Infosys Technologies Limited

Education and Research Department

CICS

October, 1995

Document No.	Authorized By	Ver.Revision	Signature / Date
ER/CORP/CRS/TP01/002	S. YEGNESHWAR	1.00	

Modification Log

Ver. Revision	Date	Author(s)	Description	Review Details
0.00a	31/07/91	K.V.Viswanathan	Initial version	
1.00	19/10/95	Pramod K. Varma	Modified version	**Review team** 1. Aftab 2. Ramesh Adkoli **Review Comments** there is a need for : • more organized approach to the commands • focus on M/F CICS-DB2 stuff and not on PCCICS-XDB stuff • more generic info about capabilities of various CICS commands instead of specifics • separating capabilities of CICS commands from its usage in applications • some more CICS concepts and commands

Course Description

Course number	TP01	Course name	CICS
Author	Pramod K. Varma		
Pre-requisites for attending course: Good knowledge of COBOL, MVS, JCL, and RDBMS			
Estimated effort for preparation	25 days		
Estimated course duration	15 hours (6 sessions of 2.5 hours each)		

Course objectives (course objectives state the major outcomes of the training and indicate the criteria for successful completion)		
Sl#	**Objective**	**Demonstrable knowledge/skills**
1.	To help participants learn fundamentals of transaction processing systems and learn CICS in detail for on-line transaction processing	Good programming skill in CICS. Conceptual understanding of issues in OLTP, CICS

References

- *CICS for microcomputers,*
  Joseph J. Le Bert,
  McGraw-Hill Publishing Company
- *CICS using COBOL,*
  Andrew M. Suhy,
  Wadsworth Publishing Company
- IBM CICS manuals
  - CICS Application Programmer's Primer
  - CICS Application Programmer's Guide

Course Design

Course number	TP01	Course name	CICS
Author	Pramod K. Varma		

Units **(A unit is a self-contained instructional segment that focuses on a single topic or a related set of topics)**			
Sl#	**Unit name**	**Unit objectives** **(A unit could have multiple objectives. Each one will have an importance level, type and evaluation strategy. Pre-requisites, est development time and est delivery time are at the unit level)**	**Est time (deliver)**
1.	Introduction	Why CICS? History of CICS MVS context Batch systems On-line systems CICS like a multi-user operating system CICS as an ongoing MVS job/CICS Region Other platforms (briefly) What is CICS? CICS as an on-line transaction processing monitor What CICS can do for you? Concept of a transaction Concept of a task Concept of a screen Formatted/Unformatted Character data/Stream data 32xx terminal Keys - Program attention keys, Program function keys CICS tables (pct, ppt, fct, dct, tct, etc.) Command level programming/Macro level programming Conversational programming Pseudo conversational programming Need, How's, Comparison with conversational programming	2.5h
2.	Basic Mapping Support	BMS Concept of a map/macro	2.5h

		Symbolic/Physical maps Modified data tag, & Other attributes Map definition Sending/Receiving SDF	
3.	Application programming	OS/VS COBOL (74 standards) VS COBOL II (85 standards) Reentrant code Writing a CICS program using COBOL EXEC interface stubs EIB, Commarea, Exceptional conditions, Handle condition, Ignore condition, Checking for return values (RESP) ABEND Verbs not to be used Access to system information ADDRESS CWA, TWA, TCTUA, ASSIGN Other languages Program types - Inquiry, List, Maintenance Creating a run-unit Assembling a map Translating a CICS program Translator options Compiling a CICS program Compiler options Linking a CICS program Linker options Static/Dynamic Linker libraries	5.0h
4.	Control Operations	Task control ENQUEUE/DEQUEUE SUSPEND Program control LINK, XCTL, RETURN LOAD, RELEASE Storage control GETMAIN, FREEMAIN Temporary storage control WRITEQ, READQ, DELETEQ Transient data control Intrapartition TD queues Extrapartition TD queues Commands - WRITEQ, READQ, DELETE Automatic task initiation (ATI)	2.5h

		Interval control ASKTIME, FORMATTIME START, Automatic task initiation (ATI) CANCEL RETRIEVE DELAY, POST, WAIT EVENT Terminal control Concept of an LU SEND, RECEIVE Other commands	
5.	File Handling	Access methods VSAM, BDAM VSAM considerations ESDS, KSDS, RRDS Recoverable files Commands READ, WRITE, REWRITE, DELETE UNLOCK STARTBR, READNEXT, READPREV, RESETBR, ENDBR Dynamic transaction backout	2.5h

Table of Contents

1. Introduction to CICS

In this unit you will learn:

- what is CICS?
- why is a system like CICS needed in MVS context?
- about IBM keyboard - AID keys, PF keys, and PA keys
- about 3270 terminals
- what is an application?
- what is a transaction?
- what is a task?
- what is LUW (logical unit of work)?
- what is multitasking in CICS?
- basic structure of a COBOL program having embedded CICS commands
- about the important CICS tables - PCT, PPT, FCT, RCT
- how to start and end a transaction through the terminal
- what is conversational Programming?
- what is pseudo Conversational Programming?
- an example of conversational program
- an example of pseudo conversational program

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. 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.

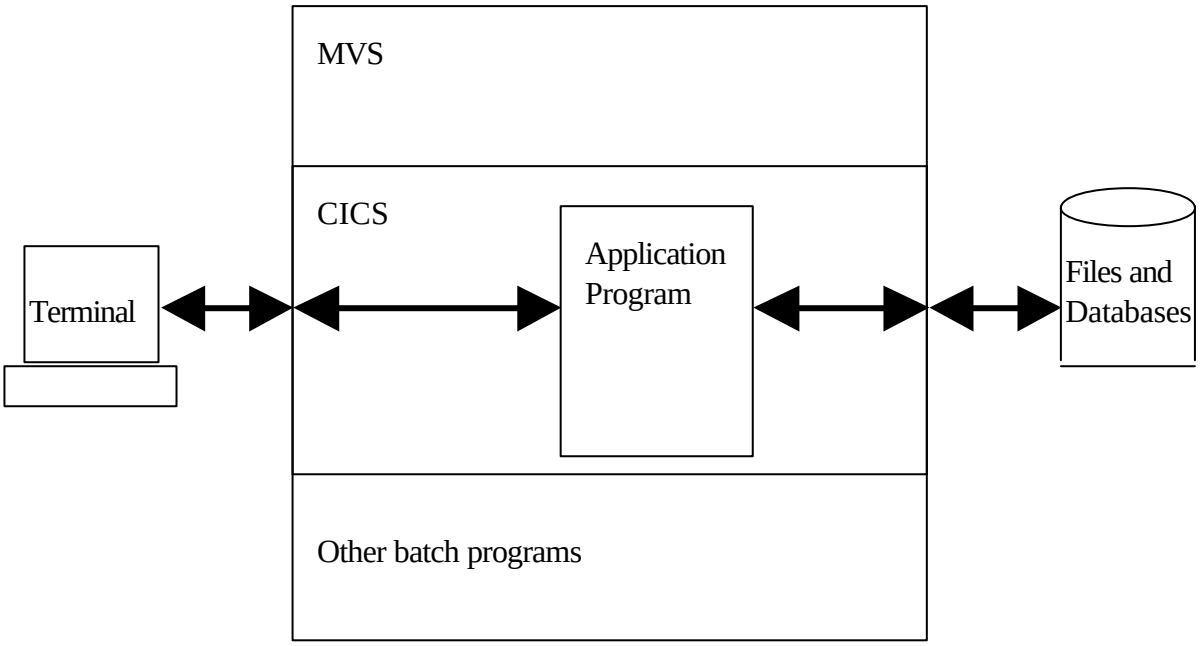


Figure 1. The CICS online environment

The total system is known as a database/data-communication system, usually known as 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.

Why is a system like CICS needed in MVS context?

MVS as an operating system is purely a batch OS, i.e., we can execute only batch programs in MVS. But, nowadays, your users want accurate, up-to-date information 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. CICS is a DB/DC system which allow users to develop and execute online application in an MVS environment.

Why have CICS?

CICS can make application development very much easier by supplying all the basic components needed to handle data communications. This allows you to concentrate on developing application programs and not to concern yourself with the details of data transmission, buffer handling, or the properties of individual terminal devices.

