays

This option causes EDF to suppress its displays until one of the stop conditions (see next item) is met.

stop conditions

Selecting this option causes EDF to present a menu, in which you can specify conditions under which you want a display. You use this feature when you are about to suppress the displays, to indicate when they should be resumed again. However, you can also use it to get displays at points in the code between the normal EDF interception points. This is particularly helpful in locating loops and finding the cause of incorrect output.

switch hex/char

If EDF is displaying information in character form, this option causes it to switch to hexadecimal for subsequent displays, and back again. It applies only to the basic interrupt display and does not affect Working-Storage displays, stop condition displays, or remembered displays.

user display

This option causes EDF to display what would be on the screen if the transaction were not running under EDF. To get back to EDF from the user display, simply press the ENTER key.

working storage

This option allows you to see the contents of the Working-Storage area in your program, or of any other address in the CICS region. The address of Working-Storage is displayed at the top of the screen. You can browse through the entire area using the scroll commands, or you can simply enter a new address at the top of the screen. This address can be anywhere within the CICS region.

remember display

This option allows you to record displays that EDF does not ordinarily save. EDF can save up to ten displays, and it keeps the last ten command displays unless you use this option to save something else. Note, however, that if you save a working storage display, only the screen on view is saved; otherwise all the pages that make up the display are saved and can be recalled.

5.1.3.1.6 Modifying execution with EDF

You can modify the execution of a transaction in four different ways:

Changing the contents of Working-Storage and the EI
Changing the argument values before a command is execute
Changing the response code afterwar
Suppressing commands altogether

You make these changes by typing over the information shown on the screen with the information you want used instead. You can change any area of the screen where the cursor stops when you use the tab keys, except for the menu area at the bottom.

When you change the screen, you must observe the following conventions:

If you want to suppress the execution of a command entirely, type NO over the first three characters of the command.

You can change argument values in commands, but not keywords

When you change an argument in the command display (as opposed t Working-Storage), you can change only the part shown on the display. If it is such a long argument that only part of it appears on the screen, you should change the area in Working-Storage to which the argument points.

You can change the response code from a command to any response cod that applies to that command, including the all-purpose ERROR. In this way you can test your program's error recovery routines.

Conversely, if the response code from a command was some exceptiona condition, and you want to see what would happen if you'd had a normal response to the command, type NORMAL over the response code.

When you overtype a field representing a data area in your program the change is made directly in program storage and is permanent. However, if you change a field that represents a constant (a program literal), program storage isn't changed, because this might affect any other parts of the program that use the same constant. The command is executed with the changed data, but when the command is displayed after execution, the original argument values re-appear. If you execute the same command more than once, you must enter this type of change afresh each time.

When arguments are displayed in character form, any character tha cannot be displayed on the screen is shown as a period (.). So you're not allowed to change any character to a period in a character display. If you must do this, use the switch hex/char option to change to a hexadecimal display and then use "4B" for period.

EDF only accepts uppercase characters. If your terminal ha lowercase, and uppercase translation is not specified for it (this will have been specified by your system programmer), be careful to use uppercase at all times.

5.1.3.1.7 A session with EDF

What follows is an "as it happened" reproduction of an EDF session after we'd found that the example application had a nasty little bug in it.

Note: If you want to follow this sample EDF session at your own terminal, you need to get the following things done first:

Make a copy of ACCT02, and add the bug, as shown i "Lines 466 through 470 (UPDT-DELETE)" in topic 4.3 .

Ask your system programmer to make a copy of the EDF group EDF an change resource security checking to RESSEC=NO.

It all began innocently enough, simply by trying to delete account record number 11111 from our account file....

The first thing we did, of course, was type in the transaction id:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦ acct                                                                   ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 55. Invoking the account file transaction

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦  ACCOUNT FILE: MENU                                                    ¦
¦     TO SEARCH BY NAME, ENTER:                          ONLY SURNAME    ¦
¦                                                        REQUIRED. EITHER¦
¦        SURNAME:               FIRST NAME:              MAY BE PARTIAL. ¦
¦     FOR INDIVIDUAL RECORDS, ENTER:                                     ¦
¦                                                        PRINTER REQUIRED¦
¦        REQUEST TYPE:     ACCOUNT:        PRINTER:      ONLY FOR PRINT  ¦
¦                                                        REQUESTS.       ¦
¦        REQUEST TYPES: D = DISPLAY  A = ADD    X = DELETE              ¦
¦                       P = PRINT    M = MODIFY                         ¦
¦     THEN PRESS "ENTER"             -OR-   PRESS "CLEAR" TO EXIT        ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 56. The account file menu

This gave us the menu, as shown above. Next, we had to say which record we wanted to delete.

So we typed in x (for delete) and 11111 (the record number) and pressed the ENTER key.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦  ACCOUNT FILE: MENU                                                    ¦
¦     TO SEARCH BY NAME, ENTER:                          ONLY SURNAME    ¦
¦                                                        REQUIRED. EITHER¦
¦        SURNAME:               FIRST NAME:              MAY BE PARTIAL. ¦
¦     FOR INDIVIDUAL RECORDS, ENTER:                                     ¦
¦                                                        PRINTER REQUIRED¦
¦        REQUEST TYPE: x   ACCOUNT: 11111   PRINTER:     ONLY FOR PRINT  ¦
```

```
¦                                                          REQUESTS.         ¦
¦        REQUEST TYPES: D = DISPLAY  A = ADD    X = DELETE                   ¦
¦                       P = PRINT    M = MODIFY                              ¦
¦     THEN PRESS "ENTER"             -OR-   PRESS "CLEAR" TO EXIT            ¦
¦                                                                            ¦
+----------------------------------------------------------------------------+
```

Figure 57. Let's delete account number 11111

```
+----------------------------------------------------------------------------+
¦                                                                            ¦
¦  ACCOUNT FILE: DELETION                                                    ¦
¦  ACCOUNT NO: 11111       SURNAME:   LOCKS                                  ¦
¦                          FIRST:     GOLDIE       MI: X  TITLE: LADY      ¦
¦  TELEPHONE: 2345212341   ADDRESS:   THE COTTAGE                            ¦
¦                                     WOODLANDS                              ¦
¦                                     HANTS                                  ¦
¦  OTHERS WHO MAY CHARGE:                                                    ¦
¦  THE 3 BEARS                                                               ¦
¦  NO. CARDS ISSUED: 4     DATE ISSUED: 05 04 89      REASON: N              ¦
¦  CARD CODE: 2            APPROVED BY: HRH           SPECIAL CODES:         ¦
¦  ACCOUNT STATUS: N       CHARGE LIMIT:  1000.00                            ¦
¦  HISTORY:   BALANCE      BILLED        AMOUNT    PAID       AMOUNT         ¦
¦               0.00      00/00/00        0.00    00/00/00     0.00         ¦
¦               0.00      00/00/00        0.00    00/00/00     0.00         ¦
¦               0.00      00/00/00        0.00    00/00/00     0.00         ¦
¦  ENTER "Y" TO CONFIRM OR "CLEAR" TO CANCEL                                 ¦
¦                                                                            ¦
+----------------------------------------------------------------------------+
```

Figure 58. Now confirm the deletion...

As you see, this fetched Goldie Locks' record, and asked us to confirm the deletion.

```
+----------------------------------------------------------------------------+
¦                                                                            ¦
¦  ACCOUNT FILE: DELETION                                                    ¦
¦  ACCOUNT NO: 11111       SURNAME:   LOCKS                                  ¦
¦                          FIRST:     GOLDIE       MI: X  TITLE: LADY      ¦
¦  TELEPHONE: 2345212341   ADDRESS:   THE COTTAGE                            ¦
¦                                     WOODLANDS                              ¦
¦                                     HANTS                                  ¦
¦  OTHERS WHO MAY CHARGE:                                                    ¦
¦  THE 3 BEARS                                                               ¦
¦  NO. CARDS ISSUED: 4     DATE ISSUED: 05 04 89   REASON: N                 ¦
¦  CARD CODE: 2            APPROVED BY: HRH        SPECIAL CODES:            ¦
¦  ACCOUNT STATUS: N       CHARGE LIMIT:  1000.00                            ¦
¦  HISTORY:   BALANCE      BILLED        AMOUNT    PAID       AMOUNT         ¦
¦               0.00      00/00/00        0.00    00/00/00     0.00         ¦
¦               0.00      00/00/00        0.00    00/00/00     0.00         ¦
¦               0.00      00/00/00        0.00    00/00/00     0.00         ¦
¦  ENTER "Y" TO CONFIRM OR "CLEAR" TO CANCEL                    y            ¦
¦                                                                            ¦
+----------------------------------------------------------------------------+
```

Figure 59. ... by typing "Y"

```
+----------------------------------------------------------------------------+
```

```
¦                                                                          ¦
¦  ACCOUNT FILE: ERROR REPORT                                             ¦
¦  TRANSACTION  AC02  HAS FAILED IN PROGRAM  ACCT02    BECAUSE OF          ¦
¦  A PROGRAM OR FCT TABLE ERROR (INVALID FILE REQUEST).                   ¦
¦  COMMAND  DELETE               RESP  INVREQ                              ¦
¦  THE FILE IS: ACCTFIL .                                                 ¦
¦  PLEASE ASK YOUR SUPERVISOR TO CONVEY THIS INFORMATION TO THE           ¦
¦  OPERATIONS STAFF.                                                       ¦
¦  THEN PRESS "CLEAR".  THIS TERMINAL IS NO LONGER UNDER CONTROL OF       ¦
¦  THE "ACCT" APPLICATION.                                                 ¦
¦  DFH2206 12:31:13 CIDCICSC TRANSACTION AC02 HAS FAILED WITH ABEND EACC.  ¦
¦   RESOURCE BACKOUT WAS SUCCESSFUL.                                       ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 60. Hold it! We've got a problem -- and we've been backed out

Well! That didn't work, so what do we do next? First, we'd better delete the scratchpad entry for this record, so we can try again, this time with EDF on. (You see, we've just reserved account number 11111 in ACCT01 before we displayed the Account Detail screen. So, unless we now remove the reservation, we won't be able to try the deletion again for ten minutes. We'll use CECI, a useful CICS transaction -- the command interpreter.)

First, we CLEAR the screen, then we can type:

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦ ceci deleteq ts queue(ac011111)                                          ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 61. Deleting the scratchpad record. We have to do this so that we can retry the deletion.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦   DELETEQ TS QUEUE (AC011111)                                            ¦
¦   STATUS:  ABOUT TO EXECUTE COMMAND                          NAME=       ¦
¦    EXEC CICS  DELETEQ TS                                                 ¦
¦     Queue( 'AC011111' )                                                  ¦
¦     < Sysid() >                                                          ¦
¦  PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF  ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 62. Going, going, ...

We just press ENTER to delete the queue entry.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦   DELETEQ TS QUEUE (AC011111)                                            ¦
¦   STATUS:  COMMAND EXECUTION COMPLETE                        NAME=       ¦
¦    EXEC CICS  DELETEQ TS                                                 ¦
¦     Queue( 'AC011111' )                                                  ¦
¦     < Sysid() >                                                          ¦
```

```
¦    RESPONSE: NORMAL                        EIBRESP=+0000000000            ¦
¦  PF 1 HELP 2 HEX 3 END 4 EIB 5 VAR 6 USER 7 SBH 8 SFH 9 MSG 10 SB 11 SF  ¦
¦                                                                           ¦
+---------------------------------------------------------------------------+
```

Figure 63. Gone!

First, we hit PF3 to end the CECI transaction, and CLEAR to get a clear screen. Now we can use EDF to try and find out what's going wrong. To invoke the facility, we simply type CEDF:

```
+---------------------------------------------------------------------------+
¦                                                                           ¦
¦ cedf                                                                      ¦
¦                                                                           ¦
+---------------------------------------------------------------------------+
```

Figure 64. Now activate EDF

```
+---------------------------------------------------------------------------+
¦                                                                           ¦
¦ THIS TERMINAL: EDF MODE ON                                                ¦
¦                                                                           ¦
+---------------------------------------------------------------------------+
```

Figure 65. OK

Now we can CLEAR the screen and re-enter the ACCT transaction:

```
+---------------------------------------------------------------------------+
¦                                                                           ¦
¦ acct                                                                      ¦
¦                                                                           ¦
+---------------------------------------------------------------------------+
```

Figure 66. Now re-enter the account file transaction

```
+---------------------------------------------------------------------------+
¦                                                                           ¦
¦   TRANSACTION: ACCT   PROGRAM: ACCT00  TASK NUMBER: 0000089  DISPLAY: 00 ¦
¦   STATUS:  PROGRAM INITIATION                                             ¦
¦      EIBTIME      = 123343                                              ¦
¦      EIBDATE      = 89170                                               ¦
¦      EIBTRNID     = 'ACCT'                                              ¦
¦      EIBTASKN     = 89                                                  ¦
¦      EIBTRMID     = '037L'                                              ¦
¦      EIBCPOSN     = 4                                                   ¦
¦      EIBCALEN     = 0                                                   ¦
¦      EIBAID       = X'7D'                              AT X'00543F1E'¦
¦      EIBFN        = X'0000'                            AT X'00543F1F'¦
¦      EIBRCODE     = X'000000000000'                    AT X'00543F21'¦
¦      EIBDS        = '........'                                          ¦
¦  +   EIBREQID     = '........'                                          ¦
¦   RESPONSE:                                                             ¦
¦                                                          REPLY:         ¦
¦  ENTER:  CONTINUE                                                       ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION  ¦
```

```
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS  ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED        ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 67. And into EDF

Here's our first EDF screen. From now on, we'll use the PF4 key to suppress displays. EDF goes on building (and remembering) its displays -- we simply don't want to be overwhelmed by seeing them all. (At any point, you can use the PF10 key to step back through a maximum of ten previous displays. We'll see how later on.)

