Introduction to CICS programming

Now that you understand the critical concepts and terms related to CICS, you’re ready to learn how to develop CICS programs. To get you started, this chapter presents a simple but complete CICS program, including the BMS mapset that defines the user interface. When you complete this chapter, you should have a good understanding of how a CICS program works and what you have to do to develop one.
How a CICS program works

To understand how a CICS program works, this topic starts by presenting a sample terminal session for a program that displays customer information. That will give you an idea of what the user sees as a CICS program executes. What the user doesn’t see, though, is that each time a new screen is displayed, the program actually ends. This is the basic idea of pseudo-conversational programming, and it’s the key to understanding how CICS programs work.

The operation of the customer inquiry program

Figure 2-1 presents four screens from the customer inquiry program. To start this program, the user can enter the trans-id INQ1 or select the program from a menu that specifies that trans-id. In either case, CICS searches the Program Control Table to determine what program is associated with that trans-id. Then, CICS starts that program and displays the first screen shown in this figure.

On the first line of this screen, you can see the name of the map that’s used to display this screen and the trans-id that’s used to start the program. In addition, you can see the name of the program, Customer Inquiry. This information helps the user identify the map and the program that displays it.

Below the trans-id and program name are instructions for the user. In this case, the instructions tell the user to type a customer number and press the Enter key. In the second screen shown in this figure, you can see that the user has entered a customer number. Then, when the user presses the Enter key, the third screen shown in part 2 of this figure is displayed.
Screen 1

When the program first starts, it displays the customer inquiry map.

Type a customer number. Then press Enter.

Customer number: ...

Name and address: ...

Screen 2

The user can then enter a customer number and press the Enter key.

Type a customer number. Then press Enter.

Customer number: 123456

Name and address: ...

Figure 2-1 The operation of the customer inquiry program (part 1 of 2)
As you can see in screen 3, the program has retrieved the record for the requested customer and displayed the data for that customer on the screen. At this point, the user can do one of two things. First, the user can enter another customer number and then press the Enter key to display the information for that customer. Second, the user can press the PF3 or PF12 key to exit from the program. Although both keys cause the program to end in this example, they may perform different functions in more complex programs.

In screen 4, you can see that the user entered another customer number and pressed the Enter key. This time, though, no record was found for that customer number. Because of that, the program has displayed an error message near the bottom of the screen. At this point, the user can either enter another customer number and press the Enter key or press PF3 or PF12 to end the program.

Incidentally, the screens in figure 2-1 show the mainframe output as it appears in a PC window when using a program called EXTRA! for Windows 98. As you learned in chapter 1, this is one of many third-party programs that make it possible to emulate a 3270 terminal on a PC.
Screen 3

If the customer number is valid, the program retrieves the customer record and displays the data for that customer on the screen.

Type a customer number. Then press Enter.
Customer number: 123456
Name and address:
RAUL
888 CICS WAY
FRESNO CA 93711-2765

F3=Exit   F12=Cancel
05/27

Screen 4

To display the data for another customer, the user can type over the current customer number and then press the Enter key. If the customer number is invalid, an error message is displayed.

Type a customer number. Then press Enter.
Customer number: 123457
Name and address:

That customer does not exist.

F3=Exit   F12=Cancel
05/27

Figure 2-1 The operation of the customer inquiry program (part 2 of 2)
How conversational programming works

In the screens in figure 2-1, the customer inquiry program appears to be sitting idle, waiting for the user to enter data. An online program that does sit idle while it waits for data is called a conversational program. As you can imagine, a conversational program spends almost all of its time doing nothing but waiting. On a single user system, this isn’t a problem because there’s nothing else for the computer to do. But in a system like CICS that has many users, it is a problem.

Figure 2-2 illustrates how a conversational program works. After it sends its initial map to the terminal, it waits for the user’s input. Then, it retrieves the input, processes it, and sends output back to the terminal. This continues until the user ends the program.

Although CICS is free to dispatch other programs for execution while the conversational program is waiting for input, the execution of the conversational program itself (the task) must remain in main storage. That includes the working storage for the program and all of the CICS control blocks that are required to keep track of the program’s execution.

Clearly, then, conversational programs are inefficient when it comes to the use of virtual storage…and virtual storage is one of the most critical CICS resources. In fact, one of the most common CICS problems is a condition known as Short On Storage, or, appropriately, SOS. When CICS goes short on storage, it suspends all work and begins terminating tasks in an effort to free up storage. As you can imagine, the result of an SOS condition can be disruptive as CICS grinds to a halt. And conversational programs are often a major cause of SOS conditions.
Conversational processing

Description

- A conversational program is an online program that sits idle while it waits for user input.
- When a conversational program first starts, it sends its initial map to the terminal. Then, it enters into a loop where it waits for input, retrieves the input data from the terminal, processes the data, sends its output to the terminal, and waits again.
- While a conversational program waits for user input, the task associated with that program remains in storage. Because of that, conversational programs are inefficient in terms of virtual storage usage.
- Conversational programs are often the cause of a CICS condition known as Short on Storage, or SOS.
How pseudo-conversational programming works

The solution to the inefficiencies of conversational programs is to remove the task associated with a program from storage while the program is waiting for terminal input. And that’s just what happens with a pseudo-conversational program. For example, while the map for the customer inquiry program in figure 2-1 is displayed, the inquiry program itself has terminated and its task is no longer in storage. When the user enters another customer number and presses the Enter key, the inquiry program is restarted. The result is that the task is in storage only when it needs to be: when the program is processing data. Because pseudo-conversational programs use main storage and other CICS resources far more efficiently than conversational programs, almost all CICS installations require that programs be pseudo-conversational.

Figure 2-3 illustrates how a pseudo-conversational program works. Like a conversational program, it starts by sending its initial map to the terminal. Unlike a conversational program, though, a pseudo-conversational program ends after it sends the map. Then, when the user presses one of the terminal’s attention identifier (AID) keys—the Enter key, a PF key, a PA key, or the Clear key—it signals CICS to restart the program. At that point, the task is loaded back into storage, the data is retrieved from the terminal and processed, output is sent back to the terminal, and the program ends again. This continues until the user indicates that the program should end.

Unfortunately, pseudo-conversational programming is more difficult than conversational programming because each program must be coded so it can figure out what processing it should do each time it’s started. This requires a different type of design and logic than you’ve seen before. You’ll see how this works in the customer inquiry program later in this chapter.
Pseudo-conversational processing

Description

- With *pseudo-conversational programming*, a program ends after it sends data to a terminal. This releases some of the resources that are used by the program. Then, CICS restarts the program when the user completes an entry and presses one of the terminal’s *attention identifier (AID) keys*.

- A pseudo-conversational program requires a different type of design and logic than a conversational program because it must be able to figure out what to do each time it’s restarted.
How to code CICS commands

When you code a CICS program, you request CICS services by issuing CICS commands. To do that, you use the syntax shown at the top of figure 2-4. As you can see, you start each command with EXEC CICS, and you end each command with END-EXEC. Within these statements, you code the command and any options it requires. In this figure, you can see a list of some of the most common CICS commands for screen interactions, for passing control from one program to another, and for processing VSAM files.

In the example in this figure, you can see how the RECEIVE MAP command is coded. In accordance with standard CICS command syntax, the words outside the parentheses (MAP, MAPSET, and INTO) are all options of the command, while the values inside the parentheses are supplied by the programmer. As a result, this command tells CICS to receive data from a terminal using a mapset named INQSET1 that defines a map named INQMAP1. The data in that mapset should be returned to a group item in working storage named INQMAP1I. As you’ll see in a moment, the MAP, MAPSET, and INTO values are all related to the BMS mapset that defines the maps for the program.
The general syntax of a CICS command

```
EXEC CICS
  command option(value)...
END-EXEC.
```

CICS commands for doing BMS-controlled screen interactions

- RECEIVE MAP: Retrieves input data from the terminal.
- SEND MAP: Sends information to the terminal for display.

CICS commands for passing control from one program to another

- ABEND: Transfers control to an abend routine or returns control to CICS.
- LINK: Invokes a program at a lower level.
- RETURN: Returns control to CICS or the invoking program.
- XCTL: Transfers control to another program.

CICS commands for processing VSAM files

- DELETE: Deletes a record from a file.
- READ: Retrieves a specified record from a file.
- REWRITE: Updates a record in a file.
- SYNCPOINT: Commits or reverses updates made to one or more files.
- WRITE: Adds a record to a file.