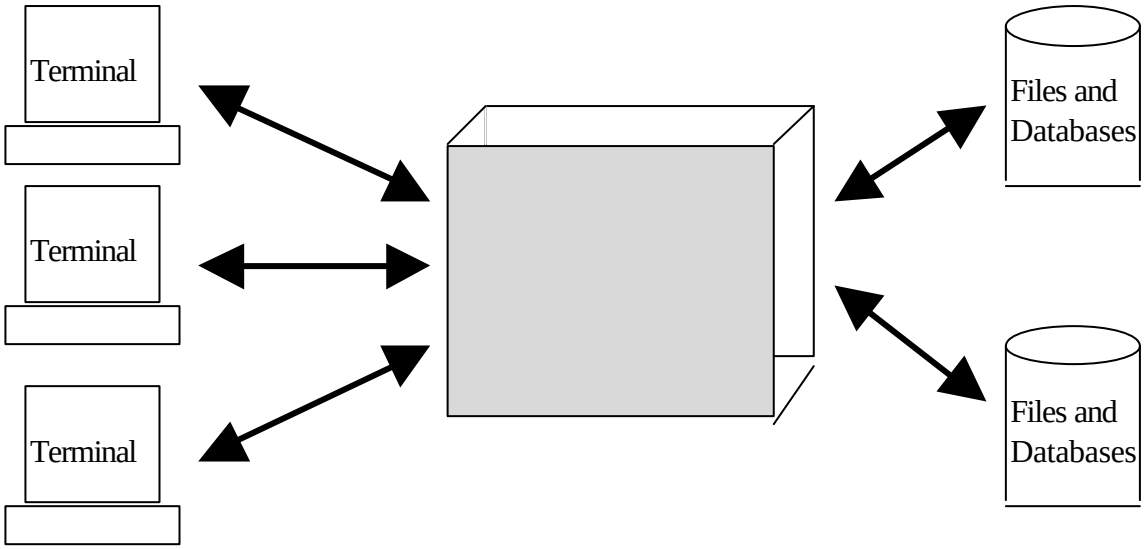


Figure 2. A DB/DC system

What does CICS do?

- ***Telecommunication:*** The functions required by application programs for communication with remote and local terminals and subsystems
- ***Multitasking:*** Control of concurrently running programs serving many online users
- ***Data access & Transaction control:*** Facilities for accessing databases and files, in conjunction with the various IBM database products and data access methods that are available
- ***Inter system communication:*** The ability to communicate with other CICS systems and database systems, both in the same computer and in connected computer systems.

IBM Keyboard

Before going into the next topic, let us look at the keys on a typical IBM keyboard. The keys are divided into two categories - AID (Attention Identifier) keys and non-AID keys. CICS (or in turn the program) can detect only AID keys. When the user types text, numbers etc. using non-aid keys CICS will not even know if the user is typing anything or not. After typing all the input, only when the user presses one of the AID keys, CICS takes control.

AID Keys: ENTER, PF1 to PF24, PA1 to PA3, CLEAR
non-AID Keys: All other keys for alphabets, numeric, punctuation etc.

AID keys are further split into PF (Program Function) keys and PA (Program Access) keys. The difference between them is that PF keys allow the transfer of data from terminal to CICS where as PA keys do not.

PF Keys: ENTER, PF1 to PF24
PA Keys: PA1 to PA3, CLEAR

3270 terminals

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.

When your application program writes to the screen, 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 over typed from the keyboard later, and so on.

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. 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. More discussion on screen will be done when we discuss Basic Mapping Support.

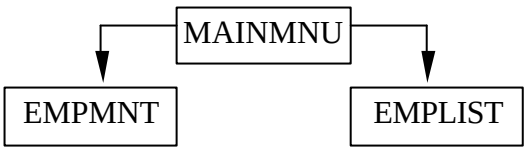
What is an application?

An application is a collection of programs which accomplish some specific user work. For example, in a company, we could think about having an application which allow users to:

- maintain employee details
- view the employee list etc.

For the above application, we may have the following 2 programs:

- MAINMNU - which allow users to choose any one of the above two functions
- EMPMNT - which allow users to add, delete, modify data for a specified employee
- EMPLIST - which allow users to browse through employee details



What is a transaction?

A CICS transaction is a collection of logically related programs in an application, i.e., the whole application could be logically divided into several transactions. Users tell CICS what type of transaction they want to do next by using a ***transaction identifier*** which is 1 to 4 characters long. The programmer associates one of the programs to a transaction identifier and this program in turn invokes all other programs which belong to that transaction. Finally, a single transaction , when run, will carry out the processing needed. See figure 3.

In the above example, let us make MAINMNU and EMPMNT as single transaction and EMPLIST as another transaction. Let those transaction identifiers be MENU and LIST respectively.

What is a task?

Users invoke an application by using one of the transaction identifiers. 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.*** That means, a transaction could be completed through several tasks (we will discuss this in detail when we deal with pseudo-conversational programming). Like transactions, tasks also can span across several programs.

A task can receive data from and send data 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. But note that 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.

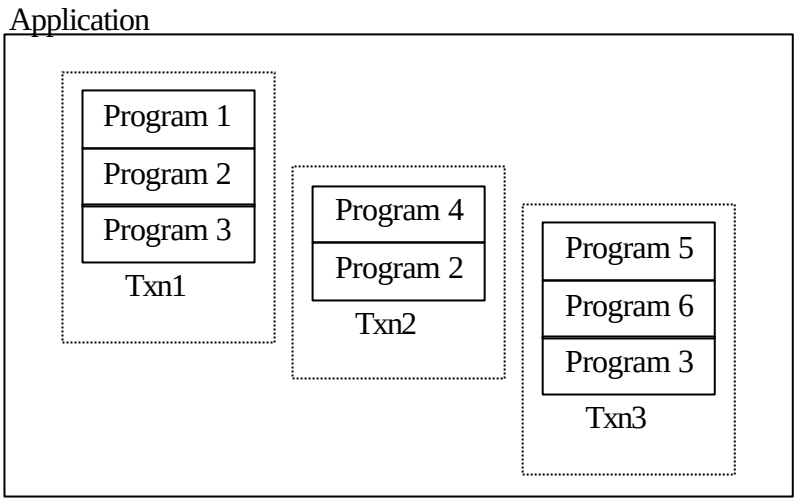


Figure 3. An application divided into transactions

What is an LUW?

An LUW (or logical unit of work) is a piece of work (like file update, database update) which, when done, is fully done or not done at all. (In RDBMS terminology, an LUW starts when the first SQL command executes and ends when a COMMIT or ROLLBACK is done.) In CICS, a task can contain several LUWs (we will discuss this later when we deal with the CICS command 'SYNCPOINT').

Thus, an application is a collection of programs which are logically grouped to form several transactions. Each transaction when gets executed, a task is created. Within each task we could have multiple LUWs.

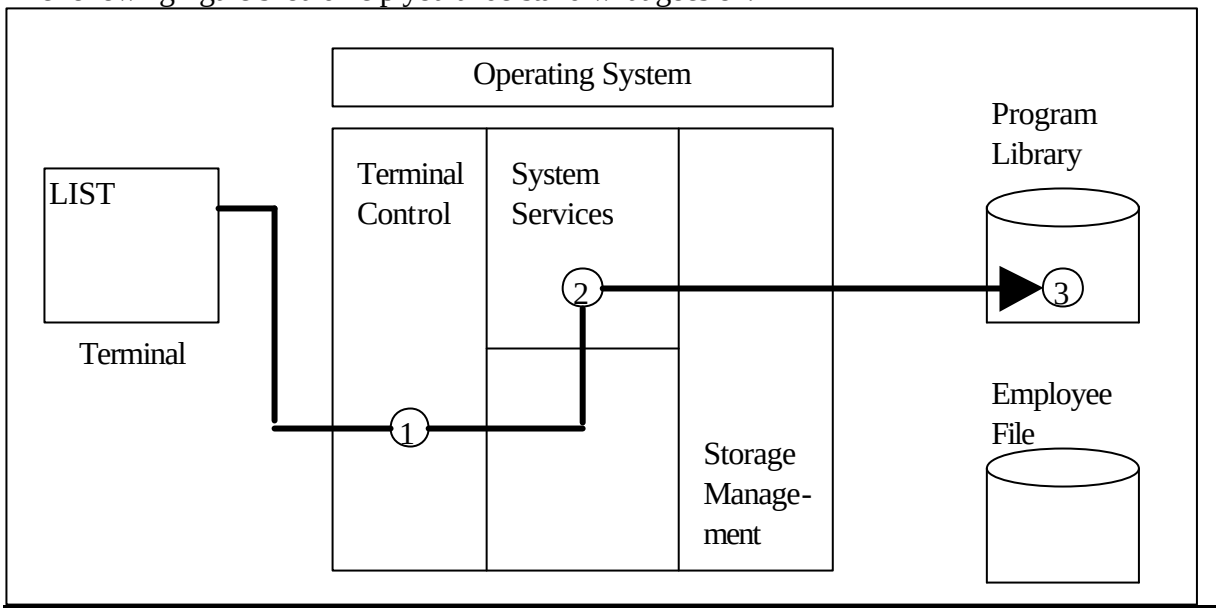
Starting a transaction

To begin an online session, first end user sign on to CICS. Once signed-on, they invoke the particular application they intend to use by typing the transaction identifier of the logically first transaction (in the above example, MENU) at the start of their initial request. Other ways of invoking a transaction: set up a particular program function(PF) key to invoke a transaction with a single keystroke, associate a terminal to invoke a particular transaction.