Any abnormal response, or any program output, or the end of the task, will all end the display suppression and show us the appropriate screen. Press the PF4 key, then, and away we go!

And the next screen we see is this one:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   TRANSACTION: ACCT   PROGRAM:          TASK NUMBER: 0000089  DISPLAY: 00¦
¦   STATUS:  TASK TERMINATION                                             ¦
¦   RESPONSE:                                                             ¦
¦  TO CONTINUE EDF SESSION REPLY YES                              REPLY: NO¦
¦  ENTER:  CONTINUE                                                       ¦
¦  PF1 : UNDEFINED           PF2 : SWITCH HEX/CHAR    PF3 : END EDF SESSION¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE    PF6 : USER DISPLAY   ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS¦
¦  PF10: PREVIOUS DISPLAY    PF11: UNDEFINED          PF12: UNDEFINED      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 68. OK so far

We overtype the "REPLY: NO" with our "yes" and (as usual) press ENTER.

This ensures that EDF will continue monitoring the next transaction (AC01) in our pseudoconversational sequence.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   TRANSACTION: ACCT   PROGRAM:          TASK NUMBER: 0000089  DISPLAY: 00¦
¦   STATUS:  TASK TERMINATION                                             ¦
¦   RESPONSE:                                                             ¦
¦  TO CONTINUE EDF SESSION REPLY YES                             REPLY: yes¦
¦  ENTER:  CONTINUE                                                       ¦
¦  PF1 : UNDEFINED           PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE   PF6 : USER DISPLAY   ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS¦
¦  PF10: PREVIOUS DISPLAY    PF11: UNDEFINED         PF12: UNDEFINED      ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 69. Again "yes" to continue with the next transaction

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
```

```
¦  ACCOUNT FILE: MENU                                                      ¦
¦  TO SEARCH BY NAME, ENTER:                              ONLY SURNAME     ¦
¦                                                       REQUIRED. EITHER ¦
¦     SURNAME:              FIRST NAME:                  MAY BE PARTIAL.  ¦
¦  FOR INDIVIDUAL RECORDS, ENTER:                                          ¦
¦                                                       PRINTER REQUIRED ¦
¦     REQUEST TYPE:     ACCOUNT:         PRINTER:       ONLY FOR PRINT   ¦
¦                                                       REQUESTS.        ¦
¦     REQUEST TYPES:  D = DISPLAY    A = ADD     X = DELETE               ¦
¦                     P = PRINT      M = MODIFY                           ¦
¦  THEN PRESS "ENTER"             -OR-   PRESS "CLEAR" TO EXIT             ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 70. Back to the menu

Suppression of EDF displays ends for the time being with our user screen, the menu. Now we type in that troublesome record, 11111.

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦  ACCOUNT FILE: MENU                                                     ¦
¦  TO SEARCH BY NAME, ENTER:                              ONLY SURNAME     ¦
¦                                                       REQUIRED. EITHER ¦
¦     SURNAME:            FIRST NAME:                   MAY BE PARTIAL.  ¦
¦  FOR INDIVIDUAL RECORDS, ENTER:                                         ¦
¦                                                       PRINTER REQUIRED ¦
¦     REQUEST TYPE: x   ACCOUNT: 11111   PRINTER:       ONLY FOR PRINT   ¦
¦                                                       REQUESTS.        ¦
¦     REQUEST TYPES:  D = DISPLAY    A = ADD     X = DELETE              ¦
¦                     P = PRINT      M = MODIFY                          ¦
¦  THEN PRESS "ENTER"             -OR-   PRESS "CLEAR" TO EXIT            ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 71. Now we can enter record 11111

We press ENTER, cross our fingers, and see what happens...

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   TRANSACTION: AC01  PROGRAM: ACCT01   TASK NUMBER: 0000096  DISPLAY: 00 ¦
¦   STATUS:  PROGRAM INITIATION                                           ¦
¦      EIBTIME      = 123616                                              ¦
¦      EIBDATE      = 89170                                               ¦
¦      EIBTRNID     = 'AC01'                                              ¦
¦      EIBTASKN     = 96                                                  ¦
¦      EIBTRMID     = '037L'                                              ¦
¦      EIBCPOSN     = 691                                                 ¦
¦      EIBCALEN     = 0                                                   ¦
¦      EIBAID       = X'7D'                              AT X'00543F1E'   ¦
¦      EIBFN        = X'0000'                            AT X'00543F1F'   ¦
¦      EIBRCODE     = X'000000000000'                    AT X'00543F21'   ¦
¦      EIBDS        = '........'                                          ¦
¦  +   EIBREQID     = '........'                                          ¦
¦   RESPONSE:                                                             ¦
¦                                                                REPLY:   ¦
¦  ENTER:  CONTINUE                                                       ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY    ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS ¦
```

```
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED        ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 72. Ready to begin the request analysis. Here's the next EDF display.

Again, we'll use PF4 to suppress displays until something unusual happens.

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦   TRANSACTION: AC01   PROGRAM: ACCT01   TASK NUMBER: 0000096  DISPLAY: 00¦
¦   STATUS:  COMMAND EXECUTION COMPLETE                                    ¦
¦   EXEC CICS READQ TS                                                     ¦
¦    QUEUE ('AC011111')                                                    ¦
¦    INTO ('    ........')                                                 ¦
¦    LENGTH (12)                                                           ¦
¦    ITEM (1)                                                              ¦
¦    NOHANDLE                                                              ¦
¦   OFFSET:X'000F6C'    LINE:00281          EIBFN=X'0A04'                  ¦
¦   RESPONSE: QIDERR                        EIBRESP=44                     ¦
¦                                                          REPLY:          ¦
¦  ENTER:  CONTINUE                                                        ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION  ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS  ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: ABEND USER TASK  ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 73. Response: QIDERR. This tells us no-one else owns our record.

And here we are with a QIDERR condition. However, it's what we expect when reading the scratchpad entry, so we can proceed. Using PF4 again to suppress displays, let's carry on....

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦   TRANSACTION: AC01  PROGRAM:          TASK NUMBER: 0000096   DISPLAY: 00¦
¦   STATUS:  TASK TERMINATION                                              ¦
¦   RESPONSE:                                                              ¦
¦  TO CONTINUE EDF SESSION REPLY YES                          REPLY: NO    ¦
¦  ENTER:  CONTINUE                                                        ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION  ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS  ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED        ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 74. OK, carry on

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦   TRANSACTION: AC01  PROGRAM:         TASK NUMBER: 0000096  DISPLAY: 00  ¦
¦   STATUS:  TASK TERMINATION                                              ¦
¦   RESPONSE:                                                              ¦
¦  TO CONTINUE EDF SESSION REPLY YES                          REPLY: yes   ¦
```

```
¦  ENTER:  CONTINUE                                                          ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION  ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS  ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED        ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 75. "yes" to carry on into AC02

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦  ACCOUNT FILE: DELETION                                                  ¦
¦  ACCOUNT NO: 11111       SURNAME:   LOCKS                                ¦
¦                          FIRST:     GOLDIE          MI: X  TITLE: LADY   ¦
¦  TELEPHONE: 2345212341   ADDRESS:   THE COTTAGE                          ¦
¦                                     WOODLANDS                            ¦
¦                                     HANTS                                ¦
¦  OTHERS WHO MAY CHARGE:                                                  ¦
¦  THE 3 BEARS                                                             ¦
¦  NO. CARDS ISSUED: 4     DATE ISSUED: 05 04 89      REASON: N            ¦
¦  CARD CODE: 2            APPROVED BY: HRH           SPECIAL CODES:       ¦
¦  ACCOUNT STATUS: N       CHARGE LIMIT:  1000.00                          ¦
¦  HISTORY:   BALANCE      BILLED        AMOUNT       PAID          AMOUNT  ¦
¦               0.00       00/00/00        0.00       00/00/00        0.00  ¦
¦               0.00       00/00/00        0.00       00/00/00        0.00  ¦
¦               0.00       00/00/00        0.00       00/00/00        0.00  ¦
¦  ENTER "Y" TO CONFIRM OR "CLEAR" TO CANCEL                               ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 76. OK -- the big moment is (nearly) here

Let's type "y" and see what happens. Now we should at least find out a bit more about the problem....

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦  ACCOUNT FILE: DELETION                                                  ¦
¦  ACCOUNT NO: 11111       SURNAME:   LOCKS                                ¦
¦                          FIRST:     GOLDIE          MI: X  TITLE: LADY   ¦
¦  TELEPHONE: 2345212341   ADDRESS:   THE COTTAGE                          ¦
¦                                     WOODLANDS                            ¦
¦                                     HANTS                                ¦
¦  OTHERS WHO MAY CHARGE:                                                  ¦
¦  THE 3 BEARS                                                             ¦
¦  NO. CARDS ISSUED: 4     DATE ISSUED: 05 04 89      REASON: N            ¦
¦  CARD CODE: 2            APPROVED BY: HRH           SPECIAL CODES:       ¦
¦  ACCOUNT STATUS: N       CHARGE LIMIT:  1000.00                          ¦
¦  HISTORY:   BALANCE      BILLED        AMOUNT       PAID          AMOUNT  ¦
¦               0.00       00/00/00        0.00       00/00/00        0.00  ¦
¦               0.00       00/00/00        0.00       00/00/00        0.00  ¦
¦               0.00       00/00/00        0.00       00/00/00        0.00  ¦
¦  ENTER "Y" TO CONFIRM OR "CLEAR" TO CANCEL                   y           ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 77. Here we go

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   TRANSACTION: AC02  PROGRAM: ACCT02   TASK NUMBER: 0000113  DISPLAY: 00 ¦
¦   STATUS:  PROGRAM INITIATION                                          ¦
¦      COMMAREA     = 'X11111'                                           ¦
¦      EIBTIME      = 123914                                             ¦
¦      EIBDATE      = 89170                                              ¦
¦      EIBTRNID     = 'AC02'                                             ¦
¦      EIBTASKN     = 113                                                ¦
¦      EIBTRMID     = '037L'                                             ¦
¦      EIBCPOSN     = 1743                                               ¦
¦      EIBCALEN     = 6                                                  ¦
¦      EIBAID       = X'7D'                                AT X'00543F1E'¦
¦      EIBFN        = X'0000'                              AT X'00543F1F'¦
¦      EIBRCODE     = X'000000000000'                      AT X'00543F21'¦
¦      EIBDS        = '........'                                         ¦
¦  +   EIBREQID     = '........'                                         ¦
¦   RESPONSE:                                                            ¦
¦                                                           REPLY:       ¦
¦  ENTER:  CONTINUE                                                      ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY    ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED       ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 78. Ready?

Again we use PF4 to suppress displays, as usual.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦   TRANSACTION: AC02  PROGRAM: ACCT02   TASK NUMBER: 0000113   DISPLAY: 00¦
¦   STATUS:  COMMAND EXECUTION COMPLETE                                  ¦
¦   EXEC CICS DELETE                                                     ¦
¦    FILE ('ACCTFIL ')                                                   ¦
¦    RIDFLD ('11111')                                                    ¦
¦   OFFSET:X'0018CE'    LINE:00472          EIBFN=X'0608'                ¦
¦   RESPONSE: INVREQ                        EIBRESP=16                   ¦
¦                                                            REPLY:      ¦
¦  ENTER:  CONTINUE                                                      ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY    ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: ABEND USER TASK ¦
¦                                                                        ¦
+------------------------------------------------------------------------+
```

Figure 79. The INVREQ (invalid request) condition

And here's an INVREQ (invalid request) condition. This is **not** what we expect. If the RIDFLD field looked odd (it doesn't here) we might want to use PF5 to start looking at Working-Storage, or PF6 to examine the user display. However, using PF4 again, let's carry on....

The following screen flashes up briefly and disappears again:

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦  ACCOUNT FILE: ERROR REPORT                                           ¦
```

```
¦  TRANSACTION  AC02  HAS FAILED IN PROGRAM  ACCT02    BECAUSE OF          ¦
¦  A PROGRAM OR FCT TABLE ERROR (INVALID FILE REQUEST).                     ¦
¦  COMMAND  DELETE               RESP  INVREQ                               ¦
¦  THE FILE IS: ACCTFIL .                                                   ¦
¦  PLEASE ASK YOUR SUPERVISOR TO CONVEY THIS INFORMATION TO THE             ¦
¦  OPERATIONS STAFF.                                                        ¦
¦  THEN PRESS "CLEAR".  THIS TERMINAL IS NO LONGER UNDER CONTROL OF         ¦
¦  THE "ACCT" APPLICATION.                                                  ¦
¦                                                                           ¦
+---------------------------------------------------------------------------+
```

Figure 80. The error report

```
+---------------------------------------------------------------------------+
¦                                                                           ¦
¦   TRANSACTION: AC02  PROGRAM: ACCT04  TASK NUMBER: 0000113  DISPLAY: 00   ¦
¦   STATUS:  AN ABEND HAS OCCURRED                                          ¦
¦      COMMAREA     = 'ACCT02  ....16'                                      ¦
¦      EIBTIME      = 123914                                                ¦
¦      EIBDATE      = 89170                                                 ¦
¦      EIBTRNID     = 'AC02'                                                ¦
¦      EIBTASKN     = 113                                                   ¦
¦      EIBTRMID     = '037L'                                                ¦
¦      EIBCPOSN     = 1743                                                  ¦
¦      EIBCALEN     = 14                                                    ¦
¦      EIBAID       = X'7D'                                AT X'00543F1E'¦
¦      EIBFN        = X'0E0C'  ABEND                       AT X'00543F1F'¦
¦      EIBRCODE     = X'000000000000'                      AT X'00543F21'¦
¦      EIBDS        = 'ACCTFIL '                                            ¦
¦  +   EIBREQID     = '........'                                            ¦
¦   ABEND :   EACC                                                         ¦
¦                                                              REPLY:       ¦
¦  ENTER:  CONTINUE                                                        ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION  ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS  ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED        ¦
¦                                                                           ¦
+---------------------------------------------------------------------------+
```

Figure 81. Here's our abend, EACC

And the next EDF display we stop at is this ABEND status warning.