A CICS command that receives data from the terminal

```
EXEC CICS
  RECEIVE MAP('INQMAP1')
    MAPSET('INQSET1')
    INTO(INQMAP1I)
END-EXEC.
```

Description

- Within a COBOL program, each CICS command must begin with the words EXEC CICS and end with the words END-EXEC.
- The command specifies the operation to be performed, and the options, or parameters, provide information CICS needs to perform the operation.
- When a command requires more than one option, you use spaces to separate them.
- To make your programs easier to read, we recommend that you code each option of a CICS command on a separate line. No special coding is required to continue a command from one line to the next.
COBOL statements that aren't supported under CICS

Before I show you the code for the customer inquiry program, you should know about the COBOL statements that you can’t use in a CICS program. These statements are summarized in figure 2-5. For the most part, CICS provides services that perform the functions of these statements. The exception is the Merge statement, whose function is not provided by CICS.

Although you can’t code a Merge statement in a CICS program, you can code Sort, Release, and Return statements. Because of the restrictions for using these statements in a CICS program, however, you’ll rarely use them. Instead, if you need to retrieve records in a particular sequence, you’ll define that sequence through the primary key for the file or through an alternate index.
COBOL statements not allowed in CICS programs

**Operator communication statements**
- Accept
- Display

**File I/O statements**
- Open
- Close
- Read
- Write
- Rewrite
- Delete
- Start

**Other statements**
- Accept Date/Day/Day-Of-Week/Time
- Merge

**Description**
- Under CICS, all terminal I/O is handled by the terminal control module, whose services are typically requested by issuing SEND MAP and RECEIVE MAP commands. Because of that, COBOL Accept and Display statements aren’t allowed in CICS programs.
- Under CICS, all file I/O is handled by the file control module, whose services are requested by issuing CICS file control commands like READ, WRITE, REWRITE, and DELETE. Because of that, the COBOL statements for file I/O aren’t allowed. In addition, Environment Division and File Section statements that pertain to data management should not be coded.
- COBOL statements that invoke operating system functions, like Accept Date and Accept Time, are not allowed.
- Although the COBOL statements for sorting (Sort, Release, and Return) are allowed, their functionality is severely restricted. Because of that, you’ll probably never use them. Merge statements are never allowed.
The specifications and BMS mapset for an inquiry program

Before you can code a CICS program, you need to create a BMS mapset to define the screens that are going to be used by the program. This mapset includes all the fields that are going to be displayed on the screens as well as the fields that need to be entered by the user. Before you can create a mapset, though, you need to know what the program will do and what its screens will look like. Although you saw the general operation of the customer inquiry program earlier in this chapter, the topics that follow will present the complete specifications for that program, including the program overview and screen layout. Then, you’ll see the BMS mapset that’s based on those specifications.

The program overview

Figure 2-6 presents the program overview for the customer inquiry program that you saw illustrated in figure 2-1. This overview begins with a brief description of the program’s function. Then, it lists the I/O requirements for the program. Last, it presents a detailed list of processing specifications. These specifications state what the program is to do in response to various input actions. For example, if the user presses PF3 or PF12, the program is to return to the menu that invoked it.

The screen layout

Figure 2-6 also presents the screen layout for the one screen used by this program. This layout shows the positions of headings, captions, and data on the screen. For example, you can see that the caption “Customer number” begins in column 2 of line 5; that the user will enter the customer number in the entry field at column 27 on line 5; and that the program will display the customer’s name and address in the fields on lines 7, 8, 9, and 10.
The program overview for the customer inquiry program

<table>
<thead>
<tr>
<th>Program</th>
<th>CUSTINQ1: Customer inquiry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans-id</td>
<td>INQ1</td>
</tr>
<tr>
<td>Overview</td>
<td>Display customer information from the customer master file based on customer numbers entered by the user.</td>
</tr>
<tr>
<td>Input/output specifications</td>
<td>CUSTMAS Customer master file</td>
</tr>
<tr>
<td></td>
<td>INQMAP1 Customer inquiry map</td>
</tr>
</tbody>
</table>

Processing specifications
1. Control is transferred to this program via XCTL from the menu program INVMENU with no communication area. The user can also start the program by entering the trans-id INQ1. In either case, the program should respond by displaying the customer inquiry map.
2. If the user enters a customer number, read the customer record from CUSTMAS and display it. If the record doesn’t exist, display an error message.
3. If the user presses the Clear key, redisplay the customer inquiry map without any data.
4. If the user presses PF3 or PF12, return to the menu program INVMENU by issuing an XCTL command.

The screen layout for the customer inquiry program

<table>
<thead>
<tr>
<th>Map name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>INQMAP1</td>
<td>2/13/2001</td>
</tr>
<tr>
<td>Program name</td>
<td>CUSTINQ1</td>
</tr>
<tr>
<td>Designer</td>
<td>Doug Lowe</td>
</tr>
</tbody>
</table>

Figure 2-6 The specifications for the customer inquiry program
The BMS mapset

The two parts of figure 2-7 present the BMS mapset for the customer inquiry screen. This is a special type of assembler language program that defines the format of each map that’s used by a program. After you code the mapset, you run a program called an assembler that compiles (or assembles) the code into a physical mapset and a symbolic mapset. The physical mapset is used by CICS to determine the location, appearance, and operation of the data on the screen when the program that uses the mapset is run. The symbolic mapset is a copy member that can be copied into the COBOL program that uses the mapset. You use the fields in the symbolic mapset to work with the data on the screen.

When you code a mapset, you use three macros: DFHMSD to start the mapset; DFHMDI to start each map; and DFHMDF to define each field within a map. To end a mapset, you code another DFHMSD, as you can see on the second page of this mapset.

In this mapset, the first DFHMSD macro defines a mapset named INQSET1. Its parameters specify that the symbolic map that BMS generates from this mapset should be in COBOL (LANG=COBOL), that the symbolic map should include fields for both input and output (MODE=INOUT), and that the mapset should work with a standard 3270-type display station (TERM=3270-2). Although this macro includes other parameters, you don’t need to worry about them now. You’ll learn all the details in chapter 4.

Next, the DFHMDI macro defines a map named INQMAP1 that represents the one screen (map) that this program requires. This macro specifies that the size of the screen display is 24 lines of 80 characters each and that the map should be displayed starting in column 1 of line 1.

The other macros in this mapset except the last are the DFHMDF macros that define the fields on the screen. For example, the sixth DFHMDF macro defines a customer number field named CUSTNO. It says that this field should be positioned at column 26 in the fifth line. Actually, the POS parameter determines the location of the attribute byte for each field. This byte, which precedes each field, determines the characteristics of the field. As a result, the customer number field will actually start in column 27 of line 5.

The other parameters for a field give the length, attributes, color, and initial value for the field. Of these, the only entries that you may not understand are those for the ATTRB parameter. You’ll learn more about these attributes in the chapter 4. For now, just realize that the PROT attribute indicates that the field is protected so it can’t be changed, and UNPROT means that the field can be changed; NORM means that the field is displayed with normal intensity, and BRT means it’s displayed with high intensity; and IC means that the cursor will be positioned on the field when the screen is displayed.