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 is not already loaded), and starts a task to execute it. 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***. When the transaction which user invoked is complete, CICS returns the terminal to a standby state.

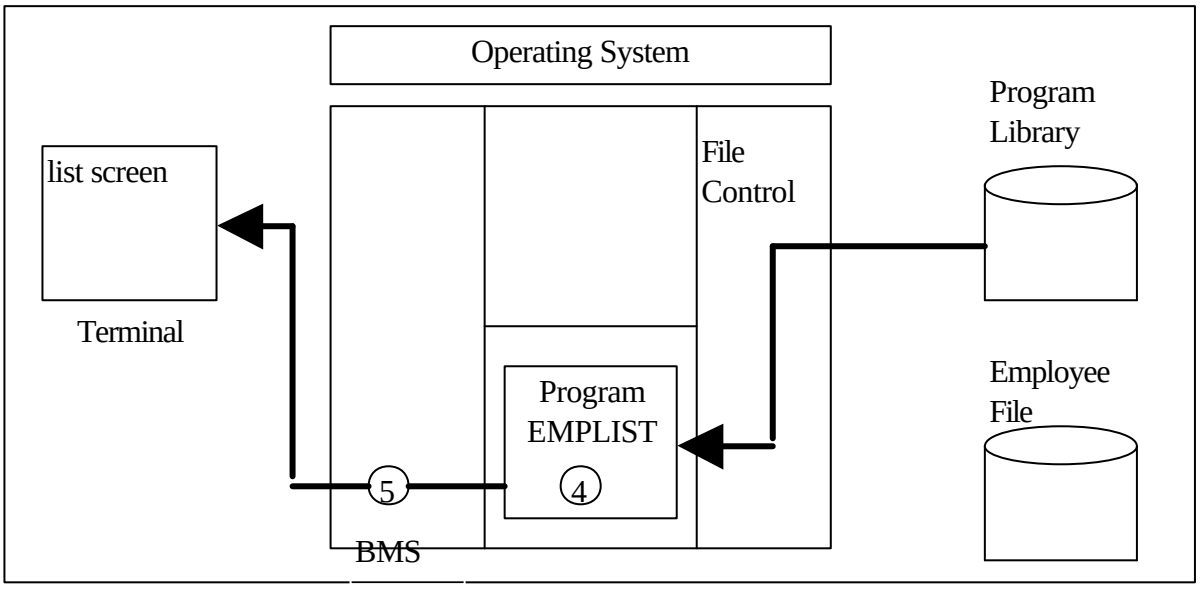
The following figure should help you understand what goes on.



PICTURE 3

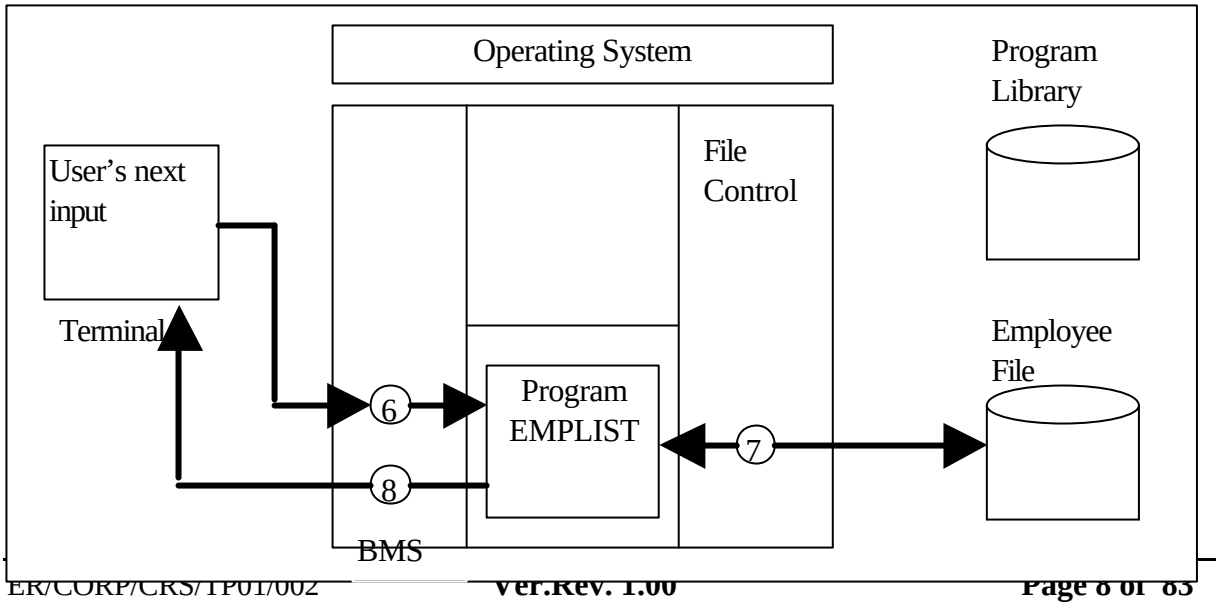
The flow of control during the list transaction (transaction identifier LIST) is shown by the sequence of numbers 1 to 8 on the panels. The meanings of these eight stages are as follows:

1. Terminal control accepts characters LIST, typed at the terminal, and puts them in working storage.
2. System services interpret the transaction id LIST as a call for an application program called EMPLIST. 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 EMPLIST 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.



PICTURE 5

6. BMS and terminal control also handle the user's next input, returning it to EMPLIST which then invokes....
7. File control to read the appropriate file for the information the terminal user has requested. Finally, EMPLIST invokes....
8. BMS and terminal control to format the retrieved data and present it on the terminal.

Figure 3. The flow of control during a transaction

The task continues to run until it reaches a place in the program at which it is 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. Thus the amount of processing done by the application program between CICS calls should be minimal. That means have minimum COBOL code between two CICS commands. CICS passes control back to the operating system to allow other batch work to run only when there is no work to do on behalf of any CICS task. This allows CICS to maintain the priority of online working over batch work in other MVS address spaces.

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

Features of COBOL programs running under CICS

- Embedded CICS commands
- Programs run under CICS.
- No File control section and open/close statements.
- No Screen section or Display/Accept statements.
- Must have a Linkage section.

CICS and OS scheduling

- CICS runs in one 'region' or 'partition' or 'address space' and handles scheduling for programs running under it.
- Since CICS is itself a batch job, it is scheduled by the OS and will compete with other batch jobs for CPU time. Therefore CICS is usually run with high priority.

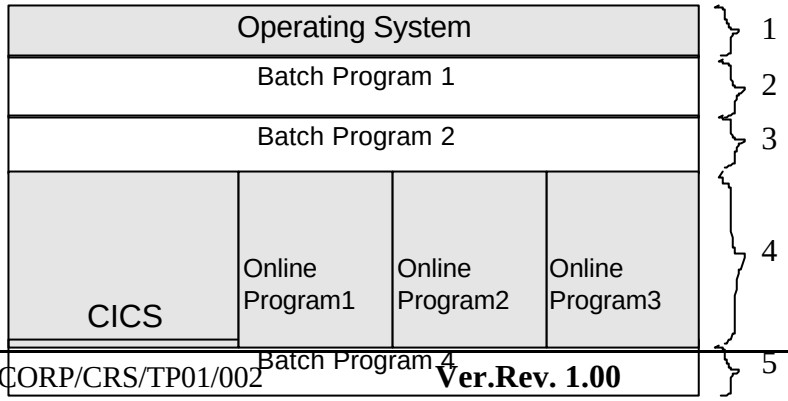


Figure 4. Each numbered rectangular area depicts a collection of page frames in main memory allocated to different processes.

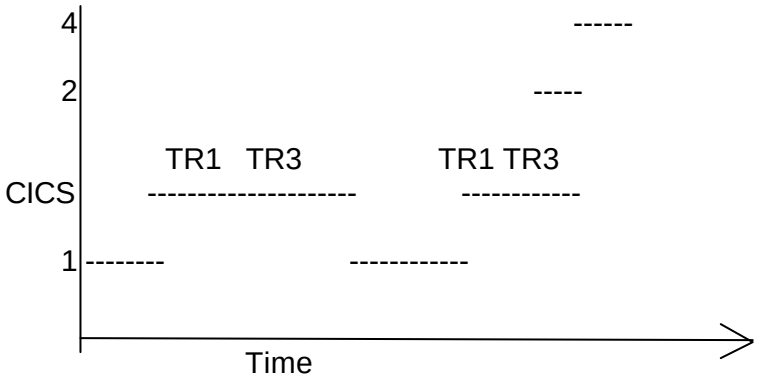


Figure 5. While OS gives time to CICS, CICS schedules all the programs under it.