Now we'll use the PF10 key to step back through the remembered displays (that we've been suppressing), in the hope that the cause of the problem will become clearer. Watch the "DISPLAY:" number in the top right hand corner of each screen.

```
+---------------------------------------------------------------------------+
¦                                                                           ¦
¦   TRANSACTION: AC02  PROGRAM: ACCT04  TASK NUMBER: 0000113  DISPLAY: -01 ¦
¦   STATUS:  ABOUT TO EXECUTE COMMAND                                       ¦
¦   EXEC CICS ABEND                                                         ¦
¦    ABCODE ('EACC')                                                        ¦
¦    NODUMP                                                                 ¦
¦   OFFSET:X'00035E'    LINE:00646          EIBFN=X'0E0C'                  ¦
¦   RESPONSE:                                                               ¦
¦                                                              REPLY:       ¦
¦  ENTER:  CURRENT DISPLAY                                                 ¦
¦  PF1 : UNDEFINED          PF2 : UNDEFINED        PF3 : UNDEFINED         ¦
```

```
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE  PF6 : USER DISPLAY      ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD   PF9 : STOP CONDITIONS   ¦
¦  PF10: PREVIOUS DISPLAY   PF11: NEXT DISPLAY     PF12: UNDEFINED         ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 82. Just prior to the ABEND command

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦ TRANSACTION: AC02  PROGRAM: ACCT04   TASK NUMBER: 0000113  DISPLAY: -02  ¦
¦ STATUS:  COMMAND EXECUTION COMPLETE                                      ¦
¦ EXEC CICS WRITEQ TS                                                      ¦
¦ QUEUE ('ACERLOG ')                                                       ¦
¦ FROM ('.......AC02...ACCT02  ...A PROGRAM OR FCT TABLE ERROR (   I'...)  ¦
¦ LENGTH (156)                                                             ¦
¦ AUXILIARY                                                                ¦
¦   OFFSET:X'000320'    LINE:00644          EIBFN=X'0A02'                  ¦
¦   RESPONSE: NORMAL                        EIBRESP=0                      ¦
¦                                                              REPLY:      ¦
¦  ENTER:  CURRENT DISPLAY                                                 ¦
¦  PF1 : UNDEFINED           PF2 : UNDEFINED           PF3 : UNDEFINED     ¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE     PF6 : USER DISPLAY  ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD      PF9 : STOP CONDITION¦
¦  PF10: PREVIOUS DISPLAY    PF11: NEXT DISPLAY        PF12: UNDEFINED     ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 83. Sent the error map

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦ TRANSACTION: AC02   PROGRAM: ACCT04  TASK NUMBER: 0000089  DISPLAY: -04  ¦
¦ STATUS:  COMMAND EXECUTION COMPLETE                                      ¦
¦ EXEC CICS SEND MAP                                                       ¦
¦ MAP ('ACCTERR')                                                          ¦
¦ FROM ('.......AC02...ACCT02  ...A PROGRAM OR FCT TABLE ERROR (I'...)     ¦
¦ LENGTH (156)                                                             ¦
¦ MAPSET ('ACCTSET')                                                       ¦
¦ TERMINAL                                                                 ¦
¦ WAIT                                                                     ¦
¦ FREEKB                                                                   ¦
¦ ERASE                                                                    ¦
¦   OFFSET:X'0002CE'    LINE:00639          EIBFN=X'1804'                  ¦
¦   RESPONSE: NORMAL                        EIBRESP=0                      ¦
¦                                                          REPLY:          ¦
¦  ENTER:  CURRENT DISPLAY                                                 ¦
¦  PF1 : UNDEFINED           PF2 : UNDEFINED         PF3 : UNDEFINED       ¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE   PF6 : USER DISPLAY    ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD    PF9 : STOP CONDITION  ¦
¦  PF10: PREVIOUS DISPLAY    PF11: NEXT DISPLAY      PF12: UNDEFINED       ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 84. Writing to temporary storage queue

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦ TRANSACTION: AC02   PROGRAM: ACCT04  TASK NUMBER: 0000089  DISPLAY:  -05 ¦
```

```
¦ STATUS:  ABOUT TO EXECUTE COMMAND                                          ¦
¦ EXEC CICS SEND MAP                                                         ¦
¦ MAP ('ACCTERR')                                                            ¦
¦ FROM ('.......AC02...ACCT02  ...A PROGRAM OR FCT TABLE ERROR (I'...)       ¦
¦ LENGTH (156)                                                               ¦
¦ MAPSET ('ACCTSET')                                                         ¦
¦ TERMINAL                                                                   ¦
¦ WAIT                                                                       ¦
¦ FREEKB                                                                     ¦
¦ ERASE                                                                      ¦
¦   OFFSET:X'0002CE'    LINE:00639          EIBFN=X'1804'                    ¦
¦   RESPONSE:                                                                ¦
¦                                                            REPLY:          ¦
¦  ENTER:  CURRENT DISPLAY                                                   ¦
¦  PF1 : UNDEFINED           PF2 : UNDEFINED         PF3 : UNDEFINED         ¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE   PF6 : USER DISPLAY      ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS   ¦
¦  PF10: PREVIOUS DISPLAY    PF11: NEXT DISPLAY      PF12: UNDEFINED         ¦
¦                                                                            ¦
+------------------------------------------------------------------------------+
```

Figure 85. About to write to temporary storage queue

```
+------------------------------------------------------------------------------+
¦                                                                              ¦
¦ TRANSACTION: AC02   PROGRAM: ACCT04  TASK NUMBER: 0000113   DISPLAY: -03     ¦
¦ STATUS:  ABOUT TO EXECUTE COMMAND                                            ¦
¦ EXEC CICS WRITEQ TS                                                          ¦
¦ QUEUE ('ACERLOG ')                                                           ¦
¦ FROM ('.......AC02...ACCT02  ...A PROGRAM OR FCT TABLE ERROR (I'...)         ¦
¦ LENGTH (156)                                                                 ¦
¦ AUXILIARY                                                                    ¦
¦   OFFSET:X'000320'    LINE:00644          EIBFN=X'0A02'                      ¦
¦   RESPONSE:                                                                  ¦
¦                                                              REPLY:          ¦
¦  ENTER:  CURRENT DISPLAY                                                     ¦
¦  PF1 : UNDEFINED          PF2 : UNDEFINED        PF3 : UNDEFINED            ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE  PF6 : USER DISPLAY         ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD   PF9 : STOP CONDITIONS      ¦
¦  PF10: PREVIOUS DISPLAY   PF11: NEXT DISPLAY     PF12: UNDEFINED            ¦
¦                                                                              ¦
+------------------------------------------------------------------------------+
```

Figure 86. About to send the error map

```
+------------------------------------------------------------------------------+
¦                                                                              ¦
¦   TRANSACTION: AC02  PROGRAM: ACCT04  TASK NUMBER: 0000113  DISPLAY: -06     ¦
¦   STATUS:  PROGRAM INITIATION                                                ¦
¦      COMMAREA     = 'ACCT02  ....16'                                         ¦
¦      EIBTIME      = 123914                                                   ¦
¦      EIBDATE      = 89170                                                    ¦
¦      EIBTRNID     = 'AC02'                                                   ¦
¦      EIBTASKN     = 113                                                      ¦
¦      EIBTRMID     = '037L'                                                   ¦
¦      EIBCPOSN     = 1743                                                     ¦
¦      EIBCALEN     = 14                                                       ¦
¦      EIBAID       = X'7D'                                  AT X'00543F1E'    ¦
¦      EIBFN        = X'0E02'  LINK                          AT X'00543F1F'    ¦
¦      EIBRCODE     = X'000000000000'                        AT X'00543F21'    ¦
¦      EIBDS        = 'ACCTFIL '                                               ¦
```

```
¦  +   EIBREQID     = '........'                                          ¦
¦   RESPONSE:                                                            ¦
¦                                                                REPLY:  ¦
¦  ENTER:  CURRENT DISPLAY                                               ¦
¦  PF1 : UNDEFINED           PF2 : UNDEFINED         PF3 : UNDEFINED     ¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE   PF6 : USER DISPLAY  ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS ¦
¦  PF10: PREVIOUS DISPLAY    PF11: NEXT DISPLAY      PF12: UNDEFINED     ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Figure 87. Starting the error-handling program, ACCT04

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦  TRANSACTION: AC02   PROGRAM: ACCT02  TASK NUMBER: 0000113  DISPLAY: -07 ¦
¦  STATUS:  ABOUT TO EXECUTE COMMAND                                     ¦
¦  EXEC CICS LINK                                                       ¦
¦   PROGRAM ('ACCT04  ')                                                ¦
¦   COMMAREA ('ACCT02  ....16')                                         ¦
¦   LENGTH (14)                                                         ¦
¦   OFFSET:X'000B1A'    LINE:00180          EIBFN=X'0E02'               ¦
¦   RESPONSE:                                                           ¦
¦                                                          REPLY:       ¦
¦  ENTER:  CURRENT DISPLAY                                              ¦
¦  PF1 : UNDEFINED           PF2 : UNDEFINED         PF3 : UNDEFINED     ¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE   PF6 : USER DISPLAY  ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS ¦
¦  PF10: PREVIOUS DISPLAY    PF11: NEXT DISPLAY      PF12: UNDEFINED     ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Figure 88. Linking to the error program, ACCT04

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦   TRANSACTION: AC02   PROGRAM: ACCT02  TASK NUMBER: 0000113  DISPLAY: -08 ¦
¦   STATUS:  COMMAND EXECUTION COMPLETE                                 ¦
¦   EXEC CICS HANDLE CONDITION                                          ¦
¦    ERROR                                                              ¦
¦   OFFSET:X'000AC0'    LINE:00179          EIBFN=X'0204'               ¦
¦   RESPONSE: NORMAL                        EIBRESP=0                   ¦
¦                                                          REPLY:       ¦
¦  ENTER:  CURRENT DISPLAY                                              ¦
¦  PF1 : UNDEFINED           PF2 : UNDEFINED          PF3 : UNDEFINED    ¦
¦  PF4 : SUPPRESS DISPLAYS   PF5 : WORKING STORAGE    PF6 : USER DISPLAY  ¦
¦  PF7 : SCROLL BACK         PF8 : SCROLL FORWARD     PF9 : STOP CONDITIONS ¦
¦  PF10: PREVIOUS DISPLAY    PF11: NEXT DISPLAY       PF12: UNDEFINED    ¦
¦                                                                       ¦
+-----------------------------------------------------------------------+
```