In addition to the parameters that are specified for each field, you should notice that most of the DFHMDF macros include a label, such as CUSTNO or LNAME, that gives a name to the field being defined. These labels will be used to generate the fields that are included in the symbolic mapset that’s created from the BMS mapset. Since the only way to refer to a field in the BMS mapset...
The code for the BMS mapset

```
  PRINT NOGEN
  INQSET1 DFHMSD TYPE=&SYSPARM, X
    LANG=COBOL, X
    MODE=INOUT, X
    TERM=3270-2, X
    CTRL=FREEKB, X
    STORAGE=AUTO, X
    TIOAPFX=YES
  ***********************************************************************
  INQMAP1 DFHMDI SIZE=(24,80), X
    LINE=1, X
    COLUMN=1
  ***********************************************************************
  DFHMDF POS=(1,1), X
    LENGTH=7, X
    ATTRB=(NORM,PROT), X
    COLOR=BLUE, X
    INITIAL='INQMAP1'
  DFHMDF POS=(1,20), X
    LENGTH=16, X
    ATTRB=(NORM,PROT), X
    COLOR=GREEN, X
    INITIAL='Customer Inquiry'
  TRANID DFHMDF POS=(1,76), X
    LENGTH=4, X
    ATTRB=(NORM,PROT), X
    COLOR=BLUE, X
    INITIAL='XXXX'
  ***********************************************************************
  DFHMDF POS=(3,1), X
    LENGTH=42, X
    ATTRB=(NORM,PROT), X
    COLOR=NEUTRAL, X
    INITIAL='Type a customer number. Then press Enter.'
  DFHMDF POS=(5,1), X
    LENGTH=24, X
    ATTRB=(NORM,PROT), X
    COLOR=GREEN, X
    INITIAL='Customer number. . . . . '
  CUSTNO DFHMDF POS=(5,26), X
    LENGTH=6, X
    ATTRB=(NORM,UNPROT,IC), X
    COLOR=TURQUOISE, X
    INITIAL='___'
  DFHMDF POS=(5,33), X
    LENGTH=1, X
    ATTRB=ASKIP
  ***********************************************************************
```

Description

- A BMS mapset is an assembler language program that defines the format of the maps that are used by a program. After you code a mapset, you assemble it to produce a physical mapset and a symbolic mapset.
- The physical mapset is used by CICS to determine the location, appearance, and operation of data that’s displayed on a screen. The symbolic mapset is a COBOL copy member that allows you to manipulate the screen data in your COBOL program.
from a COBOL program is through the symbolic mapset, you need to include a label on the DFHMDF macro for any field you’ll need to refer to in the code.

With that as background, you should be able to understand how each field in the inquiry screen is defined. For instance, CUSTNO will be unprotected, it will be displayed in turquoise with normal intensity, and it will have an initial display of underscores; LNAME will be protected and will be displayed in turquoise with normal intensity; and MESSAGE will be protected and will be displayed in yellow with high intensity. All of these definitions follow IBM’s CUA (Common User Access) standards, which are designed to insure that programs interact with users in consistent ways. You’ll learn more about these standards in chapter 4.

The last macro in the mapset is a DFHMSD macro. On this macro, you code a single parameter to mark the end of the mapset: TYPE=FINAL.

In addition to the macros for defining the mapset, maps, and fields, PRINT NOGEN can be coded at the beginning of a mapset. This is a command that tells the assembler not to print the statements that are generated as a result of assembling the macros. And at the end of a mapset, you need to code END. This command tells the assembler that there are no more source statements.

To enter the code for a BMS mapset, you use an editor like the ISPF editor. Another alternative, though, is to use a program called a screen painter. This lets you design the screen layout at your terminal interactively. Then, the screen painter generates the assembler language code that you need for the mapset. Even though the code is generated for you, you still need to understand it because you may need to refer to it later on.
The code for the BMS mapset

```
DFHMDF POS=(7,1),
  LENGTH=24,
  ATTRB=(NORM, PROT),
  COLOR=GREEN,
  INITIAL='Name and address . . . :'
LNAME  DFHMDF POS=(7,26),
  LENGTH=30,
  ATTRB=(NORM, PROT),
  COLOR=TURQUOISE
FNAME  DFHMDF POS=(8,26),
  LENGTH=20,
  ATTRB=(NORM, PROT),
  COLOR=TURQUOISE
ADDR   DFHMDF POS=(9,26),
  LENGTH=30,
  ATTRB=(NORM, PROT),
  COLOR=TURQUOISE
CITY   DFHMDF POS=(10,26),
  LENGTH=20,
  ATTRB=(NORM, PROT),
  COLOR=TURQUOISE
STATE  DFHMDF POS=(10,47),
  LENGTH=2,
  ATTRB=(NORM, PROT),
  COLOR=TURQUOISE
ZIPCODE DFHMDF POS=(10,50),
  LENGTH=10,
  ATTRB=(NORM, PROT),
  COLOR=TURQUOISE

***********************************************************************
MESSAGE DFHMDF POS=(23,1),
  LENGTH=79,
  ATTRB=(BRT, PROT),
  COLOR=BLUE,
  INITIAL='F3=Exit   F12=Cancel'
DUMMY DFHMDF POS=(24,79),
  LENGTH=1,
  ATTRB=(DRK, PROT, FSET),
  INITIAL=' ' 

***********************************************************************
```

**Description**

- To code a BMS mapset, you use two assembler commands (PRINT NOGEN and END) and three macros (DFHMSD, DFHMDI, and DFHMDF).

- A DFHMSD macro marks the start and end of each mapset. A DFHMDI macro marks the beginning of each map in the mapset. And a DFHMDF macro defines each field in a mapset. Each field definition can include, among other things, the field’s starting position on the screen, its length, its display attributes, its color, and its initial value.

---

Figure 2-7 The BMS mapset for the customer inquiry program (part 2 of 2)
The symbolic map

Figure 2-8 presents the symbolic mapset that the assembler produces from the BMS mapset in figure 2-7. This is a copy member that can be copied into your COBOL program. Because this symbolic mapset contains just one map, you can also refer to it as a symbolic map. Similarly, if a symbolic mapset contains more than one map, you can refer to each map as a symbolic map.

If you look at both pages of the symbolic map for the inquiry program, you can see that the group item named INQMAP1I is redefined by a group item named INQMAP1O. The fields in the INQMAP1I group item are intended for use with input operations, and the fields in the INQMAP1O group item are intended for use with output operations. As a result, different pictures can be used for input and output. This is most useful when you work with numeric fields. Then, the output fields can be defined as numeric edited so they’re displayed properly, and the input fields can be defined as numeric or alphanumeric. Because the fields in the inquiry program are all alphanumeric, all of their pictures are the same for input and output.

For each screen field that’s named in the mapset, the symbolic map contains a data field, a length field, an attribute field, and a field that indicates if the value of the field has changed. The names for these fields are created by adding a one-character suffix to the label that was coded on the DFHMDF macro in the mapset. So the name of the input field for the field labeled CUSTNO is CUSTNOI, the name of the length field is CUSTNOL, the name of the attribute field is CUSTNOA, and the name of the field that indicates a change is CUSTNOF.
### The code for the symbolic map

```cobol
01 INQMAP1I.
   03 FILLER PIC X(12).
   03 TRANIDL PIC S9(4) COMP.
   03 TRANIDF PIC X.
   03 FILLER REDEFINES TRANIDF.
      05 TRANIDA PIC X.
   03 TRANIDI PIC X(4).
   03 CUSTNOL PIC S9(4) COMP.
   03 CUSTNOF PIC X.
   03 FILLER REDEFINES CUSTNOF.
      05 CUSTNOA PIC X.
   03 CUSTNOI PIC X(6).
   03 LNAMEL PIC S9(4) COMP.
   03 LNAMEF PIC X.
   03 FILLER REDEFINES LNAMEF.
      05 LNAMEA PIC X.
   03 LNAMEI PIC X(30).
   03 FNAMEL PIC S9(4) COMP.
   03 FNAMEF PIC X.
   03 FILLER REDEFINES FNAMEF.
      05 FNAMEA PIC X.
   03 FNAMEI PIC X(20).
   03 ADDRDL PIC S9(4) COMP.
   03 ADDRF PIC X.
   03 FILLER REDEFINES ADDRF.
      05 ADDRA PIC X.
   03 ADDRI PIC X(30).
   03 CITYL PIC S9(4) COMP.
   03 CITYF PIC X.
   03 FILLER REDEFINES CITYF.
      05 CITYA PIC X.
   03 CITYI PIC X(20).
   03 STATEL PIC S9(4) COMP.
   03 STATEF PIC X.
   03 FILLER REDEFINES STATEF.
      05 STATEA PIC X.
   03 STATEI PIC X(2).
```

### Description

- A **symbolic map** is a COBOL copy member that’s created when you assemble a mapset. The fields in a symbolic map represent the data that’s sent to and received from a terminal by a COBOL program.

- You use a Copy statement to copy the symbolic map into the Working-Storage Section of your program.

- A symbolic map includes two 01-level items: one for input and one for output. Because the second item contains a Redefines clause (see the next page), it occupies the same storage space as the first item. That way, different Picture clauses can be used for input to a field and output from it.