Important CICS tables - PCT, PPT, FCT, RCT

PCT - ***Program Control Table***

Contains:

Transaction ids (Transid) and the corresponding program names.

Transid is unique.

PPT - ***Processing Program Table***

Contains:

Program name/Mapset name, Language, Size, Main storage address (if it is loaded), Task Use Counter, Load library address (address of the load module in the disk), Whether main storage resident.

Program/Mapset name is unique.

When a transaction id is received by CICS, it gets the corresponding program name from PCT. Then it checks if the associated program is already loaded. If so it merely increases the task use counter for the program by 1 in PPT. If the program is not loaded, then it gets the load library address from PPT and loads the program and make the task use counter 1. Similarly, as tasks complete, the task use counter of that program is decremented. When the task use counter becomes zero then CICS may free the main memory where the load module is loaded, if it has not been designated as being main-storage resident.

FCT - ***File Control Table***

Contains:

File name (datasets), File type, record length (maximum and minimum in case of variable length records), etc.

All the files we are using in a CICS program must be declared in FCT and they are opened and closed by CICS itself. (i.e., we will not have file control section, open file, close file in a CICS program.)

RCT - ***Resource Control Table***

Contains:

Transaction identifier, DB2 plan name (this table is used only if there are SQL statements in your transaction)

Conversational and Pseudo-conversational programming

In a typical online transaction which takes some user input and process it, we can foresee several problems:

- Every transaction involves a wait for the user to enter data, and the 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 a one-user-at-a-time resource during one of the waits, contradicting the guideline to minimize the duration of transactions that use such resources.

Take, for example, the employee list transaction. The sequence of major events would be as shown here in Figure 6:

Operations
1. Display list screen.
2. Wait for user input.
3. Receive list screen.
4. Read employee records from the employee file.
5. Display records in formatted form.
6. Wait for the user input.
7. Receive the screen.
8. Depending on the function key populate new list.
9. Redisplay the list.

Figure 6. The conversational sequence of the list transaction

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.

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. That means every time the transaction is run, a new task is created. In the case we were just looking at, the pseudoconversational structure is shown in Figure 7:

Tasks	Operations
First	1. Display list screen.
Second	3. Receive list screen.
	4. Read employee records from the employee file.
	5. Display the records in formatted form.
Third	7. Receive the screen.
	8. Depending on the function key populate new list.
	9. Redisplay the list.

Figure 7. The pseudoconversational structure

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

Let us look at a program which asks the user for the name and responds with a Hello Mr.X . We will develop our employee transactions in the following units.

A Conversational program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CONV.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-PROMPT            PIC X(07)
    VALUE 'NAME'.
01 WS-INPUT             PIC X(20).
01 WS-MESSAGE           PIC X(10)
    VALUE 'HELLO MR.'.
01 WS-LENGTH       PIC S9(4) COMP
    VALUE +20.
LINKAGE SECTION.
01 DFHCOMMAREA                PIC X(01).
PROCEDURE DIVISION.
000-MAIN.

    EXEC CICS SEND
        FROM (WS-PROMPT)
        LENGTH (7)
        ERASE
    END-EXEC.

    EXEC CICS RECEIVE
```

```
            INTO (WS-INPUT)
            LENGTH (WS-LENGTH)
        END-EXEC.

        EXEC CICS SEND
            FROM (WS-MESSAGE)
            LENGTH (10)
            ERASE
        END-EXEC.

        EXEC CICS SEND
            FROM (WS-INPUT)
            LENGTH (20)
        END-EXEC.

        EXEC CICS RETURN
        END-EXEC.

    099-MAIN-EXIT.
            EXIT.
```

The Pseudo Conversational version of the above program:

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. PSCONV.
    ENVIRONMENT DIVISION.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 WS-PROMPT            PIC X(07)
        VALUE 'NAME'.
    01 WS-INPUT             PIC X(20).
    01 WS-MESSAGE           PIC X(10)
        VALUE 'HELLO MR.'.
    01 WS-LENGTH        PIC S9(4) COMP
        VALUE +20.
    01 WS-COMMAREA              PIC X(01).

    LINKAGE SECTION.
        01 DFHCOMMAREA          PIC X(01).

    PROCEDURE DIVISION.
    000-MAIN.
      IF EIBCALEN = 0
        PERFORM 100-INITIAL-ENTRY THRU
                199-INITIAL-ENTRY-EXIT
      ELSE
        PERFORM 200-NEXT-ENTRY THRU
                299-NEXT-ENTRY-EXIT
      END-IF.

      EXEC CICS RETURN
          TRANSID (EIBTRNID)
          COMMAREA (WS-COMMAREA)
```

```
           LENGTH (1)
       END-EXEC.

     099-MAIN-EXIT.
           EXIT.
    *---------------------------------------------------
     100-INITIAL-ENTRY.
           EXEC CICS SEND
                     FROM (WS-PROMPT)
                     LENGTH (7)
                     ERASE
           END-EXEC.

     199-INITIAL-ENTRY-EXIT.
                     EXIT.

     200-NEXT-ENTRY.
           EXEC CICS RECEIVE
                     INTO (WS-INPUT)
                     LENGTH (WS-LENGTH)
           END-EXEC.

           EXEC CICS SEND
                     FROM (WS-MESSAGE)
                     LENGTH (10)
                     ERASE
           END-EXEC.

           EXEC CICS SEND
                     FROM (WS-INPUT)
                     LENGTH (20)
           END-EXEC.

           EXEC CICS RETURN
           END-EXEC.

     299-NEXT-ENTRY-EXIT.
                     EXIT.
```

Review Questions

1. What is CICS?
2. Why do we need a system like CICS in MVS context?
3. Name few functions CICS do which are normally performed by an operating system?
4. What are AID keys? Why do we need them?
5. What are PF & PA keys?
6. What is the difference between PF keys and PA keys?
7. Explain what are transactions and tasks in CICS context.
8. Why is CICS said to be multitasking?
9. What are the few ways of starting a new transaction?
10. Since CICS is a batch program and MVS schedules CICS as any other batch job, who schedules online CICS transactions?
11. What are the few CICS tables which we discussed? And what do they contain?
12. Explain briefly the difference between a conversational program and a pseudo-conversational program. Discuss the advantages and disadvantages.

2. Basic mapping support (BMS)

In this unit you will learn:

- what is BMS
- why do we use BMS
- how to create a BMS
- the DFHMDF, DFHMDI, DFHMSD macros in detail
- what are physical and symbolic maps and their use
- how do we create these maps
- how to send and receive a map
- how to change attributes of a field through the program
- how to position the cursor on a field through the program (symbolic cursor positioning)
- sending control information using SEND CONTROL command
- how to detect which key the user has pressed
- what is EIB and some of the important contents of it
- what is MAPFAIL and when does it occur

What BMS does?

In the example we discussed earlier (programs MAINMNU, EMPMNT, EMPLIST), the first thing we need is to create screens which has fields in the proper positions with proper attributes etc. i.e, we need to design formatted screens which are then send to the input terminal. This requires the use of CICS terminal input/output services. In particular, we will need to use Basic Mapping Support (BMS).

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. Before we start to look at the BMS commands, we need to explain in a little more detail what BMS does for you. It is probably easiest to define what BMS does by examining the menu screen we need. You can see what it looks like in Figure 8.