Figure 89. The HANDLE CONDITION ERROR command

```
+-----------------------------------------------------------------------+
¦                                                                       ¦
¦ TRANSACTION: AC02   PROGRAM: ACCT02  TASK NUMBER: 0000113  DISPLAY: -09  ¦
¦ STATUS:  ABOUT TO EXECUTE COMMAND                                     ¦
¦ EXEC CICS HANDLE CONDITION                                            ¦
¦  ERROR                                                                ¦
```

```
¦   OFFSET:X'000AC0'    LINE:00179          EIBFN=X'0204'                  ¦
¦   RESPONSE:                                                              ¦
¦                                                              REPLY:      ¦
¦  ENTER:  CURRENT DISPLAY                                                 ¦
¦  PF1 : UNDEFINED          PF2 : UNDEFINED         PF3 : UNDEFINED       ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS ¦
¦  PF10: PREVIOUS DISPLAY   PF11: NEXT DISPLAY      PF12: UNDEFINED       ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 90. Do the HANDLE CONDITION ERROR command

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦ TRANSACTION: AC02   PROGRAM: ACCT02  TASK NUMBER: 0000113  DISPLAY: -10  ¦
¦ STATUS:  COMMAND EXECUTION COMPLETE                                     ¦
¦ EXEC CICS DELETE                                                        ¦
¦  FILE ('ACCTFIL ')                                                      ¦
¦  RIDFLD ('11111')                                                       ¦
¦   OFFSET:X'0018CE'    LINE:00472          EIBFN=X'0608'                 ¦
¦   RESPONSE: INVREQ                        EIBRESP=16                    ¦
¦                                                        REPLY:           ¦
¦  ENTER:  CURRENT DISPLAY                                                ¦
¦  PF1 : UNDEFINED          PF2 : UNDEFINED         PF3 : UNDEFINED       ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS ¦
¦  PF10: PREVIOUS DISPLAY   PF11: NEXT DISPLAY      PF12: UNDEFINED       ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

Figure 91. Here's our failing instruction again

The delete command is returning with INVREQ.

As we said in "The COBOL code of our example application" in topic 4.0, when discussing Lines 333 to 336 of ACCT02, the problem is that we're trying to delete a record that's been read for update. Our mistake is to quote a value for the RIDFLD at this point.

We shall now press ENTER....

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦   TRANSACTION: AC02  PROGRAM: ACCT04  TASK NUMBER: 0000113  DISPLAY: 00  ¦
¦   STATUS:  AN ABEND HAS OCCURRED                                        ¦
¦      COMMAREA     = 'ACCT02  ....16'                                    ¦
¦      EIBTIME      = 123914                                              ¦
¦      EIBDATE      = 89170                                               ¦
¦      EIBTRNID     = 'AC02'                                              ¦
¦      EIBTASKN     = 113                                                 ¦
¦      EIBTRMID     = '037L'                                              ¦
¦      EIBCPOSN     = 1743                                                ¦
¦      EIBCALEN     = 14                                                  ¦
¦      EIBAID       = X'7D'                              AT X'00543F1E'   ¦
¦      EIBFN        = X'0E0C'  ABEND                     AT X'00543F1F'   ¦
¦      EIBRCODE     = X'000000000000'                    AT X'00543F21'   ¦
¦      EIBDS        = 'ACCTFIL '                                          ¦
¦  +   EIBREQID     = '........'                                          ¦
¦   ABEND :   EACC                                                        ¦
¦                                                              REPLY:     ¦
```

```
¦  ENTER:  CONTINUE                                                          ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION  ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS  ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED        ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 92. Back with our abend, EACC, again

And ENTER again...

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦   TRANSACTION: AC02   PROGRAM:       TASK NUMBER: 0000113   DISPLAY: 00  ¦
¦   STATUS:  ABNORMAL TASK TERMINATION                                     ¦
¦      EIBTIME      = 123914                                               ¦
¦      EIBDATE      = 89170                                                ¦
¦      EIBTRNID     = 'AC02'                                               ¦
¦      EIBTASKN     = 113                                                  ¦
¦      EIBTRMID     = '037L'                                               ¦
¦      EIBCPOSN     = 1743                                                 ¦
¦      EIBCALEN     = 6                                                    ¦
¦      EIBAID       = X'7D'                              AT X'00543F1E'    ¦
¦      EIBFN        = X'0A02'  WRITEQ                    AT X'00543F1F'    ¦
¦      EIBRCODE     = X'000000000000'                    AT X'00543F21'    ¦
¦      EIBDS        = 'ACCTFIL '                                           ¦
¦  +   EIBREQID     = '........'                                           ¦
¦   ABEND :   EACC                                                         ¦
¦  TO CONTINUE EDF SESSION REPLY YES                          REPLY: NO    ¦
¦  ENTER:  CONTINUE                                                        ¦
¦  PF1 : UNDEFINED          PF2 : SWITCH HEX/CHAR   PF3 : END EDF SESSION  ¦
¦  PF4 : SUPPRESS DISPLAYS  PF5 : WORKING STORAGE   PF6 : USER DISPLAY     ¦
¦  PF7 : SCROLL BACK        PF8 : SCROLL FORWARD    PF9 : STOP CONDITIONS  ¦
¦  PF10: PREVIOUS DISPLAY   PF11: UNDEFINED         PF12: UNDEFINED        ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 93. The abnormal task termination

Pressing ENTER one final time brings us to this:

```
+--------------------------------------------------------------------------+
¦                                                                          ¦
¦  ACCOUNT FILE: ERROR REPORT                                              ¦
¦  TRANSACTION  AC02  HAS FAILED IN PROGRAM  ACCT02    BECAUSE OF          ¦
¦  A PROGRAM OR FCT TABLE ERROR (INVALID FILE REQUEST).                    ¦
¦  COMMAND  DELETE               RESP  INVREQ                              ¦
¦  THE FILE IS: ACCTFIL .                                                  ¦
¦  PLEASE ASK YOUR SUPERVISOR TO CONVEY THIS INFORMATION TO THE            ¦
¦  OPERATIONS STAFF.                                                       ¦
¦  THEN PRESS "CLEAR".  THIS TERMINAL IS NO LONGER UNDER CONTROL OF        ¦
¦  THE "ACCT" APPLICATION.                                                 ¦
¦  DFH2206 12:47:47 CIDCICSC TRANSACTION AC02 HAS FAILED WITH ABEND EACC.  ¦
¦   RESOURCE BACKOUT WAS SUCCESSFUL.                                       ¦
¦                                                                          ¦
+--------------------------------------------------------------------------+
```

Figure 94. This is the CICS message. Message DFH2206 tells us that all
recoverable resources associated with the failed transaction

have been successfully backed out following the abend.

If we'd chosen **not** to suppress displays, you would have faced about another 45 screens to reach this point.

Of course, although you know the EXEC CICS DELETE command is failing, you have to go off and read the CICS/ESA Application Programming Reference carefully to pinpoint the exact reason. Studying a transaction dump leads you to the same conclusion by a different route.

The beauty of EDF as a testing tool is the way you can home in on a problem, and the way you can force your code to behave as though a problem had arisen. We hope you find EDF a useful weapon in your bug-killing armory

5.1.3.2 Temporary storage browse facility (CEBR)

We'll describe another diagnostic tool here. This is the CEBR transaction that allows you to look at temporary storage queues. If you need to do this while debugging, enter the transaction identifier CEBR to produce the display shown in Figure 95.

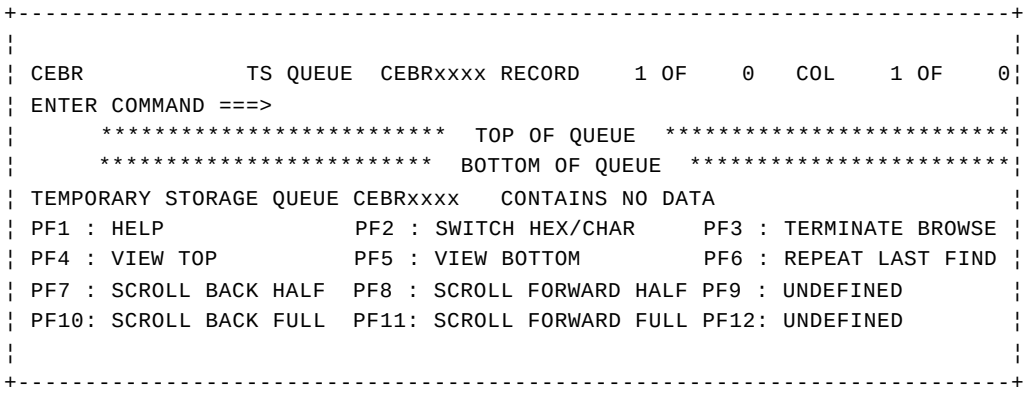


Figure 95. The temporary storage browse (CEBR) display

This shows the browse display for the temporary storage queue named "CEBRtttt" ("tttt" is the terminal identifier of the terminal from which you made the entry). Unless you happen to be interested in this particular queue (and this is unlikely), the first thing you do is to enter "QUEUE xxxxxxxx" in the command area, where "xxxxxxxx" is the name of the queue you *do* want to see. The command area is the space right after "ENTER COMMAND" at the top of the screen.

If a queue by this name exists, you'll see a display of it. The items in the queue are displayed one per line, in the area between the command line and the PF key menu. Only as much of each item as will fit on one line of the screen is shown.

Initially the display starts with the first character in the item. However, if you need to see characters beyond those displayed, you can shift the starting character by entering "COLUMN(n)" in the command area. This causes the display of each item to begin with the nth character in the item; "n" can be up to four digits.

You can tell which character the display starts at, and how long the longest item in the queue is, from the "Col X of Y" information at the top of the screen. "X" is the position of the record displayed in the first column of the screen, and "Y" is the length of the longest item. The "Line N of M" message just before that tells you that the "Nth" item in the queue is in the first one on the screen, and there are "M" items in the queue.

You can look through the items in the queue by using the scroll keys shown in the figure (PF7, PF8, PF10, and PF11), or you can specify that the display should start with a particular item in the queue. The scroll keys work just as they do for EDF. To display a particular item, enter "LINE (n)" in the command line. CEBR responds by starting the display one item before the number you specify; this number, too, can be up to four digits long.

You can redisplay the beginning of the queue either by entering "TOP" in the command area or by pressing PF4. Similarly, you can display the last screen's worth of items by entering "BOTTOM" or pressing PF5.

You can also search the items in the queue for the occurrence of a

particular character string. If you were looking for the characters "MOUNCE", for example, you would put:

```
FIND /MOUNCE
```

in the command area. CEBR would scroll the display forward until it displayed the first item that contained "MOUNCE".

The slash (/) in the command above is a delimiter. It can be any non-space character that isn't in the search string. That is,

```
FIND X05/07/89      and      FIND S05/07/89
```

are equivalent.

```
FIND /05/07/89
```

will not work, however, because there is a slash in the search string. If there are any spaces in the search string, you must repeat the delimiter at the end of the string. For example:

```
FIND /JOHN JONES/
```

Once you've entered a find command, you can repeat it (that is, find the next occurrence of the string) by pressing PF6.

You can use PF2 to switch the display from character to hexadecimal format, and back again, just like the corresponding switch hex/char command in EDF.

Indeed, you can use the CEBR transaction while under control of EDF, by using the PF key assigned for BROWSE TEMPORARY STORAGE. Your EDF transaction is suspended; CEBR starts and continues until you end it with the PF3 key. If you are in EDF, PF3 returns you to the point at which you requested CEBR. If you were not in EDF but came in by entering CEBR, PF3 terminates the transaction in the normal way, and frees the terminal for the next transaction.

The CEBR transaction also allows you to delete a temporary storage queue, by entering PURGE in the command area. And finally, there is a HELP facility, explaining how to use CEBR, which you can access by pressing