- For each input field in the mapset (I), the symbolic map contains a field that indicates the length of the data in the field (L), a field that indicates if the user made changes to the field (F), and a field that contains the display attributes (A).
In the redefined area for the output fields, you’ll notice that only the data fields are named, like CUSTNOO and LNAMEO. The length, attribute, and change indicator fields are coded as FILLER. To change the value of any of these fields for an output operation, then, you have to refer to the fields in the input area.

As you work with a symbolic map, you don’t need to be concerned with the details of how the fields it contains are defined in the BMS mapset. You just need to know the names of the fields that you’re going to use in your COBOL programs. The fields you’ll use the most are the length fields (with data names ending in $L$) and the fields that contain the data (with data names ending in $I$ for input fields and $O$ for output fields). Next, you’ll see how these fields are used by the customer inquiry program.
The code for the symbolic map

```assembly
03 ZIPCODEL     PIC S9(4) COMP.
03 ZIPCODEF     PIC X.
03 FILLER REDEFINES ZIPCODEF.
   05 ZIPCODEA     PIC X.
03 ZIPCODEI     PIC X(10).
03 MESSAGEL     PIC S9(4) COMP.
03 MESSAGEF     PIC X.
03 FILLER REDEFINES MESSAGEF.
   05 MESSAGEA     PIC X.
03 MESSAGEI     PIC X(79).
03 DUMMYL       PIC S9(4) COMP.
03 DUMMYF       PIC X.
03 FILLER REDEFINES DUMMYF.
   05 DUMMYA       PIC X.
03 DUMMYI       PIC X(1).
01 INQMAP1O REDEFINES INQMAP1I.
   03 FILLER       PIC X(12).
   03 FILLER       PIC X(3).
   03 TRANIDO      PIC X(4).
   03 FILLER       PIC X(3).
   03 CUSTNOO      PIC X(6).
   03 FILLER       PIC X(3).
   03 LNAMEO       PIC X(30).
   03 FILLER       PIC X(3).
   03 FNAMENO      PIC X(20).
   03 FILLER       PIC X(3).
   03 ADDRNO       PIC X(30).
   03 FILLER       PIC X(3).
   03 CITYO        PIC X(20).
   03 FILLER       PIC X(3).
   03 STATEO       PIC X(2).
   03 FILLER       PIC X(3).
   03 ZIPCODEO     PIC X(10).
   03 FILLER       PIC X(3).
   03 MESSAGEO     PIC X(79).
   03 FILLER       PIC X(3).
   03 DUMMYO       PIC X(1).
```

Description

- The definitions of the input fields are redefined by the definitions of the output fields. For each output field (O), the symbolic map contains a Picture for the output data.
- Because all of the fields in the output area except for the output field are defined as Filler, you can’t refer to these fields from the output area. Instead, you have to use the field names in the input area.
The design and COBOL code for the inquiry program

Before you code a CICS program, you should take the time to develop an event/response chart and a structure chart for the program. These design tools let you create a program that’s easier to code, test, and debug. The charts for the inquiry program are presented next, followed by the COBOL code for the program.

The event/response chart

A pseudo-conversational CICS program must be written so it responds appropriately to each type of user action that can occur. Because this is so important, it’s worth taking the time to summarize all of the possible actions (events) and the appropriate responses before you start coding. The best way to do that is to prepare an event/response chart like the one in figure 2-9.

As you can see, this chart goes beyond what’s in the program overview to clarify the processing that must be done to retrieve and display a customer record. As a result, it provides for several input actions that weren’t included in the overview. When you develop a chart like this, you do your best to include every possible user action so your program can provide for it.

The structure chart

To plan the overall structure of a program, we recommend the techniques of top-down design. The basic idea of top-down design is to design a program by dividing it into its major functional modules, then dividing those modules into their functional components, and so forth until each module can be coded in a single COBOL paragraph. To develop and document this design, you can use a structure chart like the one in figure 2-9.

Once the design is complete, the structure chart becomes your guide to coding the Procedure Division of the CICS program. There, each module is implemented as a single COBOL paragraph, and the paragraph name consists of the module number followed by the module name. For instance, module number 1000 in the chart in this figure will be implemented by a COBOL paragraph named 1000-PROCESS-CUSTOMER-MAP.

In chapter 3, you’ll learn the details of how to design a CICS program with a structure chart because we think that’s a critical step in the development process. You’ll also learn more about creating event/response charts because you can’t design a CICS program without first knowing what events the program has to respond to.
An event/response chart for the customer inquiry program

<table>
<thead>
<tr>
<th>Event</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start the program</td>
<td>Display the customer map.</td>
</tr>
<tr>
<td>Enter key</td>
<td>Receive the customer map.</td>
</tr>
<tr>
<td></td>
<td>Edit the customer number.</td>
</tr>
<tr>
<td></td>
<td>If valid</td>
</tr>
<tr>
<td></td>
<td>read the record.</td>
</tr>
<tr>
<td></td>
<td>If the record exists</td>
</tr>
<tr>
<td></td>
<td>display it.</td>
</tr>
<tr>
<td></td>
<td>If the number isn’t valid or the record doesn’t exist</td>
</tr>
<tr>
<td></td>
<td>display an error message.</td>
</tr>
<tr>
<td>PF3 or PF12</td>
<td>Return to the menu program.</td>
</tr>
<tr>
<td>Clear key</td>
<td>Redisplay the customer map without any data.</td>
</tr>
<tr>
<td>Any PA key</td>
<td>Ignore the key.</td>
</tr>
<tr>
<td>Any other key</td>
<td>Display an error message.</td>
</tr>
</tbody>
</table>

The structure chart for the customer inquiry program

Description

- A pseudo-conversational program must be designed to respond appropriately to each type of user action that might occur.
- The best way to identify the user actions and responses is to create an event/response chart. This chart summarizes each user action (event) and the program’s response to that event.
- The program structure chart specifies the overall program structure. For a pseudo-conversational program, the top-level module must determine what processing is to be done each time it’s executed. Those processing options are identified by the modules in the second level of the chart.

Figure 2-9 The design for the customer inquiry program
The Execute Interface Block

When you prepare a CICS program for execution, CICS inserts code that’s needed for the program to operate properly under CICS. One of these blocks of code is called the Execute Interface Block, or EIB. The EIB is inserted into the Linkage Section of a program, and its fields provide information about the current task.

In figure 2-10, you can see the start of this block of fields. Note that this block doesn’t appear in the source listing of your COBOL program. As a result, you have to get the names of the fields that you want to use from other documentation.

The two EIB fields you’ll use most often are EIBAID and EIBCALEN. The EIBCALEN field contains the length of the data that is passed to the program through its communication area. A program can use this area to store the data that it will need the next time it’s executed. That’s necessary because the contents of the working-storage fields are lost when a pseudo-conversational program ends. If the length of this area is zero, it means that no data was passed to the program. In other words, the program is being executed for the first time in a pseudo-conversational session.

In contrast, the EIBAID field gives the value of the last AID key that was pressed. You can use it to determine the processing the user has requested.

The DFHAID copy member

To make it easy to write the code that tests the values in the EIBAID field, IBM supplies a copy member named DFHAID. The first part of this member is shown in figure 2-10. It gives a name to each of the values that represents an AID key. For instance, DFHENTER is the name for the value that represents the Enter key, and DFHPF3 is the name for the value that represents the PF3 key. As a result, you can code

\[\text{IF EIBAID} = \text{DFHPF3}\]

when you want to see if the AID key that was pressed was the PF3 key, instead of coding

\[\text{IF EIBAID} = '3'\]

Once you get used to the names in this copy member, you’ll have no trouble using them.
Two of the fields in the Execute Interface Block

```assembly
01 DFHEIBLK.
  .
  02 EIBCALEN PIC S9(4) COMP.
  02 EIBAID PIC X(1).
  .

The DFHAID copy member