To help us in this discussion, we have added row and column numbers to the figure and underlined the fields that would otherwise not show unless filled in with data. We have also marked the position of the attribute byte for the "stopper" fields with a vertical bar (|) and for other fields with a plus sign (+). (Notice that the attribute byte of every field occupies a position on the screen just before the field although these markers will not show up on the screen we're building.)

```
          1         2         3         4         5         6         7         8
  12345678901234567890123456789012345678901234567890123456789012345678901234567890
 1+MAINMNU                          +MAIN MENU                       +DATE:+MM/DD/YY
 2                                                                   +TIME:+HH:MM:SS
 3
 4
 5
 6
 7
 8                     +1. EMPLOYEE RECORD MAINTENANCE
 9
10
11                     +2. EMPLOYEE DETAILS BROWSE
12
13
14
15                          +SELECTION:+_|
16
17
18
19
20
21+_____________________________(msg field)_____________________
22
23 +PF1 - HELP     +PF3/CLEAR - EXIT     +ENTER - PROCESS SELECTION
24
```

Figure 8. A detailed look at the menu screen

You define this screen with BMS macros, which are a form of assembler language. When you have defined the whole map, you use job control language (JCL) to assemble it. Figure 9 shows a sample JCL that assembles a BMS.

```
//EDGS12TH JOB LA3020,'MAINMNU - ASSMB',
//*     RESTART=*,
//      CLASS=D,MSGCLASS=X,
//      USER=TRAINEE,PASSWORD=TRAINEE
//************************************
//* JCL FILE IS TRAINEE.COMP.JCL(MAPM) *
//*****************************************************
//*  PHYSICAL AND SYMBOLIC ASSEMBLY OF BMS MAPS         *
//*  CHANGE ALL MAINMNU TO YOUR MAP NAME                *
//*****************************************************
//*
//JOBLIB    DD DSN=SYS0.DB2.PROD.DSNLOAD,DISP=SHR
//          DD DSN=EDSUBRT.UTIL.LOADLIB,DISP=SHR
//PROCLIB   DD DSN=EDGS03T.DART.PROCLIB,DISP=SHR
//*******************************************
//*    BMS MAP - PHYSICAL AND SYMBOLIC ASSEMBLY *
//*******************************************
//STEP2   EXEC DFHMAPS,OUTC='*',
//             CICSREG='TG',
//             DSCTLIB='TRAINEE.PROJ.COPLIB',
//             MAPNAME=MAINMNU
//** BY DEFAULT THE LOAD MODULE GOES IN THE CICSTB REGION
//COPY.SYSUT1 DD DSN=TRAINEE.COMP.MAP (MAINMNU),DISP=SHR
//*
```

Figure 9. A sample JCL that assembles a BMS

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 other assembly produces a ***symbolic map***.

The physical map contains the executable to do the following:

- Build the screen, with all the titles and labels in their proper places and all the proper attributes for the various fields.
- Merge the data from your program in the proper places on the screen when the screen is sent to the terminal.
- Extract the data for your program when the screen is received.

The symbolic map contains the following:

- A COBOL structure which defines all the fields which you might want to manipulate through the program, so that you can refer to them by name. This data structure gets placed in a library. You can copy it into your program to refer to screen fields.

CICS allows the application programmer to send either the physical map or symbolic map or both maps. This reduces the network traffic between the system and the terminal.

The BMS macros

To show you how this works, let us go ahead and define the menu map. Figure 10 shows the complete BMS macro. We will explain the three map-definition macros as we go. Do not be put off by the syntax; it is really quite simple when you get used to it. We will go from the inside out, starting with the individual fields.

The DFHMDF macro: generate BMS field definition

Now we can define the fields in our menu map. We will "do" the fields in order. Although this is no longer required in CICS, it is a good idea for clarity. Figure 10 shows the DFHMDF macros for the menu map.

```
MENUSET   DFHMSD TYPE=&&SYSPARM,                                      X
               CTRL=(FREEKB,FRSET),                                   X
               LANG=COBOL,                                            X
               STORAGE=AUTO,                                          X
               TERM=3270,                                             X
               TIOAPFX=YES
MENUMAP   DFHMDI SIZE=(24,80),                                        X
               TIOAPFX=YES
          DFHMDF POS=(01,01),                                         X
               ATTRB=(ASKIP,NORM),                                    X
               LENGTH=007,                                            X
               INITIAL='MAINMNU'
          DFHMDF POS=(01,34),                                         X
               ATTRB=(ASKIP,NORM),                                    X
               LENGTH=009,                                            X
               INITIAL='MAIN MENU'
          DFHMDF POS=(01,65),                                         X
               ATTRB=(ASKIP,NORM),                                    X
               LENGTH=005,                                            X
               INITIAL='DATE:'
DATEM     DFHMDF POS=(01,71),                                         X
               ATTRB=(PROT,NORM),                                     X
               LENGTH=008
          DFHMDF POS=(02,65),                                         X
               ATTRB=(ASKIP,NORM),                                    X
               LENGTH=005,                                            X
               INITIAL='TIME:'
TIMEM     DFHMDF POS=(02,71),                                         X
               ATTRB=(PROT,NORM),                                     X
               LENGTH=008
          DFHMDF POS=(08,22),                                         X
               ATTRB=(ASKIP,NORM),                                    X
               LENGTH=031,                                            X
               INITIAL='1. EMPLOYEE RECORD MAINTENANCE'
          DFHMDF POS=(11,22),                                         X
               ATTRB=(ASKIP,NORM),                                    X
               LENGTH=026,                                            X
               INITIAL='2. EMPLOYEE DETAILS BROWSE'
          DFHMDF POS=(15,28),                                         X
               ATTRB=(ASKIP,NORM),                                    X
               LENGTH=010,                                            X
               INITIAL='SELECTION:'
SELECTM   DFHMDF POS=(15,39),                                         X
               ATTRB=(UNPROT,NORM,FSET),                              X
               LENGTH=001
          DFHMDF POS=(15,41),                                         X
               ATTRB=(ASKIP,DRK),                                     X
               LENGTH=001
MSGM      DFHMDF POS=(21,01),                                         X
               ATTRB=(PROT,BRT),                                      X
               LENGTH=060
          DFHMDF POS=(22,06),                                         X
```

```
            ATTRB=(ASKIP,NORM),                                    X
            LENGTH=010,                                            X
            INITIAL='PF1 - HELP'
      DFHMDF POS=(22,22),                                          X
            ATTRB=(ASKIP,NORM),                                    X
            LENGTH=018,                                            X
            INITIAL='PF3 / CLEAR - EXIT'
      DFHMDF POS=(22,46),                                          X
            ATTRB=(ASKIP,NORM),                                    X
            LENGTH=025,                                            X
            INITIAL='ENTER - PROCESS SELECTION'
      DFHMSD TYPE=FINAL
      END
```

Figure 10. The DFHMDF macros for the menu map

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

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

fldname: This is the name of the field, as you will use it in your program (or almost so, as we will explain). Name every field that you intend to send or receive in your program, but do not name any field that is 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 is the position relative to the beginning of the map.) 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.

LENGTH=number This is the length of the field, not counting the attribute byte.

INITIAL='text' This is the character data for an output field. It is 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 will code:
INITIAL='MENU'

ATTRB=(attr1,attr2,attr3,attr4,attr5) These are the attributes of the field.

attr1 = ASKIP / PROT / UNPROT

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.

attr2 = NORM / BRT / DRK

NORM - Normal display intensity, BRT - Bright (highlighted) intensity,
DRK - Dark (not displayed).

attr3 = FSET / FRSET

FSET - Turns on the modified data tag. This causes the field to be sent from terminal to memory on the subsequent read whether or not the user keys into it. If you do not specify this, the field is sent back only if the user changes it (that state is called FRSET).

attr4 = NUM

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

attr4 = IC

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 do not need the ATTRB parameter. If you omit it, the field will be ASKIP and NORM, with FRSET and no IC specified.

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 will 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.

The DFHMDI macro: generate BMS map definition

Now that we have sorted out all the fields of the map, we need to give a map definition. The map definition for the menu screen is given in figure 11.

```
MENUMAP  DFHMDI SIZE=(24,80),                                    X
               TIOAPFX=YES
```

Figure 11. Map definition for the main menu screen

The definition and parameters are as follows:

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

mapname This is the map's name, which you will use when you issue a CICS command to send or receive the map. It is 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 are starting a new map.

SIZE=(line,column) This parameter gives the size of the map. You need it for the type of maps we are using. BMS allows you to build a screen using several maps, and this parameter becomes important when you are doing that.

CTRL=(ctrl1,ctrl2,ctrl3,ctrl4) This control parameter defines the screen and keyboard control information that you want sent along with a map.

ctrl1 = PRINT Specify this for any map that might be sent to a printer terminal.

ctrl2 = FREEKB This means "free the keyboard." The keyboard locks automatically (why?) 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 will 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.

ctrl3 = ALARM This parameter sounds the audible alarm at the terminal (if the terminal has this feature; otherwise it does nothing).

ctrl4 = FSET / FRSET This parameter has the same meaning as in the control parameter of DFHMDF macro.

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, even for a single map we must define a map set. For efficiency reasons, it is 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 are loaded together at execution time.

The map set definition for the menu screen is given in figure 12.

```
MENUSET   DFHMSD TYPE=&&SYSPARM,                                  X
             CTRL=(FREEKB,FRSET),                                 X
             LANG=COBOL,                                          X
```

```
STORAGE=AUTO,                                    X
TERM=3270,                                       X
TIOAPFX=YES
```

Figure 12. Map set definition for the main menu screen

The definition and parameters are as follows:

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

setname This is the name of the map set. You will use it when you issue a CICS command to receive or send one of the maps in the set. It is 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.