```
PF1.
```

5.2 Finding the problem

Subtopics
5.2.1 Preliminary checklist
5.2.2 Documentation
5.2.3 Reference materials
5.2.4 More testing considerations
5.2.5 Abends
5.2.6 Loops
5.2.7 Waits
5.2.8 Incorrect output
5.2.9 CICS system problems

5.2.1 Preliminary checklist

Before looking in detail at how to cope with the various classes of errors, there are some "simple" things for you to check first which may turn up a number of mistakes. For example:

1. Go back and make sure that your translator, compiler and linkage editor outputs were all error-free.

2. Check that the required PROGRAM definitions are present and correct in the CSD, and that the you have the correct entries for files in the CSD or the FCT.

3. If you are using RDO and you DEFINE or ALTER a transaction, program or mapset, then be sure to use the INSTALL option to get the changes invoked.

4. If you changed any maps, be sure that you created both a new load module (TYPE=MAP) and a new DSECT (TYPE=DSECT), and that you then recompiled every program using that new DSECT.

5. If you changed any program or mapset since CICS was last started, make sure that you are executing the most recent version, by using the transaction:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦                                                                         ¦
¦   CEMT SET PROGRAM(pgrmid) NEWCOPY                                      ¦
¦                                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

5.2.2 Documentation

Next, collect all the documentation of the problem. There are many sources of information, including:

1. Output from the translator, compiler and link editor.

2. Messages to the terminal associated with the failing transaction, and messages to the master terminal.

3. Observations from the terminal operator and the master terminal operator. In the case of the master terminal, you should note any unusual messages associated with the startup of CICS and any that occurred for some time before the actual problem.

4. Dumps. (You may not want to bother to print the dumps until you have tried other techniques. You should be prepared to do so however, because sometimes they are absolutely necessary.)

5. Shutdown statistics. These aren't usually necessary, and you should not automatically shut down your system after a transaction abend to get them. However, there are occasions on which they may give you insight into problems. Among other things, they show:

   Which transactions were used
   Which programs were executed
   Which terminals were used
   A summary of temporary storage activity
   A summary of file activity.

6. CEMT output. You can use CEMT to find out status information about files, programs, transactions, and executing tasks.

5.2.3 Reference materials

You should also collect certain reference materials for debugging. These include:

CICS/ESA Application Programming Reference . This book contains definitive information on the error conditions possible on the various commands, and on the EIB.

CICS/ESA Messages and Codes. This book describes all the "DFHxxxx" messages that CICS issues and all the CICS-generated transaction abend codes.

CICS/ESA Problem Determination Guide . This manual includes guidance on:

1. Techniques and tools for problem determination in CICS
2. Causes of waits and loops in applications, and how to solve them
3. Extended and abbreviated trace format details
4. CICS system and transaction dump format and content.

5.2.4 More testing considerations

Subtopics

5.2.4.1 Regression testing

A **regression** test is used to make sure that all the transactions in a system continue to do their processing in the same way both before and after changes are applied to the system. This is to ensure that fixes that have been applied to solve one problem don't go on to cause further problems. It's often a good idea to build a set of miniature files to perform your tests on, because it's much easier to examine a small data file for changes.

A good regression test will exercise all the code in every program -- that is, it will explore all tests and possible conditions. As your system develops to include more transactions, more possible conditions, and so on, add these to your test system to keep it in step. The results of each test should match those from the previous round of testing. Any discrepancies are grounds for suspicion. You can compare terminal output, file changes, and log entries for validity.

5.2.4.2 Single-thread testing

A **single-thread** test takes one application transaction at a time, in an otherwise "empty" CICS system, and sees how it behaves. This enables you to test the program logic, and also shows whether or not the basic CICS information (such as CSD or FCT entries) is correct. It's quite feasible to test this single application in one CICS region while your normal, online production CICS system is active in another.

5.2.4.3 Multi-thread testing

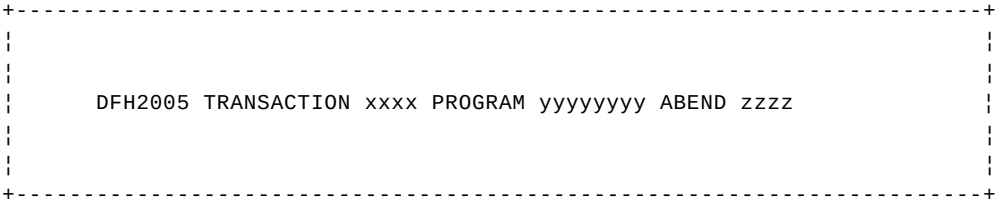
A **multi-thread** test involves several, concurrently-active transactions. Naturally, all the transactions will be in the same CICS region, so you can readily test the ability of a new transaction to co-exist with its future partners.

You may find that a transaction that sails through its single-thread testing still fails miserably in the multi-thread test. Or it may cause other transactions to fail, or even terminate CICS!

Now we can take a systematic look at abends, loops, waits, and incorrect output. We'll start with abends.

5.2.5 Abends

The message with which CICS tells you that a transaction abended:



contains several vital pieces of information. It identifies the transaction (xxxx) that failed. It tells which program (yyyyyyyy) was being executed at the time of the failure. And, most important, it indicates *which* of the many things that could go wrong did. This is the **abend** code, zzzz.

There are two kinds of abend codes: yours and CICS's. All the codes that CICS uses begin with the letter A; yours are the ones that appear in the ABCODE parameter on an ABEND command. For ease of recognition, therefore, don't start your ABCODEs with the letter A.

The first step in tracking down the cause of an abend is to look up this code. If it is one of yours, you'll know what condition it represents. From there you can look at other information (values in working storage and the sequence of calls leading up to the crash) to find out how the situation came about. For CICS abends, the place to look is the CICS/ESA Messages and Codes , which describes all of the CICS abend codes and, for many of them, has suggestions for analysis.

When you are using the subset of commands described in this Primer, you are likely to produce only a relatively small number of CICS ABENDs. With some inventiveness you could produce others, but the ones you are most likely to encounter are described under the following headings.

Subtopics
5.2.5.1 ASRA
5.2.5.2 ASRB
5.2.5.3 AICA
5.2.5.4 APCT
5.2.5.5 AFCA
5.2.5.6 AEIx and AEYx
5.2.5.7 ATNI

5.2.5.1 ASRA

To stop a simple error in one transaction from crashing the whole CICS system, CICS issues an operating system SVC to intercept abends.

So, for example, if you try to do packed arithmetic with EBCDIC variables in your COBOL code (producing what the operating system recognizes as a program check) you don't get the abend that you would in a batch program. Instead, when the operating system detects the program check, it returns control to CICS, which terminates the offending transaction with an abend of its own: ASRA. All ASRA means, therefore, is that a program has committed a violation of the program check type. In COBOL, the source of this trouble is almost always an attempt to do arithmetic with variables that are of mixed PICTURE types or that have not been initialized properly.

The first step in diagnosing an ASRA is to find out where it occurred. This means finding out the program status word (PSW) at the time of the program check. You can find this information either in a dump or by using EDF. Next, you need to know in what program it occurred, so that you can find out where in that program the offending instruction was. Usually the program is the application program that was executing at the time.

5.2.5.2 ASRB

An ASRB abend occurs in almost the same way as an ASRA, but it is the result of an operating system abend other than the common program check. If CICS can contain the damage, it terminates that transaction with ASRB. The procedure for finding the source of the trouble is the same as for ASRA. An operating system abend isn't likely to happen except as a program check in a CICS command-level program, however, and so ASRB is much less common than ASRA.

5.2.5.3 AICA

As explained earlier, an AICA abend occurs when CICS detects that an application program is looping. Whether CICS considers a program to be looping depends on the length of time that elapses between successive CICS commands. If the time is longer than the runaway task time interval (ICVR) parameter in the SIT, CICS assumes that the program is looping and terminates it with code AICA.

When you have a loop, you need to know where it is in the code. With an AICA, you know by definition that the loop started after the last CICS command was issued and ended before any other command was issued. You can tell either from the trace table in a dump, or by using EDF, what the last CICS command was and where it was in the code, and the program listing will tell you where the next one was expected. If this doesn't pinpoint the problem, look at the values of your Working-Storage variables. Often these values, in combination with your knowledge of the program logic, will tell you almost exactly how far you got in the code.

If you still need further information, however, you can use either EDF or transaction dumps to work out how far through a section of code you are getting, and what the values of the variables in Working-Storage are at each step. To do this with EDF, choose a CICS statement that you aren't sure gets executed. Using its statement number (from the translator if you used DEBUG) or its hex location otherwise, enter it as a stop condition. Then let the program run.

If the loop is far into the code, suppress the displays. If the program reaches the stop condition, then you know that the CICS statement got executed. Pick another statement beyond this one and repeat the process. If the statement does not get executed before the AICA occurs, pick another CICS statement between it and the beginning of the loop. Repeat this process until you've located the loop.

The technique with transaction dumps is very similar, except that you should pick out all the questionable statements at once, and put a DUMP command after each one, each with a different DMPCODE identifier. Then run the program and analyze the dumps. You can tell from the sequence of DMPCODEs how far you got through the code, and your Working-Storage at each point will also be available in the dumps, to help you work out what went wrong.

We'll add two notes of caution here about AICA abends.

1. Since in all but the most recent versions CICS uses real time rather than processor time to detect loops, it's possible for a transaction to get terminated, with AICA, without being in a loop. This can result from setting the runaway task time interval (ICVR) value in the SIT too low, or from too much interference with the CICS region from other regions, or a combination of both. If you've any doubt that an AICA is valid, raise the ICVR value somewhat and repeat the transaction several times. If it is a "true" AICA, the last CICS command executed will always be the same one.

2. Certain CICS commands don't pass through task control and don't, therefore, reset the runaway task time interval.

5.2.5.4 APCT

This abend occurs when you attempt to execute a program that is either (1) disabled, or (2) not defined at all in an active RDO group, or (3) the map or other load module referenced by the application cannot be found. For pure command-level programs, APCT can occur only when the first program for a transaction is invoked (before the command-level interface gets established). After that, the same type of failure (during a LINK or XCTL command, for instance) produces an AEI0 abend instead.

So, if you get an APCT, the cause is one of the following:

1. The program named in the DEFINE TRANSACTION command hasn't been defined in a DEFINE PROGRAM command.

2. The program is disabled.

Programs can be disabled by an operator or even by CICS for sufficiently unsuitable behavior. By far the most common cause, however, is that CICS could not find the program in the load library at startup time, and disabled the program for that reason. If this occurs, therefore, make sure that:

The name of the program in the load library matches the name in th CSD, and the program has been successfully linked into the library.

The program name in the DEFINE TRANSACTION command is the same as th name in the corresponding DEFINE PROGRAM command.

The program is enabled. To find out the status of the program at th time of the APCT failure, use the transaction:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦                                                                         ¦
¦   CEMT INQUIRE PROGRAM(pgrmid)                                          ¦
¦                                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

5.2.5.5 AFCA

This abend occurs when you try to use a file that has been disabled. This should happen only rarely. If the file is closed for some reason (which is more likely) and if you've not handled this condition, you'll get an AEIS abend instead. If AFCA does occur, use the CEMT transaction to find out which of the files in question is disabled:

```
+-------------------------------------------------------------------------+
¦                                                                         ¦
¦                                                                         ¦
¦   CEMT INQUIRE DATASET(fileid)                                          ¦
¦                                                                         ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The problem should disappear as soon as the file is properly available.

5.2.5.6 AEIx and AEYx

All of the abend codes that start with the letters "AEI" or "AEY" result from exceptional conditions detected in command-level programs, for which no HANDLE CONDITION command is active.

Figure 96 lists all of the AEIx and AEYx abends that may occur using the commands described in this Primer. After each code the figure shows the exceptional condition, and also the command type (such as file or BMS), and the associated EIBFN and EIBRCODE values.

Code	Condition	Service	EIBFN	EIBRCODE
AEIA	ERROR	Misc	N/A	N/A
AEIK	TERMIDERR	Time	10	12
AEIL	FILENOTFOUND	File	06	01
AEIM	NOTFND	File	06	81
		or Time	10	81
AEIN	DUPREC	File	06	82
AEIP	INVREQ	File	06	08
		or Temp Stge	0A	20
		or Program	0E	E0
AEIQ	IOERR	File	06	80
AEIR	NOSPACE	File	06	83
		or Temp Stge	0A	08
AEIS	NOTOPEN	File	06	0C
AEIT	ENDFILE	File	06	0F
AEIU	ILLOGIC	File	06	02
AEIV	LENGERR	File	06	E1
		or Temp Stge	0A	E1
		or Time	10	E1
AEIZ	ITEMERR	Temp Stge	0A	01
AEI0	PGMIDERR	Program	0E	01
AEI1	TRANSIDERR	Time	10	11
AEI3	INVTSREQ	Time	10	14
AEI8	IOERR	Temp Stge	0A	04
		or Time	10	04
AEI9	MAPFAIL	BMS	18	04
AEYB	INVMPSZ	BMS	18	08
AEYH	QIDERR	Temp Stge	0A	02

Figure 96. AEIx and AEIy abend conditions

For the most part, the reasons for these abends are exactly what is stated in the CICS/ESA Application Programming Reference for the corresponding condition. Some of the errors may have multiple causes, such as ILLOGIC and INVREQ. For example, on an ILLOGIC abend, byte 1 of EIBRCODE is the VSAM return code and byte 2 is the VSAM error code.

If you determine that the condition was the result of a logic error in the program, then you can correct that error and retry. If, however, it turns out that the condition could arise naturally, then you should add a HANDLE CONDITION command to the program to deal with it.

5.2.5.7 ATNI

A terminal error will lead to an ATNI transaction abend, and a CICS transaction dump. In other words, the application will not get control back, and contact with the screen will be lost.

5.2.6 Loops

We've already described a technique for finding loops that do not contain any CICS commands. (It was in the discussion of AICA abends, and involved using either EDF or transaction dumps.) For loops that *do* include CICS commands, the same tools apply.

Using EDF, the easiest method is to invoke the transaction and let it run until you're satisfied that it is looping. Then go to another terminal and invoke EDF for the terminal running the suspect transaction. EDF will interrupt the execution of the transaction at every CICS command, and send a display to this second terminal. As each command is executed, note it in the associated program listing. Let the program continue executing commands until a clear pattern of repetition emerges.

Having located the loop, the next step is to find the cause. There will usually be one or more points in the loop at which the program should exit, provided certain conditions are met. The problem is that the conditions are never met. When, under EDF, you reach the command that is causing the problem, you may need to examine the values in Working-Storage to find out why this is occurring. The next time the loop is executed, you may want to pause at the preceding command and look at the same variables at that time. If there's too much code between these two commands to see exactly what's going wrong, you can then use the techniques for the other kind of loops (AICA abends) to locate the error within the CICS statements between the CICS commands.

The process is very similar using a transaction dump. Let the transaction run until it's clearly looping, and then cancel it. Use the trace table in the resulting abend dump to find the repeated sequence of CICS commands. At this point the contents of Working-Storage may or may not give you enough information to work out the problem. If they do not, put further dump requests near the expected exit point(s) from the loop, and use the technique described above to close in on the problem.

5.2.7 Waits

Remember we're assuming you have a batch programming background.

With that in mind, you can avoid WAITs by avoiding two programming practices you may be bringing with you from that background. You see, the most common cause of a WAIT in a COBOL program is an ACCEPT FROM CONSOLE or STOP statement to which the operator failed to reply. Check for these before going any further with your debugging of a wait.

Now, what about approaching WAITs from a CICS point of view?

The key to recognizing a wait is the operator's observation. In other words, he or she has typed in some data, pressed the ENTER key, and nothing much seems to be happening.

When you first suspect a wait, use the CEMT transaction to make sure there **is** still a task associated with the terminal. If there isn't, you've got an "incorrect output". A waiting task will show as suspended or active.

If we leave aside the question of database access (as beyond the scope of the Primer), there are then just five reasons for a task to get suspended:

Terminal control wai

Unsuccessful enqueue -- when a task needs, but has failed to gai access to, a resource owned by some other task

Interval control wai

Not enough main storag

Not enough auxiliary storage

There are a further four reasons for a task to be active but waiting:

Dispatchabl

Dispatchable, but on the point of an ABEND comman

Non-dispatchable, because of too many other tasks in the system, o some other CICS workload control

Waiting for some external or internal event to complete (for example file input/output or no VSAM string available, respectively).

Whatever the case, purge the task and print the dump. Work through the dump to find the last CALL made by the program. If the troublesome task was suspended, look for the KCP SUSPEND trace table entry. Just before this should be a clue to the reason for the suspend, bearing in mind the above five reasons.

If, on the other hand, the task was active, look for the KCP WAIT trace table entry. Just before this should be a clue to the reason for the wait.

Between them, the source code of the last CALL and the request causing either the wait or the suspend should cast some light on the problem.