```assembly
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 DFHOPID PIC X VALUE IS 'W'.
  02 DFHMSRE PIC X VALUE IS 'X'.
  02 DFHSTRF PIC X VALUE IS 'h'.
  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 DFHPF3 PIC X VALUE IS '3'.
  02 DFHPF4 PIC X VALUE IS '4'.
  02 DFHPF5 PIC X VALUE IS '5'.
  02 DFHPF6 PIC X VALUE IS '6'.
  02 DFHPF7 PIC X VALUE IS '7'.
  02 DFHPF8 PIC X VALUE IS '8'.
  02 DFHPF9 PIC X VALUE IS '9'.
  02 DFHPF10 PIC X VALUE IS ':'.
  02 DFHPF11 PIC X VALUE IS '#'.
  02 DFHPF12 PIC X VALUE IS '@'.
  .
```

Description

- The **Execute Interface Block (EIB)** is a CICS area that contains information related to the current task, such as the date and time the task was started and the transaction-id that was used to start it. The definition of this area is inserted into the Linkage Section of the program when the program is prepared for execution.

- The EIBCALEN field contains the length of the data passed to the program through its **communication area** (DFHCOMMAREA). A length of zero indicates that no data was passed to the program, which means that it’s the first execution of the program.

- When the user presses an AID key, CICS passes a one-byte value to the program through the EIBAID field in the Execute Interface Block. You can use the value of this field to determine the processing the user has requested.

- The DFHAID copy member contains literal values that correspond to the AID keys that the user can press to communicate with the system.
The COBOL code

Figure 2-11 presents the source code for the customer inquiry program. The first thing you should notice is that the Environment Division doesn’t include any entries, and the Data Division doesn’t include a File Section. That’s because the customer master file is defined in the CICS File Control Table (FCT). Because the FCT keeps track of the characteristics of the file, you don’t have to code Select or FD statements for it.

In the Working-Storage Section, you can see the one switch and the one flag used by this program. The three conditions that are defined for the flag will be used to determine which options are coded in the SEND MAP command when the customer map is sent to the screen.

The next field is for data related to the communication area. As you’ll recall, this area can be used to store the data that’s passed to and from the program. To use the communication area, you need to provide two definitions for it in your program: one in the Working-Storage Section and one in the Linkage Section. The working-storage definition in this program is named COMMUNICATION-AREA, and the Linkage Section definition is named DFHCOMMAREA. Although you can use any name for the working-storage field, you must use the name DFHCOMMAREA for the Linkage Section field.

When a CICS program starts, the data in the CICS communication area is available through the DFHCOMMAREA field. The program can then use this information to determine the processing to be done. Then, when the program ends, it can specify that the data in the working-storage communication area be stored in the CICS communication area so it’s available for the next program execution. In the inquiry program, the communication area is a one-byte field, but a more complicated program may require many fields.

The next working-storage entry is RESPONSE-CODE. This field is used to test the completion status of the CICS READ command that retrieves records from the customer master file.

After the response-code field, you can see the record description for the customer master records that are read by this program. Normally, a record description like this is copied into a CICS program, but we’ve included it here to make the program easier to follow.

This description is followed by two Copy statements. The first one copies the symbolic map for the mapset named INQSET1, while the second one copies the DFHAID member that is supplied by IBM.

Before I go on, I want to point out that each time a program is executed, a fresh copy of working storage is obtained. As a result, changes you make to the contents of working-storage fields aren’t saved between executions of a pseudo-conversational program, and any initial values established by Value clauses are restored. If you need to preserve data from one program execution to the next, you can store it in the communication area.
The customer inquiry program

IDENTIFICATION DIVISION.
*
PROGRAM-ID. CUSTINQ1.
*
ENVIRONMENT DIVISION.
*
DATA DIVISION.
*
WORKING-STORAGE SECTION.
*
01 SWITCHES.
*
  05 VALID-DATA-SW           PIC X VALUE 'Y'.
  88 VALID-DATA                VALUE 'Y'.
*
01 FLAGS.
*
  05 SEND-FLAG            PIC X.
  88 SEND-ERASE             VALUE '1'.
  88 SEND-DATAONLY           VALUE '2'.
  88 SEND-DATAONLY-ALARM     VALUE '3'.
*
01 COMMUNICATION-AREA     PIC X.
*
01 RESPONSE-CODE          PIC S9(8) COMP.
*
01 CUSTOMER-MASTER-RECORD.
*
  05 CM-CUSTOMER-NUMBER     PIC X(6).
  05 CM-FIRST-NAME          PIC X(20).
  05 CM-LAST-NAME           PIC X(20).
  05 CM-ADDRESS             PIC X(30).
  05 CM-CITY                PIC X(20).
  05 CM-STATE               PIC X(2).
  05 CM-ZIP-CODE            PIC X(10).
*
COPY INQSET1.
*
COPY DFHAID.
*
LINKAGE SECTION.
*
01 DFHCOMMAREA            PIC X.
*
Page 2 of the COBOL listing presents the top-level procedure for this program. It contains the logic required to implement the pseudo-conversational design. This logic is coded as an Evaluate statement that specifies the actions that are required for the conditions the program may encounter when it’s started.

The first When clause in this statement tests the value of EIBCALEN, which is the EIB field that contains the length of the data passed to the program through its communication area. If the length is zero, it means that no data was passed to the program, which indicates that there was no previous execution of the program. As a result, procedure 1400 is performed to send the customer map to the screen. But first, the program initializes the map by moving Low-Value to the output area, it sets the TRANIDO field in the symbolic map to INQ1 so the correct trans-id is displayed, and it sets Send-Erase to True so procedure 1400 will know to clear the screen before displaying the map.

The next four When clauses test for values in the EIBAID field, which gives the value of the AID key that the user pressed to start this execution of the program. Each of these conditions uses a data name from the DFHAID copy member. The program checks the EIBAID field so it can avoid retrieving data from the terminal if the function it’s about to perform doesn’t call for it. That reduces network use and improves overall system performance.

If the user pressed the Clear key (DFHCLEAR), the program initializes the map, sets the TRANIDO field to INQ1, sets Send-Erase to True, and performs procedure 1400 to restart with a fresh screen. If the user pressed one of the program attention (PA) keys (DFHPA1 OR DFHPA2 OR DFHPA3), no special action is taken. Then, the Continue statement causes program execution to continue with the first statement after the Evaluate statement.

If the user pressed the PF3 or PF12 key (DFHPF3 OR DFHPF12), the program issues a CICS XCTL command to end the program and transfer control to the program named INVMENU. As you’ll learn in chapter 10, this program displays a simple menu that you can use to start the inquiry program as well as other programs. Finally, if the user pressed the Enter key (DFHENTER), the program performs procedure 1000 to receive and process the customer map.

If none of these conditions are true, the When Other clause performs procedure 1400 to display an error message that indicates that an invalid key was pressed. This time, Send-Dataonly-Alarm is set to True so procedure 1400 will use the form of the SEND MAP command that sounds the alarm.

Unless the XCTL command was executed in response to PF3 or PF12, the program continues with the RETURN command that follows the Evaluate statement. This command causes CICS to invoke the same trans-id (INQ1) the next time the user presses one of the AID keys. It also says that the data in COMMUNICATION-AREA should be passed to the next execution of the program through the CICS communication area. Be aware that you can put a value in this field if it’s required by the program specifications. In this case, though, that’s not necessary because the program never checks the value of this field, only its length.
The customer inquiry program

PROCEDURE DIVISION.
*
0000-PROCESS-CUSTOMER-INQUIRY.
*
   EVALUATE TRUE
*
   WHEN EIBCALEN = ZERO
      MOVE LOW-VALUE TO INQMAP10
      MOVE 'INQ1' TO TRANIDO
      SET SEND-ERASE TO TRUE
      PERFORM 1400-SEND-CUSTOMER-MAP
*
   WHEN EIBAID = DFHCLEAR
      MOVE LOW-VALUE TO INQMAP10
      MOVE 'INQ1' TO TRANIDO
      SET SEND-ERASE TO TRUE
      PERFORM 1400-SEND-CUSTOMER-MAP
*
   WHEN EIBAID = DFHPA1 OR DFHPA2 OR DFHPA3
      CONTINUE
*
   WHEN EIBAID = DFHPF3 OR DFHPF12
      EXEC CICS