> ***Note:*** This mapset name is the load module name and should be there in PPT. This does not mean that the BMS file name and the map set name be the same. But normally, to avoid ambiguity, they are the same.

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

TYPE=type TYPE governs whether the assembly produces the physical map or the symbolic description (DSECT). If we use TYPE=MAP, the physical map is produced and if we use TYPE=DSECT, the symbolic map is produced. But, in practice, TYPE=&&SYSPARM is used which produces both physical and symbolic maps in one run of the assembly JCL. It avoids changing the map to create each separate maps.

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=language This decides the language of the DSECT structure, for copying into the application program. It can be COBOL, PL/I, 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 do not use these maps at the same time, you should omit STORAGE=AUTO to cut down the size of your WORKING-STORAGE.

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. They are reserved for CICS control information. (We will discuss this a little later.)

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 is another macro, DFHMSD TYPE=FINAL, followed by the assembler END statement:

```
+-----------------------------------------------------------------+
¦                                                                 ¦
¦        DFHMSD TYPE=FINAL                                        ¦
¦        END                                                      ¦
¦                                                                 ¦
+-----------------------------------------------------------------+
```

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 is 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 you like.
- Separate the parameters by one comma (no spaces), but do not put a comma after the last one.
- For ease of readability and maintainability, code each of the parameters in separate lines.
- If you cannot get everything into 71 columns, stop after the comma 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 have 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 of 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 is easy to forget, but this upsets the assembler.
- Always surround initial values by single quote marks. If you need a 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. Put an asterisk (*) in column 1, and use any part of columns 2 through 71 for your text. Do not go beyond 71.

Using BMS: more detail

Symbolic 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 have 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 is 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.

The symbolic 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.

Copying the map DSECT into a program

After assembling the BMS with a DSECT option, 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. For example, to get the symbolic descriptions for our maps in a program, we will write:

COPY MENUSET.

(Note that although the assembler generates a symbolic map copy book for you, some times you may choose to write your own structure with different names as you need.) Figure 13 shows you 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 is followed by similar structures for the other maps.

```
01  MENUMAPI.
    05  FILLER                          PIC X(12).
    05  DATEML                          PIC S9(4) COMP.
    05  DATEMF                          PIC X(01).
    05  FILLER REDEFINES DATEMF.
        10  DATEMA                      PIC X(01).
    05  DATEMI                          PIC X(008).
    05  TIMEML                          PIC S9(4) COMP.
    05  TIMEMF                          PIC X(01).
    05  FILLER REDEFINES TIMEMF.
        10  TIMEMA                      PIC X(01).
    05  TIMEMI                          PIC X(008).
    05  SELECTML                        PIC S9(4) COMP.
    05  SELECTMF                        PIC X(01).
    05  FILLER REDEFINES SELECTMF.
        10  SELECTMA                    PIC X(01).
    05  SELECTMI                        PIC X(001).
    05  MSGML                           PIC S9(4) COMP.
    05  MSGMF                           PIC X(01).
    05  FILLER REDEFINES MSGMF.
        10  MSGMA                       PIC X(01).
    05  MSGMI                           PIC X(060).
01  MENUMAPO  REDEFINES MENUMAPI.
    05  FILLER                          PIC X(12).
    05  FILLER                          PIC X(03).
    05  DATEMO                          PIC X(008).
    05  FILLER                          PIC X(03).
    05  TIMEMO                          PIC X(008).
    05  FILLER                          PIC X(03).
    05  SELECTMO                        PIC X(001).
    05  FILLER                          PIC X(03).
    05  MSGMO                           PIC X(060).
```

Figure 13. 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 labeled (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, OUT would have produced only the output part. (These redefinitions are not necessary. We could always use the same structure for input and output purposes.)

The generated subfields

We gave names to eight field definitions in the menu map: DATEM, TIMEM, SELECTM, and MSGM. Notice that for each of these map fields, 3 data subfields are generated. Each subfield has a name consisting of the field name in the map and a one-letter suffix (L or F/A or I/O). 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 14; here selection field is entered by the user, date, time, and message fields were send by the program.

```
MAINMNU                          MAIN MENU                    DATE: 09/23/95
                                                              TIME: 10:10:00




                    1. EMPLOYEE RECORD MAINTAINANCE

                    2. EMPLOYEE DETAILS BROWSE


                         SELECTION: 1




PLEASE TYPE IN YOUR SELECTION AND PRESS ENTER

  PF1 - HELP      PF3 / CLEAR - EXIT      ENTER - PROCESS SELECTION
```

Figure 14. The menu screen at work. All bold characters show DSECT fields.

Ultimately, when received, BMS puts the user's data into our program's WORKING-STORAGE, along with some control information. Look at Figure 13 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 are reserved for CICS control information, and are of no concern to the application program.

The first suffix is L, which stands for "length." SELECTML is the number of characters that the user keyed into the SELECTM 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, SELECTML will be one; DATEML will be 8 if its MDT were turned on while sending, and so on.

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 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 is not on, the program does not receive what is 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 SELECTM field in the menu map has a length of 1, the SELECTMI subfield is given a PICTURE value of X(01) in the symbolic map description.

If the user does not fill in the whole field, BMS pads out the field to its maximum length. If a field has the NUM attribute, it is filled on the left with leading (decimal) zeros; otherwise it is filled on the right with spaces.

The two data fields suffixed by A and O for a map field concern output rather than input, even though one of them appears in the "input" part of an INOUT map. They are redefinitions of F and I. The first one is suffixed by A (for "attribute"). When you are 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 A instead of 1 or 2. We'd want to bounce the menu screen straight back to the user with the selection field highlighted, to show our displeasure at finding a character A there. To do so, we'd simply need to move the character that represented the attributes, say, UNPROT and BRT, to SELECTMA and move appropriate message to MSGMO before sending the map.

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 15:

```
+-----------------------------------------------------------------+
|                                                                 |
|   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 15. Attribute values for the IBM 3270 data stream

You will 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. Some of the meanings are given below.

Variable	Protection	Intensity	Modified Data Tag (FSET/FRSET)
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 16. Attribute values used in the Book

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

```
MOVE DFHBMBRY TO SELECTMA
```

before sending the map back to the terminal. This would make that field unprotected-bright. It also sets the modified data tag off.

The second redefinition field for a map field is named with a suffix of O (for "output"). It is the data that you want displayed in the map field when you send it. We could have:

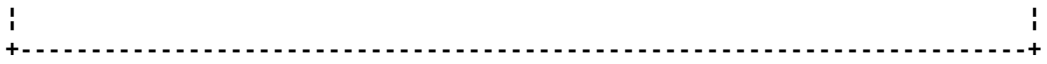
```
MOVE 'PLEASE TYPE IN EITHER 1 OR 2 AS SELECTION' TO MSGMO
```

before sending the map to have an error message displayed on that field.

The SEND MAP command

The SEND MAP command writes formatted output to a terminal. The first time we need to send a map to a terminal occurs in program MAINMNU, where we display the menu screen. The command we need is:

```
EXEC CICS SEND
          MAP('MAINMNU')
          MAPSET('MENUSET')
          MAPONLY
          ERASE
END-EXEC.
```



It syntax 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 is required. Put it in quotes if it is a literal.

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

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

FROM this option is used if you have decided to use a different DSECT name, you must use the option **`FROM (dsect-name)`** along with SEND MAP command.

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 will use this option when we send the menu map the first time, because we will 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 is 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 logically use this option only after you have sent the same map without using the DATAONLY option.

ERASE causes the entire screen to be erased before what you are 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 is most often used in preparing to read new data from a map that is already on the screen. Do not 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 are 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 will explain in coding our example.

CURSOR can be used in two ways to position the cursor. If you specify a value after CURSOR, it is 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. Why column 2? Because the attribute byte goes in column 1, and we want the cursor to appear under the first character of data.

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 topic "Positioning the cursor" later. This second manner is recommended in general, rather than the first, so that changes in the map layout do not 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.

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.

Using SEND MAP in our example

The first time when user invokes the transaction, the program MAINMNU should send the map with the following command:

```
EXEC CICS SEND
          MAP('MAINMNU')
          MAPSET('MENUSET')
          MAPONLY
          ERASE
END-EXEC.
```

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('MAINMNU')
          MAPSET('MENUSET')
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 MAINMNUO. If you have a different DSECT name use the option **`FROM (symbolic-map-name)`** along with SEND MAP command.

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('MAINMNU')                                      |
|             MAPSET('MENUSET')                                   |
|             CURSOR                                              |
|             DATAONLY                                            |
|             ERASEAUP                                            |
|   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 (Why so?).

Positioning the cursor

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 selection field all we have to do is:

```
MOVE -1 TO SELECTML
```
and specify CURSOR in our SEND MAP command.

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.

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.

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 are 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. Otherwise, use an **`INTO (symbolic-map-name)`** to specify the working-storage structure into which data has to be receieved.

For example,

```
EXEC CICS RECEIVE MAP('MENUMAP') MAPSET('MENUSET')
          RESP(RESPONSE) END-EXEC.
```

will bring the input data into a data area named MENUMAPI, which is expected to have exactly the format produced by the DSECT for map MENUMAP. (We will explain RESP(RESPONSE) in the next unit.)

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 selection field of the menu map:

```
IF SELECTML > 0, MOVE ...
```

Or we could examine the input in that field:

```
IF SELECTMI = 'A' PERFORM ...
```

Finding out what key the user pressed

There is another technique you may wish to use for processing input from a terminal. 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.

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. You can write programs without using even this one, but it contains information that can be very useful and is worth knowing about.

CICS translator generates, in the Linkage Section, field named DFHIBLK, the *Execute Interface Block* (EIB). 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.