Of course, the problem may be entirely outside your task. There are two reasons for the CICS region itself to be in a wait state:

1. No CICS tasks are currently ready to be dispatched, so task control has issued an operating system wait for the length of time specified by the ICV (a SIT operand that basically says how long CICS is to give

up control).

2. A wait has been issued from somewhere else in CICS, or an SVC (supervisor call) has been issued.

In the first case, you must check each task to find out what it's waiting for. There may also be some reason why **new** tasks aren't coming along. The system could be short on storage; or the maximum number of concurrent tasks allowed could have been reached; or terminal input could be failing to get through.

In the second case, you must find out what's going on in the operating system and also, perhaps, confirm that a badly-behaved task hasn't issued an SVC. During normal running, CICS issues only the task control operating system wait we mentioned above.

5.2.8 Incorrect output

As we've said, the symptoms of incorrect output are garbage on the screen (or printer), a terminal that simply locks up, bad data in files, or wrong screen sequences. In fact, incorrect output problems can present all kinds of bothersome symptoms and be very interesting to pin down.

Here are some suggestions for you to think about when you have a program that's compiled correctly but that seems to misbehave:

Is the input data correct

Are you correctly validating entered data

Assuming you **are** getting **some** output at the terminal or printed out, check it over:

- Is the sequence what you expect?
- Are the items correct?
- Are any totals correct?
- Are some items being repeated when they shouldn't be?
- Are any items missing?

Print any output files, data files, and so on to see if they contai what you expect.

Are you initializing or clearing program variables properly

Be sure to look up any messages or codes that come up. Work through program dump listings to see what command last executed. (Note, however, that an operation that uncovers incorrect output may be completely innocent of having caused it.)

Try to find out what resource is failing. It's usually data on a disk (on a clear disk, you can seek forever!) or data in a terminal data stream. Of course, data on the terminal may be bad because of a bad file.

Work back, if possible, from the place where the symptoms first occur, and forward from a point where the data is OK. Where you meet should be interesting.

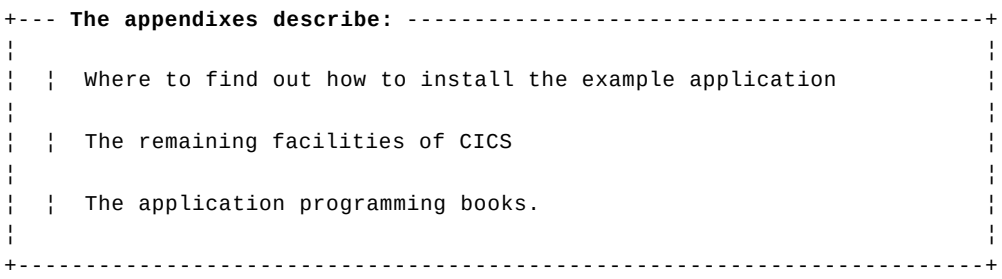
Look at map or file data structures from appropriate listings. Compare each field, as defined in the output from the map assembly, with the map as displayed in working storage. You can use EDF to do this, or a transaction dump. Note the contents of each field carefully, and look at each field suspiciously.

Paranoid patience is sometimes the best approach. Good luck!

5.2.9 CICS system problems

Problems that affect CICS as a system fall into the same four categories as those which affect transactions: abends, loops, waits, and incorrect output. As noted before, such problems are generally beyond the scope of this Primer.

6.0 Appendixes



A.0 Appendix A. Getting the application into your CICS system

Subtopics

A.1 Introduction

Your systems programmer will probably have to help you get the application into your CICS system. You'll need a copy of the CICS/ESA Installation Guide to refer to for guidance on doing so. You may also need to refer to the CICS/ESA System Definition Guide if you want to have more background guidance information about installing COBOL application programs.

A.2 What has to be done?

The COBOL source code for the application programs is supplied on the CICS distribution tape. You'll find the application source code in the following members of the **CICS330.SAMPLIB** library:

Table 1. Source code members

Name as supplied	***Primer name***	***Description***
DFHXSET	**ACCTSET**	Map definitions for 3270 displays and print
DFHXS00	**ACCT00**	Display menu
DFHXS01	**ACCT01**	Initial request processing
DFHXS02	**ACCT02**	Update processing
DFHXS03	**ACCT03**	Requests for printing
DFHXS04	**ACCT04**	Error processing
DFHXSREC	**ACCTREC**	Layout of account record
DFHXSIXR	**ACIXREC**	Layout of index record
DFHXSINX	**ACCTINDX**	Index file recovery (batch program)

Note:

> For an illustration of the data structure created when assembling mapset ACCTSET, see "The result of the SYSPARM=DSECT assembly" in topic A.2.1.

The example application uses VSAM files and 3270 display and printer terminals.

Before you can run the application, you have to prepare the mapset and programs for execution, define all the required resources to CICS, and define the VSAM files. If you are using CICS/ESA, you'll find general guidance about installing mapsets and programs, in the CICS/ESA System Definition Guide. For guidance on defining VSAM files and an example of JCL needed to do so, see the CICS/ESA Installation Guide.

Note that **ACCTINDX** is not required for normal online execution of the application. See "Recovery requirements" in topic 2.4.2.