         XCTL PROGRAM('INVMENU')
      END-EXEC
*
   WHEN EIBAID = DFHENTER
      PERFORM 1000-PROCESS-CUSTOMER-MAP
*
   WHEN OTHER
      MOVE LOW-VALUE TO INQMAP10
      MOVE 'Invalid key pressed.' TO MESSAGEO
      SET SEND-DATAONLY-ALARM TO TRUE
      PERFORM 1400-SEND-CUSTOMER-MAP
*
   END-EVALUATE.
*
   EXEC CICS
      RETURN TRANSID('INQ1')
      COMMAREA(COMMUNICATION-AREA)
   END-EXEC.
*
If the user pressed the Enter key, procedure 1000 is performed. This procedure performs procedure 1100 to get the customer number the user entered on the screen and procedure 1200 to edit the number. If the number is valid, procedure 1000 performs procedure 1300 to retrieve the requested record from the customer master file. Then, if the record is retrieved (indicating that the customer number is still valid), the program displays the customer data by performing procedure 1400 with Send-Dataonly set to True. Otherwise, procedure 1000 performs procedure 1400 with Send-Dataonly-Alarm set to True to alert the user that the customer number isn’t valid.

To get the user entry, procedure 1100 contains a single CICS RECEIVE MAP command. This command receives data from the terminal using the INQMAP1 map in the INQSET1 mapset. That data is then stored in the input area of the symbolic map (INQMAP1I).

The only editing requirement for this program is that the user must enter a customer number. Procedure 1200 does this editing by checking the length and input fields of the customer number in the symbolic map. If the length is zero, it means that the user didn’t enter anything or pressed the Erase-EOF key. In either case, no data is transmitted back to CICS so the CUSTNOI field will contain low-values. Since low-values and spaces aren’t the same, this routine also checks to see whether the input field contains spaces. If either of these error conditions is true, procedure 1200 turns the valid-data switch off so procedure 1000 can determine what processing it should do next. Procedure 1200 also moves an error message to the output MESSAGE field (MESSAGEO) in the symbolic map.

If procedure 1200 didn’t turn the valid-data switch off, procedure 1000 performs procedure 1300 to retrieve the requested record from the customer file. This procedure starts by issuing a CICS READ command that tells CICS to read a record from the file named CUSTMAS into the working-storage field named CUSTOMER-MASTER-RECORD. The RIDFLD option specifies that the key value for the record is in CUSTNOI, which is the field in the symbolic map where the customer number entered by the user was stored by the RECEIVE MAP command. And the RESP option specifies that the response code that indicates whether the operation was successful should be placed in the working-storage field named RESPONSE-CODE.

The code that follows the READ command tests the value of the response-code field. To do that, it uses the special keyword DFHRESP followed by the condition to be tested. Here, the program tests for a NORMAL response and the not-found condition (NOTFND). If the command completed normally, the program moves data from the customer record to the corresponding fields in the output map. But if the NOTFND condition occurred, the program sets the valid-data switch to N, moves an appropriate error message to MESSAGEO, and clears the output fields in the symbolic map as shown on the next page of this listing.
The customer inquiry program

1000-PROCESS-CUSTOMER-MAP.
  *
  PERFORM 1100-RECEIVE-CUSTOMER-MAP.
  PERFORM 1200-EDIT-CUSTOMER-DATA.
  IF VALID-DATA
    PERFORM 1300-GET-CUSTOMER-RECORD
  END-IF.
  IF VALID-DATA
    SET SEND-DATAONLY TO TRUE
    PERFORM 1400-SEND-CUSTOMER-MAP
  ELSE
    SET SEND-DATAONLY-ALARM TO TRUE
    PERFORM 1400-SEND-CUSTOMER-MAP
  END-IF.
  *
  1100-RECEIVE-CUSTOMER-MAP.
  *
  EXEC CICS
    RECEIVE MAP('INQMAP1')
      MAPSET('INQSET1')
      INTO(INQMAP1I)
  END-EXEC.
  *
  1200-EDIT-CUSTOMER-DATA.
  *
  IF CUSTNOL = ZERO
    OR CUSTNOI = SPACE
    MOVE 'N' TO VALID-DATA-SW
    MOVE 'You must enter a customer number.' TO MESSAGEO
  END-IF.
  *
  1300-GET-CUSTOMER-RECORD.
  *
  EXEC CICS
    READ FILE('CUSTMAS')
      INTO(CUSTOMER-MASTER-RECORD)
      RIDFLD(CUSTNOI)
      RESP(RESPONSE-CODE)
  END-EXEC.
  *
  EVALUATE RESPONSE-CODE
    WHEN DFHRESP(NORMAL)
      MOVE SPACE         TO MESSAGEO
      MOVE CM-LAST-NAME  TO LNAMEO
      MOVE CM-FIRST-NAME TO FNAMEO
      MOVE CM-ADDRESS    TO ADDRO
      MOVE CM-CITY       TO CITYO
      MOVE CM-STATE      TO STATEO
      MOVE CM-ZIP-CODE   TO ZIPCODEO

Figure 2-11  The COBOL code for the customer inquiry program (part 3 of 4)
If RESPONSE-CODE indicates any other condition, it means that a serious error occurred when the READ command was executed. In that case, the program handles the error by issuing a CICS ABEND command to terminate the program abnormally.

Unless procedure 1300 terminates the program, procedure 1000 continues by performing procedure 1400 to issue the appropriate SEND MAP command. In this procedure, one of three SEND MAP commands is issued depending on the setting of the Send-Flag field. Notice that all three commands specify the same mapset, map, and source of the data to be sent to the screen (the output area of the symbolic map). It’s the other parameters that vary.

The first SEND MAP command is issued if the Send-Erase setting is turned on. That happens when the program is started for the first time or when the user presses the Clear key. This command includes the ERASE option, which causes the screen to be erased before the map is displayed. As a result, the screen will look like the first one in figure 2-1.

The second SEND MAP command is issued if the Send-Dataonly setting is turned on. That happens when the user enters a valid customer number. Then, the SEND MAP command includes the DATAONLY option. That means that only the data in the symbolic map is sent to the terminal, because the literals that make up the headings, captions, and instructions are already there from the previous execution of the program. This improves the performance of the program.

The third SEND MAP command is issued if the Send-Dataonly-Alarm setting is turned on. That happens when the user presses an invalid key or enters an invalid customer number. Then, the SEND MAP command includes the DATAONLY and ALARM options. The ALARM option causes an audio beep at the terminal to call the user’s attention to the error.
The customer inquiry program

WHEN DFHRESP (NOTFND)
  MOVE 'N' TO VALID-DATA-SW
  MOVE 'That customer does not exist.' TO MESSAGEO
  MOVE SPACE TO LNAMEO
  FNAMEO
  ADDRO
  CITYO
  STATEO
  ZIPCODEO

WHEN OTHER
  EXEC CICS
  ABEND
END-EXEC
END-EVALUATE.

* 1400-SEND-CUSTOMER-MAP.
* 
EVALUATE TRUE
  WHEN SEND-ERASE
    EXEC CICS
    SEND MAP('INQMAP1')
    MAPSET('INQSET1')
    FROM(INQMAP1O)
    ERASE
  END-EXEC
  WHEN SEND-DATAONLY
    EXEC CICS
    SEND MAP('INQMAP1')
    MAPSET('INQSET1')
    FROM(INQMAP1O)
    DATAONLY
  END-EXEC
  WHEN SEND-DATAONLY-ALARM
    EXEC CICS
    SEND MAP('INQMAP1')
    MAPSET('INQSET1')
    FROM(INQMAP1O)
    DATAONLY
    ALARM
  END-EXEC
END-EVALUATE.
The CICS commands used in the program

If you look back over the COBOL code in figure 2-11, you can see that the program used just six CICS commands. To make sure that you have a good grasp of what these commands do and how you should code them, the next three figures summarize them. To get you started, these figures show only the basic options for each command, but you’ll learn how to use other options in chapter 5.

The SEND MAP and RECEIVE MAP commands

Figure 2-12 summarizes the two commands that work with maps. As you have seen in the inquiry program, the SEND MAP command sends the data from a map within a mapset to a terminal. As a result, you need to identify the mapset and map on the command. You also need to use the FROM option to specify the name of the data area in the symbolic map that contains the data to be sent to the terminal.

If you code just those options, the data in the symbolic map is combined with the data in the physical map, and both are sent to the screen. To limit the amount of data that’s sent, however, you can code the MAPONLY or the DATAONLY option. In the first case, only the data in the physical map is sent, so you don’t have to code the FROM option. In the second case, only the data in the symbolic map is sent.

Usually, you’ll omit the MAPONLY and DATAONLY options the first time a map is displayed. You’ll also code the ERASE option on this command to erase the previous contents of the screen before the new screen is displayed. Then, you’ll code the DATAONLY option on subsequent SEND MAP commands so the headings aren’t sent again, and you’ll omit the ERASE option so the current headings aren’t erased.

When you code the RECEIVE MAP command, you also code the mapset and map name. But this time, you use the INTO option to provide the name of the data area in the symbolic map that will receive the data.
The syntax of the SEND MAP command

```
EXEC CICS
   SEND MAP(map-name)
   [MAPSET(mapset-name)]
   [FROM(data-name)]
   [MAPONLY | DATAONLY]
   [ERASE]
   [ALARM]
END-EXEC
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP</td>
<td>Specifies the one- to seven-character name of the map for the output data.</td>
</tr>
<tr>
<td>MAPSET</td>
<td>Specifies the one- to seven-character name of the mapset that contains the map.</td>
</tr>
<tr>
<td>FROM</td>
<td>Specifies the name of the area in the symbolic map that contains the data to be mapped.</td>
</tr>
<tr>
<td>MAPONLY/</td>
<td>MAPONLY sends only constant data from the physical map, so no FROM area is used.</td>
</tr>
<tr>
<td>DATAONLY</td>
<td>DATAONLY sends only data from the FROM area, so no constant data from the physical map is sent. If neither option is specified, both the constant data in the physical map and the data in the FROM area are sent.</td>
</tr>
<tr>
<td>ERASE</td>
<td>Causes the contents of the screen to be erased before data is displayed.</td>
</tr>
<tr>
<td>ALARM</td>
<td>Causes the alarm to sound when the map is displayed.</td>
</tr>
</tbody>
</table>

A SEND MAP command that sends only the data from the symbolic map

```
EXEC CICS
   SEND MAP('INQMAP1')
   MAPSET('INQSET1')
   FROM(INQMAP1O)
   DATAONLY
END-EXEC.
```

The syntax of the RECEIVE MAP command

```
EXEC CICS
   RECEIVE MAP(map-name)
   [MAPSET(mapset-name)]
   INTO(data-name)
END-EXEC
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP</td>
<td>Specifies the one- to seven-character name of the map for the input data.</td>
</tr>
<tr>
<td>MAPSET</td>
<td>Specifies the one- to seven-character name of the mapset that contains the map.</td>
</tr>
<tr>
<td>INTO</td>
<td>Specifies the name of the data area where the mapped data should be placed.</td>
</tr>
</tbody>
</table>

A typical RECEIVE MAP command

```
EXEC CICS
   RECEIVE MAP('INQMAP1')
   MAPSET('INQSET1')
   INTO(INQMAP1I)
END-EXEC.
```

Figure 2-12 The basic formats of the SEND MAP and RECEIVE MAP commands
The READ command

Figure 2-13 summarizes the use of the READ command. This command can be used to read data from all three types of VSAM files: key-sequenced (indexed), entry-sequenced (sequential), and relative record files. Most of the time, though, this command is used to read key-sequenced files on a random basis. So that’s what this summary emphasizes.

If you’re familiar with the COBOL commands for working with indexed files, you shouldn’t have any trouble using the CICS READ command. The three options that it requires provide the name of the file, the name of the data area where the data should be placed when a record is read, and the name of the field that contains the key of the record to be read.

To determine whether the read operation is successful, you can also code the RESP option with this command. Then, after a read operation is attempted, CICS puts a response code in the field that’s named in the RESP option. You can then test this code to see whether the read operation was successful or whether an exceptional condition occurred. If, for example, the file doesn’t contain a record with the key that the command has specified, a “not found” condition occurs.

As you have already seen, you can use the DFHRESP keyword to check the response code. To test for the not-found condition, for example, you code NOTFND in the parentheses after DFHRESP. And to test for a successful read operation, you code NORMAL in the parentheses. In chapter 5, you’ll learn all of the common codes for exceptional I/O conditions.

As you might guess, the RESP option is used most often in file I/O commands, which is why we’ve only included it in the syntax of the READ command in this chapter. Be aware, though, that you can code the RESP option on any CICS command. The response codes that CICS returns will vary according to the function of the command. So throughout this book, you’ll learn about new response codes whenever they apply to the functions presented in a chapter.
The syntax of the READ command

```
EXEC CICS
  READ FILE(filename)
      INTO(data-name)
      RIDFLD(data-name)
      [RESP(response-code)]
END-EXEC
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE</td>
<td>Specifies the name of the file that contains the record to be read.</td>
</tr>
<tr>
<td>INTO</td>
<td>Specifies the name of the data area where the input record is placed.</td>
</tr>
<tr>
<td>RIDFLD</td>
<td>For a key-sequenced file, specifies the name of the field that contains the key of the record to be read.</td>
</tr>
<tr>
<td>RESP</td>
<td>Specifies the name of the field where CICS stores the response code from the read operation.</td>
</tr>
</tbody>
</table>

A READ command that reads a record from a key-sequenced file

```
EXEC CICS
  READ FILE('CUSTMAS')
      INTO(CUSTOMER–MASTER–RECORD)
      RIDFLD(CUSTNOI)
      RESP(RESPONSE–CODE)
END-EXEC.
```

An If statement that tests the response code after a read operation

```
IF RESPONSE–CODE = DFHRESP(NORMAL)
  .
  .
```

Description

- The READ command retrieves a record from a VSAM file. Although this command is usually used with key-sequenced (indexed) files, it can also be used with entry-sequenced (sequential) and relative record files.
- The RESP option can be used with any CICS command. The *response code* it returns indicates whether the operation was successful or whether an *exceptional condition* occurred.
- To test the response code that CICS places in the response-code field, you can use the DFHRESP keyword as shown above. Within the parentheses, you code the name of the condition that you want to test.
- If a READ command successfully reads a record, the NORMAL condition occurs. If the specified record can’t be found, however, a NOTFND condition occurs.
The RETURN, XCTL, and ABEND commands

Figure 2-14 summarizes three of the commands that control the execution of the programs in a CICS application. The first of these, the RETURN command, is used to return control to CICS and set up the next execution of the program in a pseudo-conversational session. When you use it for that purpose, you include the TRANSID option to specify the trans-id of the program. You also include the COMMAREA option to name the area of working storage that will be stored in the CICS communication area between program executions.

When the RETURN command is issued in this form, the program associated with the trans-id will be started the next time the user presses an attention key. (This should be the same program that issued the RETURN command.) In addition, the data that was placed in the CICS communication area by the RETURN command is passed to that program in the DFHCOMMAREA field. In chapter 5, you’ll see other uses of this command, but this use is common to all pseudo-conversational programs.

Unlike the RETURN command, the XCTL command is commonly used to transfer control from one program to another. In the example in this figure, control is transferred to a menu program named INVMENU. A command like this is often used when the user presses an AID key to exit from a program.

The last command in this figure is the ABEND command. This command ends the program abnormally (called an abend) and displays a message at the terminal. Then, before starting another transaction, the user must press the Clear key to clear the screen. By default, this command also causes a storage dump to be produced, but you can eliminate that by coding the NODUMP option.
The syntax of the RETURN command

```
EXEC CICS
    RETURN [TRANSID(trans-id)]
        [COMMAREA(data-name)]
END-EXEC
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSID</td>
<td>Specifies the one-to four-character name of the transaction to be invoked when the user presses an attention key.</td>
</tr>
<tr>
<td>COMMAREA</td>
<td>Specifies the name of a data area that’s stored in the CICS communication area between program executions so it can be passed to the next execution of a pseudo-conversational program. The next program execution accesses the communication area via its DFHCOMMAREA field.</td>
</tr>
</tbody>
</table>

A RETURN command that sets up the next execution of a program

```
EXEC CICS
    RETURN TRANSID(INQ1)
        COMMAREA(COMMUNICATION-AREA)
END-EXEC.
```

The syntax of the XCTL command

```
EXEC CICS
    XCTL PROGRAM(program-name)
END-EXEC
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM</td>
<td>Specifies the one- to eight-character name of the program to be invoked.</td>
</tr>
</tbody>
</table>

An XCTL command that invokes a program named INVMENU

```
EXEC CICS
    XCTL PROGRAM('INVMENU')
END-EXEC
```

The syntax of the ABEND command

```
EXEC CICS
    ABEND [NODUMP]
END-EXEC
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODUMP</td>
<td>Specifies that a storage dump should not be produced.</td>
</tr>
</tbody>
</table>

An ABEND command that doesn’t produce a storage dump