Subtopics
A.2.1 The result of the SYSPARM=DSECT assembly

A.2.1 The result of the SYSPARM=DSECT assembly

The following example shows the data structure created when assembling mapset ACCTSET.

```
+------------------------------------------------------------------------+
¦                                                                        ¦
¦                                                                        ¦
¦          01  ACCTMNUI.                                                 ¦
¦              02  FILLER PIC X(12).                                     ¦
¦              02  SNAMEML    COMP  PIC  S9(4).                         ¦
¦              02  SNAMEMF    PICTURE X.                                 ¦
¦              02  FILLER REDEFINES SNAMEMF.                             ¦
¦                03 SNAMEMA    PICTURE X.                               ¦
¦              02  SNAMEMI  PIC X(12).                                   ¦
¦              02  FNAMEML    COMP  PIC  S9(4).                         ¦
¦              02  FNAMEMF    PICTURE X.                                 ¦
¦              02  FILLER REDEFINES FNAMEMF.                             ¦
¦                03 FNAMEMA    PICTURE X.                               ¦
¦              02  FNAMEMI  PIC X(7).                                    ¦
¦              02  REQML    COMP  PIC  S9(4).                          ¦
¦              02  REQMF    PICTURE X.                                 ¦
¦              02  FILLER REDEFINES REQMF.                               ¦
¦                03 REQMA    PICTURE X.                                ¦
¦              02  REQMI  PIC X(1).                                      ¦
¦              02  ACCTML    COMP  PIC  S9(4).                          ¦
¦              02  ACCTMF    PICTURE X.                                 ¦
¦              02  FILLER REDEFINES ACCTMF.                              ¦
¦                03 ACCTMA    PICTURE X.                                ¦
¦              02  ACCTMI  PIC X(5).                                     ¦
¦              02  PRTRML    COMP  PIC  S9(4).                          ¦
¦              02  PRTRMF    PICTURE X.                                 ¦
¦              02  FILLER REDEFINES PRTRMF.                              ¦
¦                03 PRTRMA    PICTURE X.                                ¦
¦              02  PRTRMI  PIC X(4).                                     ¦
¦              02  SUMTTLML    COMP  PIC  S9(4).                         ¦
¦              02  SUMTTLMF    PICTURE X.                                ¦
¦              02  FILLER REDEFINES SUMTTLMF.                            ¦
¦                03 SUMTTLMA    PICTURE X.                              ¦
¦              02  SUMTTLMI  PIC X(79).                                  ¦
¦              02  SUMLNMD OCCURS 6 TIMES.                               ¦
¦                03  SUMLNML    COMP  PIC  S9(4).                        ¦
¦                03  SUMLNMF    PICTURE X.                               ¦
¦                03  SUMLNMI  PIC X(79).                                 ¦
¦              02  MSGML    COMP  PIC  S9(4).                            ¦
¦              02  MSGMF    PICTURE X.                                   ¦
¦              02  FILLER REDEFINES MSGMF.                               ¦
¦                03 MSGMA    PICTURE X.                                  ¦
¦              02  MSGMI  PIC X(60).                                     ¦
¦           01  ACCTMNUO REDEFINES ACCTMNUI.                              ¦
¦              02  FILLER PIC X(12).                                     ¦
¦              02  FILLER PICTURE X(3).                                  ¦
¦              02  SNAMEMO  PIC X(12).                                   ¦
¦              02  FILLER PICTURE X(3).                                  ¦
¦              02  FNAMEMO  PIC X(7).                                    ¦
¦              02  FILLER PICTURE X(3).                                  ¦
¦              02  REQMO  PIC X(1).                                      ¦
¦              02  FILLER PICTURE X(3).                                  ¦
¦              02  ACCTMO  PIC X(5).                                     ¦
¦              02  FILLER PICTURE X(3).                                  ¦
¦              02  PRTRMO  PIC X(4).                                     ¦
¦              02  FILLER PICTURE X(3).                                  ¦
¦              02  SUMTTLMO  PIC X(79).                                  ¦
¦              02  DFHMS1 OCCURS 6 TIMES.                                ¦
¦                03  FILLER PICTURE X(2).                                ¦
```

```
          03  SUMLNMA    PICTURE X.
          03  SUMLNMO  PIC X(79).
        02  FILLER PICTURE X(3).
        02  MSGMO  PIC X(60).
01  ACCTDTLI.
        02  FILLER PIC X(12).
        02  TITLEDL    COMP  PIC  S9(4).
        02  TITLEDF    PICTURE X.
        02  FILLER REDEFINES TITLEDF.
          03 TITLEDA    PICTURE X.
        02  TITLEDI  PIC X(14).
        02  ACCTDL    COMP  PIC  S9(4).
        02  ACCTDF    PICTURE X.
        02  FILLER REDEFINES ACCTDF.
          03 ACCTDA    PICTURE X.
        02  ACCTDI  PIC X(5).
        02  SNAMEDL    COMP  PIC  S9(4).
        02  SNAMEDF    PICTURE X.
        02  FILLER REDEFINES SNAMEDF.
          03 SNAMEDA    PICTURE X.
        02  SNAMEDI  PIC X(18).
        02  FNAMEDL    COMP  PIC  S9(4).
        02  FNAMEDF    PICTURE X.
        02  FILLER REDEFINES FNAMEDF.
          03 FNAMEDA    PICTURE X.
        02  FNAMEDI  PIC X(12).
        02  MIDL    COMP  PIC  S9(4).
        02  MIDF    PICTURE X.
        02  FILLER REDEFINES MIDF.
          03 MIDA    PICTURE X.
        02  MIDI  PIC X(1).
        02  TTLDL    COMP  PIC  S9(4).
        02  TTLDF    PICTURE X.
        02  FILLER REDEFINES TTLDF.
          03 TTLDA    PICTURE X.
        02  TTLDI  PIC X(4).
        02  TELDL    COMP  PIC  S9(4).
        02  TELDF    PICTURE X.
        02  FILLER REDEFINES TELDF.
          03 TELDA    PICTURE X.
        02  TELDI  PIC X(10).
        02  ADDR1DL    COMP  PIC  S9(4).
        02  ADDR1DF    PICTURE X.
        02  FILLER REDEFINES ADDR1DF.
          03 ADDR1DA    PICTURE X.
        02  ADDR1DI  PIC X(24).
        02  ADDR2DL    COMP  PIC  S9(4).
        02  ADDR2DF    PICTURE X.
        02  FILLER REDEFINES ADDR2DF.
          03 ADDR2DA    PICTURE X.
        02  ADDR2DI  PIC X(24).
        02  ADDR3DL    COMP  PIC  S9(4).
        02  ADDR3DF    PICTURE X.
        02  FILLER REDEFINES ADDR3DF.
          03 ADDR3DA    PICTURE X.
        02  ADDR3DI  PIC X(24).
        02  AUTH1DL    COMP  PIC  S9(4).
        02  AUTH1DF    PICTURE X.
        02  FILLER REDEFINES AUTH1DF.
          03 AUTH1DA    PICTURE X.
        02  AUTH1DI  PIC X(32).
        02  AUTH2DL    COMP  PIC  S9(4).
        02  AUTH2DF    PICTURE X.
        02  FILLER REDEFINES AUTH2DF.
          03 AUTH2DA    PICTURE X.
```

```
02  AUTH2DI  PIC X(32).
02  AUTH3DL    COMP  PIC  S9(4).
02  AUTH3DF    PICTURE X.
02  FILLER REDEFINES AUTH3DF.
  03 AUTH3DA    PICTURE X.
02  AUTH3DI  PIC X(32).
02  AUTH4DL    COMP  PIC  S9(4).
02  AUTH4DF    PICTURE X.
02  FILLER REDEFINES AUTH4DF.
  03 AUTH4DA    PICTURE X.
02  AUTH4DI  PIC X(32).
02  CARDSDL    COMP  PIC  S9(4).
02  CARDSDF    PICTURE X.
02  FILLER REDEFINES CARDSDF.
  03 CARDSDA    PICTURE X.
02  CARDSDI  PIC X(1).
02  IMODL    COMP  PIC  S9(4).
02  IMODF    PICTURE X.
02  FILLER REDEFINES IMODF.
  03 IMODA    PICTURE X.
02  IMODI  PIC X(2).
02  IDAYDL    COMP  PIC  S9(4).
02  IDAYDF    PICTURE X.
02  FILLER REDEFINES IDAYDF.
  03 IDAYDA    PICTURE X.
02  IDAYDI  PIC X(2).
02  IYRDL    COMP  PIC  S9(4).
02  IYRDF    PICTURE X.
02  FILLER REDEFINES IYRDF.
  03 IYRDA    PICTURE X.
02  IYRDI  PIC X(2).
02  RSNDL    COMP  PIC  S9(4).
02  RSNDF    PICTURE X.
02  FILLER REDEFINES RSNDF.
  03 RSNDA    PICTURE X.
02  RSNDI  PIC X(1).
02  CCODEDL    COMP  PIC  S9(4).
02  CCODEDF    PICTURE X.
02  FILLER REDEFINES CCODEDF.
   03 CCODEDA    PICTURE X.
02  CCODEDI  PIC X(1).
02  APPRDL    COMP  PIC  S9(4).
02  APPRDF    PICTURE X.
02  FILLER REDEFINES APPRDF.
   03 APPRDA    PICTURE X.
02  APPRDI  PIC X(3).
02  SCODE1DL    COMP  PIC  S9(4).
02  SCODE1DF    PICTURE X.
02  FILLER REDEFINES SCODE1DF.
  03 SCODE1DA    PICTURE X.
02  SCODE1DI  PIC X(1).
02  SCODE2DL    COMP  PIC  S9(4).
02  SCODE2DF    PICTURE X.
02  FILLER REDEFINES SCODE2DF.
  03 SCODE2DA    PICTURE X.
02  SCODE2DI  PIC X(1).
02  SCODE3DL    COMP  PIC  S9(4).
02  SCODE3DF    PICTURE X.
02  FILLER REDEFINES SCODE3DF.
  03 SCODE3DA    PICTURE X.
02  SCODE3DI  PIC X(1).
02  STATTLDL    COMP  PIC  S9(4).
02  STATTLDF    PICTURE X.
02  FILLER REDEFINES STATTLDF.
  03 STATTLDA    PICTURE X.
```

```
           02  STATTLDI  PIC X(15).
           02  STATDL    COMP  PIC  S9(4).
           02  STATDF    PICTURE X.
           02  FILLER REDEFINES STATDF.
             03 STATDA    PICTURE X.
           02  STATDI  PIC X(2).
           02  LIMTTLDL    COMP  PIC  S9(4).
           02  LIMTTLDF    PICTURE X.
           02  FILLER REDEFINES LIMTTLDF.
             03 LIMTTLDA    PICTURE X.
           02  LIMTTLDI  PIC X(18).
           02  LIMITDL    COMP  PIC  S9(4).
           02  LIMITDF    PICTURE X.
           02  FILLER REDEFINES LIMITDF.
             03 LIMITDA    PICTURE X.
           02  LIMITDI  PIC X(8).
           02  HISTTLDL    COMP  PIC  S9(4).
           02  HISTTLDF    PICTURE X.
           02  FILLER REDEFINES HISTTLDF.
             03 HISTTLDA    PICTURE X.
           02  HISTTLDI  PIC X(71).
           02  HIST1DL    COMP  PIC  S9(4).
           02  HIST1DF    PICTURE X.
           02  FILLER REDEFINES HIST1DF.
             03 HIST1DA    PICTURE X.
           02  HIST1DI  PIC X(61).
           02  HIST2DL    COMP  PIC  S9(4).
           02  HIST2DF    PICTURE X.
           02  FILLER REDEFINES HIST2DF.
             03 HIST2DA    PICTURE X.
           02  HIST2DI  PIC X(61).
           02  HIST3DL    COMP  PIC  S9(4).
           02  HIST3DF    PICTURE X.
           02  FILLER REDEFINES HIST3DF.
             03 HIST3DA    PICTURE X.
           02  HIST3DI  PIC X(61).
           02  MSGDL    COMP  PIC  S9(4).
           02  MSGDF    PICTURE X.
           02  FILLER REDEFINES MSGDF.
             03 MSGDA    PICTURE X.
           02  MSGDI  PIC X(60).
           02  VFYDL    COMP  PIC  S9(4).
           02  VFYDF    PICTURE X.
           02  FILLER REDEFINES VFYDF.
             03 VFYDA    PICTURE X.
           02  VFYDI  PIC X(1).
       01  ACCTDTLO REDEFINES ACCTDTLI.
           02  FILLER PIC X(12).
           02  FILLER PICTURE X(3).
           02  TITLEDO  PIC X(14).
           02  FILLER PICTURE X(3).
           02  ACCTDO  PIC X(5).
           02  FILLER PICTURE X(3).
           02  SNAMEDO  PIC X(18).
           02  FILLER PICTURE X(3).
           02  FNAMEDO  PIC X(12).
           02  FILLER PICTURE X(3).
           02  MIDO  PIC X(1).
           02  FILLER PICTURE X(3).
           02  TTLDO  PIC X(4).
           02  FILLER PICTURE X(3).
           02  TELDO  PIC X(10).
           02  FILLER PICTURE X(3).
           02  ADDR1DO  PIC X(24).
           02  FILLER PICTURE X(3).
```

```
           02  ADDR2DO  PIC X(24).
           02  FILLER PICTURE X(3).
           02  ADDR3DO  PIC X(24).
           02  FILLER PICTURE X(3).
           02  AUTH1DO  PIC X(32).
           02  FILLER PICTURE X(3).
           02  AUTH2DO  PIC X(32).
           02  FILLER PICTURE X(3).
           02  AUTH3DO  PIC X(32).
           02  FILLER PICTURE X(3).
           02  AUTH4DO  PIC X(32).
           02  FILLER PICTURE X(3).
           02  CARDSDO  PIC X(1).
           02  FILLER PICTURE X(3).
           02  IMODO  PIC X(2).
           02  FILLER PICTURE X(3).
           02  IDAYDO  PIC X(2).
           02  FILLER PICTURE X(3).
           02  IYRDO  PIC X(2).
           02  FILLER PICTURE X(3).
           02  RSNDO  PIC X(1).
           02  FILLER PICTURE X(3).
           02  CCODEDO  PIC X(1).
           02  FILLER PICTURE X(3).
           02  APPRDO  PIC X(3).
           02  FILLER PICTURE X(3).
           02  SCODE1DO  PIC X(1).
           02  FILLER PICTURE X(3).
           02  SCODE2DO  PIC X(1).
           02  FILLER PICTURE X(3).
           02  SCODE3DO  PIC X(1).
           02  FILLER PICTURE X(3).
           02  STATTLDO  PIC X(15).
           02  FILLER PICTURE X(3).
           02  STATDO  PIC X(2).
           02  FILLER PICTURE X(3).
           02  LIMTTLDO  PIC X(18).
           02  FILLER PICTURE X(3).
           02  LIMITDO  PIC X(8).
           02  FILLER PICTURE X(3).
           02  HISTTLDO  PIC X(71).
           02  FILLER PICTURE X(3).
           02  HIST1DO  PIC X(61).
           02  FILLER PICTURE X(3).
           02  HIST2DO  PIC X(61).
           02  FILLER PICTURE X(3).
           02  HIST3DO  PIC X(61).
           02  FILLER PICTURE X(3).
           02  MSGDO  PIC X(60).
           02  FILLER PICTURE X(3).
           02  VFYDO  PIC X(1).
       01  ACCTERRI.
           02  FILLER PIC X(12).
           02  TRANEL    COMP  PIC  S9(4).
           02  TRANEF    PICTURE X.
           02  FILLER REDEFINES TRANEF.
             03 TRANEA    PICTURE X.
           02  TRANEI  PIC X(4).
           02  PGMEL    COMP  PIC  S9(4).
           02  PGMEF    PICTURE X.
           02  FILLER REDEFINES PGMEF.
             03 PGMEA    PICTURE X.
           02  PGMEI  PIC X(8).
           02  RSNEL    COMP  PIC  S9(4).
           02  RSNEF    PICTURE X.
```

```
        02  FILLER REDEFINES RSNEF.
          03 RSNEA    PICTURE X.
        02  RSNEI  PIC X(60).
        02  CMDEL    COMP  PIC  S9(4).
        02  CMDEF    PICTURE X.
        02  FILLER REDEFINES CMDEF.
          03 CMDEA    PICTURE X.
        02  CMDEI  PIC X(20).
        02  RESPEL    COMP  PIC  S9(4).
        02  RESPEF    PICTURE X.
        02  FILLER REDEFINES RESPEF.
          03 RESPEA    PICTURE X.
        02  RESPEI  PIC X(12).
        02  FILEEL    COMP  PIC  S9(4).
        02  FILEEF    PICTURE X.
        02  FILLER REDEFINES FILEEF.
          03 FILEEA    PICTURE X.
        02  FILEEI  PIC X(22).
    01  ACCTERRO REDEFINES ACCTERRI.
        02  FILLER PIC X(12).
        02  FILLER PICTURE X(3).
        02  TRANEO  PIC X(4).
        02  FILLER PICTURE X(3).
        02  PGMEO  PIC X(8).
        02  FILLER PICTURE X(3).
        02  RSNEO  PIC X(60).
        02  FILLER PICTURE X(3).
        02  CMDEO  PIC X(20).
        02  FILLER PICTURE X(3).
        02  RESPEO  PIC X(12).
        02  FILLER PICTURE X(3).
        02  FILEEO  PIC X(22).
    01  ACCTMSGI.
        02  FILLER PIC X(12).
        02  MSGL    COMP  PIC  S9(4).
        02  MSGF    PICTURE X.
        02  FILLER REDEFINES MSGF.
          03 MSGA    PICTURE X.
        02  MSGI  PIC X(79).
    01  ACCTMSGO REDEFINES ACCTMSGI.
        02  FILLER PIC X(12).
        02  FILLER PICTURE X(3).
        02  MSGO  PIC X(79).
```

Figure 97. Result of the SYSPARM=DSECT assembly

B.0 Appendix B. Other CICS facilities

The aim of this appendix is to mention the CICS facilities we haven't covered in the Primer, and to introduce you to the other application programming books in the CICS library.

Subtopics

B.1 Other CICS facilities

In no particular order, these include:

Getting access to control blocks and control information using ADDRES and ASSIGN commands.

The ADDRESS command gives you access to the common storage area (CSA), the common work area (CWA), the transaction work area (TWA), and so on.

The ASSIGN command allows you to get values from outside the local environment of your application program. For example, lengths of storage areas, values needed during BMS operations, information about terminal characteristics, and so on.

The use of the command interpreter transaction, CECI (which we ver briefly met in "Optional exercise" in topic 3.2.5).

CECI is very useful for deleting, repairing, inspecting, and creating all sorts of items. We use it in the EDF session to delete our temporary storage scratchpad record (to save waiting 10 minutes).

The DL/I interface

DL/I is a general-purpose database control system. CICS application programs can access DL/I databases using EXEC DLI... commands.

The DATABASE 2 interface (MVS)

DATABASE 2 is a relational database control system. CICS application programs can access DB2 databases using EXEC SQL... commands. You can find more guidance about it in the CICS/ESA Release Guide.

Terminal operations that don't use BMS

This means using native terminal control commands.

Batch Data Interchange

The CICS batch data interchange program allows your application to talk to programmable subsystems such as the IBM 8100.

The task control commands, SUSPEND, ENQ, and DEQ

The SUSPEND command allows you to give up control and allow other, higher priority, tasks to run. The task from which you issue the SUSPEND gets control back as soon as all higher priority tasks that can run have "had their turn".

ENQ (enqueue) tells CICS that a given task wants a particular resource (of the one-user-at-a-time type). CICS returns control to the task when the resource becomes available.

Similarly, DEQ (dequeue) tells CICS that a given task has finished with such a resource.

The storage control commands, GETMAIN, FREEMAIN

The GETMAIN command gets a specified amount of main storage. If you want, it can also initialize the contents of that storage to a particular bit configuration.

The FREEMAIN command, as you'd expect, releases such storage.

User journal operations

The CICS journal control facilities allow you to direct any information you want to special-purpose sequential data sets (called **journals**). These journals are to help you reconstruct events or data changes for both audit purposes and in case of system failures.

Syncpoint command

The SYNCPOINT command allows you to divide a task--usually a long-running one--into smaller units known as logical units of work. This makes it easier to recover from a task abend or a system failure.

The DUMP command (which we mentioned on 3.8.3).

This allows you to dump specified main storage areas without terminating your program, as you do with an ABEND command. You can dump the same areas as appear in a transaction abend dump, and/or other areas too.

Trace commands

CICS trace control uses a trace table (which is included for guidance in the CICS/ESA Problem Determination Guide). You can put your own entries into this table for use as "flags" to help you spot what your application program is doing.

The monitor program and its exits

You can define user event monitoring points (EMPs). At each user EMP you can accumulate all sorts of information of an accounting and performance nature. You'll find more details of this part of the programming interface in the CICS/ESA Customization Guide. For definitive application programming interface information on EXEC CICS MONITOR commands, see the CICS/ESA Application Programming Reference.

Intersystem communication (ISC) and multi-region operation (MRO facilities.

These allow independent CICS systems to talk to each other. The systems may be in the same processor or in different processors. Guidance on using ISC and MRO is in the CICS/ESA Intercommunication Guide.

Exits within the management modules

You may, despite the many options that CICS offers, still have special requirements that a standard CICS system cannot meet. In this case, you can add your own **user exit** code to certain CICS modules. This code will then be invoked whenever one of these modules is used.

See the CICS/ESA Customization Guide for product-sensitive programming interface information on user exits.

Transaction restart facilities

The CICS/ESA Recovery and Restart Guide gives you guidance on designing applications with recovery in mind.

Program error programs (PEPs), node error programs (NEPs), an terminal error programs (TEPs).

IBM supplies programs to handle certain common error situations. There's one to handle program errors (PEP), one for VTAM terminal errors (NEP), and one for non-VTAM terminal errors (TEP). If you

prefer, you can supply your own versions of these programs, and tailor the processing of certain types of error for your particular needs. There are special macros to help you build your own versions of these programs. Either way, please see the CICS/ESA Customization Guide for product-sensitive programming interface information.

The external security interface

CICS offers you an interface to an external security manager. You can write this yourself or, if you use MVS, you can use the Resource Access Control Facility (RACF) (*) program product.

See the CICS/ESA Customization Guide for guidance on doing so.

Dynamic OPEN and CLOSE

This facility allows you to open and close data sets dynamically while CICS is running. Again, see the CICS/ESA Customization Guide for programming interface information.

The phonetic key routine

CICS provides a subroutine to convert words to a condensed "phonetic" form. The major use of phonetic codes is for keys to data sets (usually names), so that you can access records in the file without knowing the exact spelling of a name or a word in the file key. We might have chosen to use this subroutine in building the name index to our account file. See the CICS/ESA Customization Guide for product-sensitive programming interface information.

The master terminal operator (CEMT) transaction application interface

The master terminal (CEMT) transaction functions are also available to an application program. See the CICS/ESA CICS-Supplied Transactions manual for product-sensitive programming interface information.

(*) IBM Trademark. For a complete list of trademarks, see "Notices" in topic FRONT_1.

B.2 The Application Programming Guide

This is the next book you should read.

It contains programming design guidance, particularly in the vital area of performance, debugging, and testing CICS applications.

It also describes the variety of sample material that is available as part of your CICS system. This material contains some useful coding hints, tips, and techniques.

B.3 The Application Programmer's Reference

This is your authoritative source of information about the command-level application programming interface.

You'll find within it all that you need to know about every CICS command-level instruction.

GLOSSARY Glossary

This glossary defines special CICS terms used in the library and words used with other than their everyday meaning. It includes terms and definitions from the *IBM Vocabulary for Data Processing, Telecommunications, and Office Systems*, GC20-1699. In some cases the definition given isn't the only one applicable to the term, but gives the particular sense in which we've used it.

American National Standards Institute (ANSI) definitions are preceded by an asterisk.

+---+
¦ **A** ¦
+---+

abend. Abnormal end of task.

access method. A technique for moving data between main storage and input/output devices.

application. This refers to a set of one or more **application units of work** designed to fulfill a particular need (or needs) of the user organization.

application program. (1) A program written for or by a user that applies to the user's work. (2) In data communication, a program used to connect and communicate with stations in a network, enabling users to perform application-oriented activities.

auxiliary storage. Data storage other than main storage; for example, storage on magnetic tape or direct access devices.

+---+
¦ **B** ¦
+---+

backout. A general term meaning to restore a previous state of all or part of a system. See dynamic transaction backout.

Basic Mapping Support (BMS). A facility that moves data streams to and from a terminal. It provides device independence and format independence for application programs.

batch. An accumulation of data to be processed.

blanks. See space.

BMS. See Basic Mapping Support.

byte. In System/370, a sequence of eight adjacent binary digits that are operated on as a unit.

+---+
¦ **C** ¦
+---+

CEMT. The master terminal transaction.

CI. See control interval.

CICS. Customer Information Control System

CICS system definition file (CSD). CSD In CICS, the CSD is a file that contains a resource definition record for every resource defined to CICS using RDO.

COBOL. Common business-oriented language. An English-like programming language designed for business data processing applications.

command. In CICS, an instruction similar in format to a high-level programming language statement. (Contrast with macro.) CICS commands invariably include the verb EXECUTE (or EXEC). They may be issued by an application program to make use of CICS facilities.

command-language statement. In CICS, synonym for command.

common system area (CSA). In CICS, a basic system control block to which all transactions have access.

*** concurrent**. Pertaining to the occurrence of two or more activities within a given interval of time.

control block. In CICS, a storage area used to hold dynamic data during the execution of control programs and application programs. Synonym for control area. Contrast with control table.

control interval (CI). A fixed-length area of direct access storage in which VSAM stores records. The unit of information that VSAM transmits to or from direct access storage.

control table. In CICS, a set of information used to define or describe the configuration or operation of the system in a relatively permanent way. Contrast with control block.

conversational. Pertaining to a program or a system that carries on a dialogue with a terminal user, alternately accepting input and then responding to the input quickly enough for the user to maintain his or her train of thought.

CSA. Common system area.

CSD. CICS system definition file.

CWA. Common work area, an extension of the common system area (CSA).

```
+---+
¦ D ¦
+---+
```

DAM. Direct access method.

DASD. Direct access storage device.

database. An organized collection of interrelated or independent data items stored together without unnecessary redundancy, to serve one or more applications.

data set. The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access. See file.

DB2. DATABASE 2, IBM's relational database management system program product for the MVS environment.

DB/DC. Data-base and data-communication.

deadlock. (1) Unresolved contention for the use of a resource. (2) An error condition in which processing cannot continue because each of two elements of the process is waiting for an action by, or a response from, the other.

device independence. An application program written in such a way that it does not depend on the physical characteristics of devices. BMS provides a measure of device independence.

direct access storage. (1) * A storage device in which the access time is in effect independent of the location of the data. (2) A storage device that provides direct access to data.

DL/I. Data Language/I, the high-level interface between a user application and an IMS/VS database.

DTB. See dynamic transaction backout.

dynamic transaction backout. The process of canceling changes made to stored data by a transaction following the failure of that transaction for whatever reason.

+---+
¦ **E** ¦
+---+

EDF. Execution (command-level) diagnostic facility for testing command-level programs interactively at a terminal.

emergency restart. The CICS facility for use following a system failure. It restores the data files of all interrupted transactions to the condition they were in after the last complete transaction (that affected them) before the failure.

end user. In CICS, a person using a terminal to cause execution of a CICS transaction. Typically, a non-data-processing professional, for example, a reservation clerk.

exception. An abnormal condition such as an I/O error encountered in processing a data set or a file.

+---+
¦ **F** ¦
+---+

*** file**. A set of related records treated as a unit, for example, in stock control, a file could consist of a set of invoices. See data set.

file control table. A CICS table containing the characteristics of the files accessed by file control.

*** format**. The arrangement or layout of data on a data medium. In CICS, the data medium is usually a display screen.

format independence. The ability to send data to a device without having to be concerned with the format in which the data will be displayed. The same data may appear in different formats on different devices.

+---+
¦ **H** ¦
+---+

high-values. Hexadecimal FF.

+---+
¦ **I** ¦
+---+

*** I/O**. Input/Output.

IMS/ESA. Information Management System/Extended System Architecture.

inquiry. A request for information from storage; for example, a request for the number of available airline seats.

installation. (1) A particular computing system, in terms of the work it does and the people who manage it, operate it, apply it to problems, service it and use the work it produces. (2) The task of making a program ready to do useful work. This task includes generating a program, initializing it, and applying PTFs to it.

installed program definition. An application program that has been defined to the CICS system by the CEDA transaction, and that is valid for processing under CICS. These definitions also keep track of whether an application program is in main storage or not. Prior to CICS/ESA 3.3, this was an entry in the processing program table (PPT).

installed transaction definition. A transaction that has been defined to the CICS system by the CEDA transaction, and that may be processed by the system. Prior to CICS/ESA 3.3, this was an entry in the program control table (PCT).

interactive. Pertaining to an application in which each entry calls forth a response from a system or program, as in an inquiry system or an airline reservation system. An interactive system may also be conversational, implying a continuous dialogue between the user and the system.

ISAM. Indexed Sequential Access Method.

+---+
¦ **J** ¦
+---+

journal. A chronological record of the changes made to a set of data; the record may be used to reconstruct a previous version of the set.

journaling. Recording transactions against a data set in such a way that the data set can be reconstructed by applying transactions in the journal against a previous version of the data set.

+---+
¦ **K** ¦
+---+

keyword. (1) A symbol that identifies a parameter. (2) A part of a command operand that consists of a specific character string.

+---+
¦ **L** ¦
+---+

label. See paragraph name.

linkage editor. A computer program used to create one load module from one or more independently-translated object modules or load modules by resolving cross references among the modules.

logging. The recording (by CICS) of recovery information onto journal 01 (the system log).

low-values. Hexadecimal 00.

+---+
¦ **M** ¦
+---+

main storage. Program-addressable storage from which instructions and data can be loaded directly into registers for subsequent execution or processing. See also real storage, storage.

map. In CICS, a format established for a page or a portion of a page.

master terminal. In CICS, the terminal at which a designated operator is signed-on.

master terminal operator. Any CICS operator authorized to use the master terminal functions.

multitasking. * Pertaining to the concurrent execution of two or more tasks by a computer.

multithreading. Pertaining to the concurrent operation of more than one path of execution within a computer. In CICS, the use, by several transactions, of a single copy of an application program.

+---+
¦ **N** ¦
+---+

null. A character encoding of hexadecimal 00--LOW-VALUE in COBOL.

+---+
¦ **O** ¦
+---+

online. (1) * Pertaining to a user's ability to interact with a computer. (2) * Pertaining to a user's access to a computer via a terminal. The term "online" is also used to describe a user's access to a computer via a terminal.

operating system. Software that controls the execution of programs; an operating system may provide services such as resource allocation, scheduling, input/output control, and data management.

OS. Operating System.

+---+
¦ **P** ¦
+---+

paragraph name. COBOL term for destination of a branch or GOTO instruction.

*** parameter**. A variable that is given a constant value for a specified application and that may denote the application.

processing program table (obsolete). See installed program definition.

program control. The CICS element that manages CICS application programs.

program control table (obsolete). See installed transaction definition.

program function (PF) key. A key that passes a signal to a program.

pseudoconversational. A series of CICS transactions designed to appear to the operator as a continuous conversation occurring as part of a single transaction.

```
+---+
¦ Q ¦
+---+
```

quasi-reentrant. Applied to a CICS application program that is serially reusable between CICS calls because it does not modify itself or store data within itself between calls on CICS facilities.

```
+---+
¦ R ¦
+---+
```

real storage. The main storage in a virtual storage system. Physically, real storage and main storage are identical. Conceptually, however, real storage represents only part of the range of addresses available to the user of a virtual storage system.

recoverable resources. Items whose integrity CICS will maintain in the event of a system failure. They include individual CICS files, and auxiliary temporary storage queues.

reentrant. The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

*** response time**. The elapsed time between the end of an inquiry or demand on a data processing system and the beginning of the response. For example, the length of time between an indication of the end of an inquiry and the display of the first character of the response at a user terminal.

```
+---+
¦ S ¦
+---+
```

SAM. Sequential Access Method.

screen page. The amount of data displayed, or capable of being displayed, at any one time on the screen of a terminal.

SIT. System initialization table. A CICS table.

space. A character encoding of hexadecimal 40.

storage. A functional unit into which data can be placed and from which it can be retrieved. See main storage, real storage.

storage control. The CICS element that manages working storage areas.

system initialization table. A table containing user-specified data that will control a system initialization process.

system log. The (only) journal data set (identification='01') that is used by CICS to log changes made to resources for the purpose of backout on emergency restart.

```
+---+
¦ T ¦
+---+
```

task. (1) A basic unit of work to be accomplished by a computer. (2) Under CICS, the execution of a transaction for a particular user. Contrast with transaction.

task control. The CICS element that controls all CICS tasks.

task control area (TCA). A basic CICS control block provided for each task.

TCA. See task control area.

TCT. Terminal control table. A CICS table.

terminal. (1) * A point in a system or communication network at which data can either enter or leave. (2) In CICS, a device, often equipped with a keyboard and some kind of display, capable of sending and receiving information over a communication channel.

terminal control. The CICS element that controls all CICS terminal activity.

terminal control table. A table describing a configuration of terminals, logical units, or other CICS systems in a CICS network with which the CICS system may communicate.

terminal operator. The user of a terminal.

terminal paging. A set of CICS commands for retrieving "pages" of an oversize output message in any order.

threading. The process whereby various transactions undergo concurrent execution.

TIOA. Terminal input/output area.

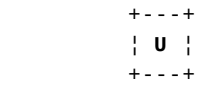
transaction. A transaction may be regarded as a unit of processing (consisting of one or more application programs) started by a single request, often from a terminal. A transaction may require the starting of one or more tasks for its execution. Contrast with task.

transaction backout. The cancellation, as a result of a transaction failure, of all updates performed by a partially-completed task.

transaction identification code. Synonym for transaction identifier. A group of up to four characters used to identify (name) a particular transaction type in the list of installed transaction definitions.

transaction identifier. Synonymous with transaction identification code.

transaction restart. The restart of a task after a transaction backout.



update. To modify a file with current information.

COMMENTS Readers' Comments
CICS
Application Programming Primer

Publication No. SC33-0674-01

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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.

Be sure to print your name and address below if you would like a reply.

International Business Machines Corporation
Attn: Dept ACV-H
1001 WT HARRIS BLVD
CHARLOTTE NC 28257-0001

Name . . . . . _____
Company or Organization _____
Address . . . . . _____
_____
_____
Phone No. . . . . _____