```
EXEC CICS
    ABEND NODUMP
END-EXEC.
```

Description

- These commands let you control CICS program execution. The RETURN command ends the program, but if you specify the TRANSID option, CICS invokes that trans-id the next time the user presses an AID key. The XCTL command passes control to another CICS program. And the ABEND command terminates the program abnormally.
How to develop a CICS program

The focus of this chapter so far has been on understanding the concepts of CICS programming and the code required to write a CICS program. But there’s more to developing CICS programs than that. In the rest of this chapter, then, you’ll learn more about the procedures for developing a CICS program.

A procedure for developing a CICS program

To develop a CICS program, you need to complete the eight steps outlined in figure 2-15. In step 1, you get the specifications for the program. This just means that you get the complete details about what the program is supposed to do. Then, in step 2, you design the program by developing an event/response chart and a structure chart. In chapter 3, you’ll learn everything you need to know about developing these charts and designing a program.

When you complete the two steps of the analysis and design phase, you can continue with the implementation phase. In this phase, you start by preparing the BMS mapset for the maps that will be used by the program (step 3). You’ll learn how to do that in chapter 4. Then, you code the program (step 4). The main purpose of this book, of course, is to teach you how to do that.

When the program is ready to test, you prepare it for execution by translating, compiling, and link-editing it (step 5). You’ll learn more about that in the next figure.

After you prepare the program, but before you execute it, you must make sure that all of the CICS table entries required to support the program are in place (step 6). For most programs, this means that entries need to be made in three tables: the Program Control Table (PCT), the Processing Program Table (PPT), and the File Control Table (FCT). In the PCT, an entry is required to define the trans-id that’s used to start the program. In the PPT, two entries are required: one for the program, the other for the program’s mapset. If the program uses any files, appropriate entries are required in the FCT. If the program uses other CICS facilities, additional table entries may be required. Although the systems programmer is usually responsible for making these entries, you should at least know what’s required so you can give the administrator the appropriate information.

Once the table entries are in place, you’re ready to execute and test the program (step 7). In this step, you make sure that the program performs according to the program specifications.

Although our eight-step procedure for developing CICS programs separates coding and testing into two distinct steps, we recommend that you code and test in phases. That way, you reduce the amount of code that’s tested in each phase, which makes it easier to isolate the cause of an error. You’ll learn more about this technique, called top-down coding and testing, in chapter 6.
An eight-step procedure for developing a CICS program

Analysis and design
1. Develop a complete set of program specifications.
2. Design the program using an event/response chart and a structure chart.

Implementation
3. Prepare the BMS mapset.
4. Code the program.
5. Prepare the program for execution.
6. Create the necessary CICS table entries.
7. Test the program under CICS.

Documentation
8. Document the program.

Description
- The program specifications should include all the information you need to implement the program. That may include a program overview that describes the processing to be done by the program and the required input and output. It may also include screen layouts for maps used by the program and information on any related copy members or subprograms.
- To design a CICS program, we recommend you develop an event/response chart that identifies each event the program must provide for along with the program’s response to that event. We also recommend that you create a structure chart that represents the overall structure of the program.
- To define the format of the BMS maps that will be displayed by a program, you code a mapset. Then, you assemble the mapset to produce a physical map and a symbolic map.
- To prepare a program for execution, you translate, compile, and link-edit it. If errors occur during this process, you must correct them and then translate, compile, and link-edit the program again.
- The CICS table entries that are required for most programs include an entry in the Program Control Table (PCT) for the trans-id, two entries in the Processing Program Table (PPT) for the program and the mapset, and an entry in the File Control Table (FCT) for each file used by the program.
- When you test a program, you make sure it performs according to its specifications. Although coding and testing are shown as separate steps in this procedure, we recommend you use a technique called top-down coding and testing, or just top-down testing. When you use top-down testing, you code and test in phases.
- The documentation for a program varies from shop to shop.
The last step in this development procedure is to document the program. This just means that you organize the program specifications, structure chart, BMS code, COBOL code, test run data, and other documentation for the program. Later, this documentation is used by the person who has to maintain the program. Since this step varies from one COBOL shop to another, this book will say no more about it. When you’re on the job, you need to find out what’s required in your shop and then put the required documents together.

How to prepare a program for execution

Figure 2-16 presents the procedure for preparing a CICS program for execution. As you can see, you must compile and link-edit the program just as you do any COBOL program. Before you compile a CICS program, however, you must translate it using the CICS command-level translator. The translator converts each CICS command into appropriate COBOL statements that invoke the CICS services specified by the command. The translated program can then be used as input to the COBOL compiler.

In this figure, you can see the translated code for a RECEIVE MAP command. Notice that the original command is still included in the COBOL code, but it’s commented out so it will be ignored by the compiler. After the original command is a series of Move statements followed by a Call statement. The Move statements assign values to the fields that are used as arguments of the Call statement. The Call statement invokes the command-level interface to invoke the required CICS services.

In addition to translating CICS commands, the translator also inserts other code into your program. You already know about one such block of code: the Execute Interface Block. The other code that’s inserted is usually of little interest to you.

In chapter 6, you’ll be introduced to the JCL for running the OS/390 procedure that translates, compiles, and link edits a CICS program. There, you can see how the DD statements identify the files that are used for each step of the procedure.
A CICS program preparation procedure

Translated code for a CICS READ command

*    EXEC CICS
*        RECEIVE MAP('INQMAP1')
*                 MAPSET('INQSET1')
*                 INTO(INQMAP1I)
*    END-EXEC.

MOVE '..}............00109   ' TO DFHEIV0
MOVE 'INQMAP1' TO DFHC0070
MOVE 'INQSET1' TO DFHC0071
CALL 'DFHEI1' USING DFHEIV0  DFHC0070 INQMAP1I DFHDUMMY
               DFHC0071.

Description

- Before you compile a CICS program, you must translate it. The CICS translator converts the CICS commands in the program to Move and Call statements that can be compiled by the COBOL compiler.
- The translator also inserts code into your program, like the Execute Interface Block, that it needs to operate under CICS.
- After you translate the program, you compile and link-edit it just as you would any other COBOL program.
Perspective

The goal of this chapter has been to introduce you to CICS programming. So at this point, you should understand how pseudo-conversational programs work. You should be familiar with the basic CICS commands. You should be familiar with the code you use to develop BMS mapsets. And you should have a basic understanding of the steps required for developing CICS programs.

If you’re an experienced COBOL programmer, you could probably develop a simple CICS program like the one presented in this chapter with just the information presented here. However, there’s a lot more that you need to know to become proficient at developing CICS programs. In the chapters in the next section, then, you’ll learn the details for designing CICS programs, creating BMS mapsets, coding CICS programs, and testing them. After that, the rest of this book will build on those skills by presenting additional CICS commands and techniques.

Incidentally, if you had any trouble understanding the COBOL that’s used in this chapter (not the CICS-specific code), you may want to refresh or enhance your COBOL skills. The best way to do that is to get our COBOL book, Murach’s Structured COBOL. It will not only teach you all of the new COBOL coding skills that you may need, but it’s a terrific reference and an excellent companion for this book.

Terms

<table>
<thead>
<tr>
<th>conversational program</th>
<th>symbolic map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short on Storage (SOS)</td>
<td>event/response chart</td>
</tr>
<tr>
<td>pseudo-conversational program</td>
<td>event</td>
</tr>
<tr>
<td>attention identifier (AID) key</td>
<td>top-down design</td>
</tr>
<tr>
<td>parameter</td>
<td>structure chart</td>
</tr>
<tr>
<td>BMS mapset</td>
<td>Execute Interface Block (EIB)</td>
</tr>
<tr>
<td>assembler</td>
<td>communication area</td>
</tr>
<tr>
<td>physical mapset</td>
<td>response code</td>
</tr>
<tr>
<td>symbolic mapset</td>
<td>exceptional condition</td>
</tr>
<tr>
<td>macro</td>
<td>abend</td>
</tr>
<tr>
<td>attribute byte</td>
<td>top-down coding and testing</td>
</tr>
<tr>
<td>Common User Access (CUA) standards</td>
<td>top-down testing</td>
</tr>
<tr>
<td>screen painter</td>
<td>CICS command-level translator</td>
</tr>
<tr>
<td></td>
<td>CICS translator</td>
</tr>
</tbody>
</table>