```
01  DFHEIBLK.
    02  EIBTIME       PIC S9(7) COMP-3.
    02  EIBDATE       PIC S9(7) COMP-3.
    02  EIBTRNID      PIC X(4).
    02  EIBTASKN      PIC S9(7) COMP-3.
    02  EIBTRMID      PIC X(4).
    02  EIBCPOSN      PIC S9(4) COMP.
    02  EIBCALEN      PIC S9(4) COMP.
    02  EIBAID        PIC X.
    ....   .....
```

EIBAID The attention identifier (AID), tells you which keyboard key was used to transmit the last input. This field is one byte long ("PIC X(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, or from a previous transaction in a pseudoconversational sequence. It is in halfword binary form
(PIC S9(4) COMP).

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 is 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+".

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.

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)").

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 17 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 17. 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 copy book, any values above that appear to be spaces are not; they correspond to bit patterns for which there is no printable character.

MAPFAIL errors on BMS commands

MAPFAIL occurs on a RECEIVE MAP command when there is no data to be transfered to your program from the screen. 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). It can also occur if the user presses the ENTER key or one of the program function keys without keying any data into the screen. 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.

SDF (Screen Definition Facility)

In addition to creating maps through BMS, IBM also provides a software package called "Screen Definition Facility" or "SDF". It allows you to 'paint' your map at a terminal in the same Way you would like the user to see it. The constants and variables are bracketed by special characters, and you are even allowed to preview your map before releasing it for generation.

Getting started in SDF

To use SDF, you must first be logged on to CICS,. clear the screen, type "SDF" and press Enter. If your installation has SDF, you should see a sign-on screen, that will ask for an ID and a password. After you enter your ID and password, you will be confronted with an array of menus that at first do not appear to be very user friendly.

This diagram may help you navigate through SDF.

```
Initial Menu
Menu 1 Map Editor
          submenu    1. Map Identification
                     2. Map Characteristics
                     3. Field Definition
                     4. Field Attribute Definition
                     5. Field Initialization (usually skipped)
                     6. Application Structure Specifications (usually skipped)
                     7. application Structure Review
                     8. Test

Menu 4  CICS/BMS Generator
       submenus (unnumbered)
          CICS/BMS Generator (two screens)
          Specification Selection
          Member List
```

Everything begins with the initial menu. This is the only menu that does not have a number in the top left corner of the screen. Most of the time you will need only menu 1 or menu 4 from the main menu.

Menu 1 is used to create, modify, and test maps and fields. Menu 4 is used to physically generate the maps.

Menu 1: The Map Editor

Whenever you need to define a new map, you must go through most of the submenus of the Map Editor. But if you only want to modify an existing map, you can skip most of the steps. Menu 1.1,

map identification, will ask for the name of your mapset, map, and terminal device. Menu 1.2, map characteristics, will show the defaults for your installation. Menu 1.3, field definition, is where you "paint" your map on the screen. Menu 1,4, field attribute definition, can be used to change attribute definitions--for example, changing unprotected fields to protected. Menus 1.5 and 1.6 can usually be skipped. Menu 1.7, application structure review, is where you assign COBOL field names to the variables you defined back on menu 1.3. Menu 1.8 is used to test your map before generating it.

Menu 4: The Map Generator

SDF will even generate the JCL necessary to create a map and mapset for you, but it needs some information from you, such as your name and account number, before it can submit a batch job. SDF also needs to know which map and mapset you want to generate and which libraries to use.

Creating a map with SDF

The first thing you need to do is to choose option 1 from the initial menu. You should now see another screen, which will ask for the name of your mapset, map, device type, password, skeleton mapset, and skeleton map. If you are creating a new mapset, you should key in all the requested information (although many programmers will skip the password protection for the map). If you wanted to modify an existing mapset, you would only need to enter the mapset and map names.

The mapset name must be a legal name that has been defined by the systems programmer in the PPT. The map name can be any name you wish as long as it meets the syntax requirements. The device type is the type of terminal that will be used to display the map.

Many installations that use SDF will have a skeleton map and mapset defined to take care of many of the technical details needed for data transmission. You can view these default options by pressing the Enter key. Pressing the Enter key will save the mapset and map name. It will also show you the next screen, which is the default specification for the skeleton. If you need to change any of the defaults, you may do so by typing the new option over the old option. Pressing Enter will save the changes for the mapset you are defining. Pressing PF3 will exit without changes. For example, if the default is to lock up the terminal after every SEND command, you can change the FREEKB option NO to YES.

The next step would be to "paint the screen". If you are using a skeleton map, you can clear the display by typing in "DELETE 24" at the command line. This will delete all the lines of the copy of the skeleton map that appears on your screen. If you do not have a skeleton map, the map-painting area will be blank.

Painting a screen

This is the easiest part of SDF. If you want a constant to appear in the middle of the screen, just move the cursor to the middle of the screen and type in the constant. Be sure to bracket every field with special characters. if you want another constant on another line, reposition the cursor, type in the constant, and bracket the filed with special characters. To create a variable, position the cursor

at the desired location and sue the special characters for variables to show where the field begins and ends.

The difficult part of this process is knowing which characters to use as brackets for constants and variables. SDF will allow you to use predefined characters or to define your own. To see which special character has which attributes, press the PF5 key. This will show a screen with the special characters currently in use. You may also add your own special characters and attributes at the end of the list.

Defining Constants

Suppose you wish to create the following constants:

```
ABC INC
ENTER ID
```

You would do this by positioning the cursor a the desired location and then typing the # character before and after the constants. For example:

```
#ABC INC#
#ENTER ID#
```

If you forget to balance the # at the beginning of a constant with a matching # at the end of a constant, SDF will assume that all the space between the first # and the next field is one large constant field.

Defining Variables

The easiest way to define a variable is to position the cursor and then bracket the field. On many, but not all, terminals, the ⌈ symbol can be used to indicate the beginning of a variable. The # character can be used to denote the end of the variable. Thus, a five-position variable would appear on the paint map screen as:

```
⌈     #
```

Combining Constants and Variables

A variable without a label would be meaningless. Providing a label for a variable would appear as follows:

```
#ENTER CUSTOMER NAME# ⌈              #
```

As with constants, it is important to indicate where the variable ends. Otherwise, SDF will assume that all the space from the beginning of the variable to the beginning of the next legal field is part of the variable.

Defining Fields

Once the map has been painted, press PF3 to save, and exit. Then choose the next menu, which will show the map variables as they will appear in the COBOL COPY member. Remember, only the variables will generate field names in the COBOL COPY member.

SDF will automatically convert the length of the variable on the screen into a generated field length. You still need to choose names for the variables. A variable name may be up to 15 characters long. Once the map is generated, an extra character will be attached to the end of each name to indicate length, attribute, or data. SDF also allows you to explicitly define fields as alphanumberic, numeric, or numeric edited, if desired. The default is alphanumeric.

If for some reason the fields are defined incorrectly, you may always return to the field definition screen and repaint the field. For example, if an address appeared to be 20 bytes long on the screen but the application structure screen, enter the field description screen, and move one of the brackets over one position. This will automatically cause the resulting field description to change form a length of 19 to a length of 20 characters.

Viewing the Map

Once you are satisfied with the field definitions, it is a good idea, although technically not required, to test the map before SDF generates the corresponding code. This is done by exiting from whatever menu screen you are currently on and choosing option 8, Test, on the Map Editor step selection menu.

Generating the Map

Once you are satisfied with the map's appearance, you may generate the map code. This is done by choosing option 4 on the initial selection menu. At this point you should see a screen that asks for the name of your mapset, map, and other information.

The choices at your installation will differ. After overstepping the necessary information, press Enter. A new screen will appear. Press Enter again. If you did everything correctly, you should see a message that says:

```
JOB HAS BEEN CREATED.
THE FOLLOWING MAPS WERE INCLUDED.
```

This means that both the physical and the symbolic masts have been generated and the symbolic mapset should be waiting for your COBOL program as a COPY member. When you write the COBOL program, remember to use the COPY command to bring the symbolic map into your program. The name of the COPY member will be whatever you named your mapset when you used SDF.

Review Questions

1. What is a BMS map?
2. What are the two kinds of BMS maps? Describe each of them.
3. Why do we need two kinds of maps?
4. What is the maximum length of a mapname or a mapsetname or a fieldname?
5. What are macros DFHMSD, DFHMDI, and DFHMDF used for?
6. What are TYPE, CTRL, LANG, MODE, TERM, TIOAPFX options for? Write some of its possible values.
7. What is storage parameter used for? Give application scenarios under which you will choose to use different values of this parameter.
8. What are SIZE, LINE, COLUMN options for? Write some of its possible values.
9. What are POS, LENGTH, ATTRIB options for? Write some of its possible values.
10. Give different cases under which you will use PROT/UNPROT/ASKIP, BRT/NORM/DRK parameters.
11. What is a stopper field? Define one with proper attributes, size etc.
12. What is MDT? What is it used for?
13. What are FSET and FRSET? How is it used?
14. How do we send a map?
15. How do we receive a map?
16. What are the three symbolic map subfields generated for each variable field? What are their PIC clauses?
17. If selection field is given length 2, what will be its subfields' total length (including attribute, length) in symbolic map?
18. How do we change an attribute of a field through the program?
19. How do we position the cursor on a field when we send a map? What if we tried positioning cursor on multiple fields?
20. What are DATAONLY and MAPONLY options used for? Explain when will you choose to use each of them.
21. What are the three ways to detect which AID key was pressed?
22. What is EIB? Who/What puts DFHEIBLK into the program and at what stage?
23. What is MAPFAIL? When does it occur?
24. When you compile a CICS program, the (pre)compiler puts an extra chunk of code. Where does it get included and what is it called? What is its length?
25. Name some important fields in the EIB block ?

Assignments

1. Create BMS macros for the maps give below and generate physical and symbolic maps.

```
          1         2         3         4         5         6         7         8
 12345678901234567890123456789012345678901234567890123456789012345678901234567890
1+EMPMNT                    +EMPLOYEE MAINTAINANCE              +DATE:+MM/DD/YY
2                                                               +TIME:+HH:MM:SS
3
4       +READ/ADD/DELETE/MODIFY:+_+(R/A/D/M)
5
6       +(DEFAULT IS R. ONLY EMPNO IS TO BE FILLED FOR CHOICES R,D, AND M)
7
8       +-----------------------------------------------------------------
9
10           +EMPNO:+____|
11
12         +EMPNAME:+_______________|
13
14      +EMPADDRESS:+_____________________________|
15
16
17
18
19
20
21+_____________________________(msg field)_____________________
22
23 +PF1-HELP    +PF3/CLEAR-EXIT  +PF4-MAIN MENU   +PF5-REFRESH   +ENTER-PROCESS
24
```

```
          1         2         3         4         5         6         7         8
 12345678901234567890123456789012345678901234567890123456789012345678901234567890
1+EMPLIST                   +EMPLOYEE DETAILS BROWSE            +DATE:+MM/DD/YY
2                                                               +TIME:+HH:MM:SS
3                           +EMPNO:+____|
4
5+EMPNO   +EMPLOYEE NAME    +EMPLOYEE ADDRESS
6+        +                 +
7+        +                 +
8+        +                 +
9+        +                 +
10+       +                 +
11+       +                 +
12+       +                 +
13+       +                 +
14+       +                 +
15+       +                 +
16+       +                 +
17+       +                 +
18+       +                 +
19+       +                 +
20
21+_____________________________(msg field)_____________________
22
```

```
23 +PF1-HELP    +PF3/CLEAR-EXIT  +PF4-MAIN MENU   +PF5-REFRESH
24 +PF7-PAGEUP  +PF8-PAGEDOWN    +ENTER-PROCESS
```

3. Application Programming

In this unit you will learn:

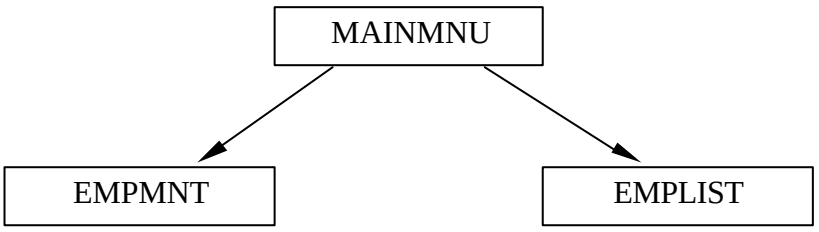
- what is reenterent code
- to write CICS programs using COBOL
- the COBOL verbs not to be used in a CICS program
- the use of EIB and COMMAREA
- to take care of exeptional conditions using HANDLE CONDITION and RESP
- to take care of ABENDs
- how to access system information briefly
- what are different types of CICS programs
- how to create a run unit

Reenterent code

A load module (or executable) is reenterent if it does not modify itself. That means, the code can be executed any number of times without any change of the way program behave. Every CICS program must be reenterent.

Writing CICS programs in COBOL

Let us start designing the employee application which we discussed in the previous two units. The following is the program flow diagram of the whole application.



We have decided to make programs MAINMNU and EMPMNT into single CICS transaction called 'MENU'. Program EMPLIST will be a separate transaction 'LIST'. In this unit we will develop the first program MAINMNU. The other two programs will be developed in the subsequent units.

COBOL verbs not to be used in a CICS program

In a CICS program you are not allowed to use any of those commands which give control directly to MVS. Doing so may create unpredictable results.

The following commands should not be used:

ACCEPT, CLOSE, DELETE, DISPLAY, OPEN, READ, REPORT WRITER commands, REWRITE, SORT, WRITE, START and never use STOP RUN & GO BACK.

If you use STOP RUN, the control goes directly to MVS. Then MVS will terminate CICS unconditionally and free that address space.

EXEC interface stubs

Each application program you write must contain an interface to CICS. This takes the form of an EXEC interface stub, used by the CICS high level programming interface. The stub must be link-edited with your application program to provide communication between your code and the EXEC interface program (DFHEIP).

There are stubs for each programming language. When we have EXEC command in COBOL, each EXEC command is translated into a COBOL CALL statement by the command translator.

Execute Interface Block (EIB)

Recall that we discussed EIB in the previous unit. As we develop programs we will be using several fields from EIB.

COMMAREA

Every CICS program must have a linkage section with DFHCOMMAREA inside it as follows:

```
LINKAGE SECTION.
01 DFHCOMMAREA                    PIC X(n).
```

Whenever a task is ended, all the working-storage variables associated with it gets destroyed. But the DFHCOMMAREA is retained until end of transaction. So, whenever we are doing a pseudo-conversational programming, we must remember to transfer all the data that we might want to retain across tasks to COMMAREA. The maximum size of DFHCOMMAREA is 32K.

Let us see how to use the COMMAREA in our MAINMNU program. The program is expected to do the following:

1. Send the MAINMNU map from mapset MENUSET.
2. End the task (since we want to have a pseudo-conversation).

----waiting for user response----

3. Receive the map (when the user presses an AID key after the input, CICS automatically starts the transaction by creating a new task)
4. validate the selection
5. if selection is 1, transfer control to EMPMNT program
   else if selection is 2, start the LIST transaction
   else move appropriate message to the message field and go to step 1.

Let us look at the pseudo-conversational execution of this program in little more detail. We end the task in step2 because while user is providing the input, we do not want the task to be active which results in more resource usage. But when we end the task, we actually were not through with our transaction. That means, we want CICS to invoke the same transaction when the user finishes the input (note that the user presses any of the AID keys to indicate the end of data entry).

The conditional termination of the transaction is achieved using the following command:

```
EXEC CICS RETURN
          TRANSID ('MENU')
END-EXEC.
```

On RETURN with TRANSID (for this we will subsequently use the term conditional return) the control goes to CICS. At that time no transaction is associated with the terminal, and the user can enter the data on the screen. After entering the data whenever the user presses an AID key, CICS

locks the keyboard, transfers the map data to CICS memory, and starts the transaction MENU by creating a new task. The MAINMNU program starts executing from the beginning (it will not come back to where we terminated conditionally!) and whenever the program executes a RECEIVE MAP command the data which is in CICS buffer is transfered to the symbolic map working-storage area.

The fact that the program starts executing afresh makes the pseudo-conversational programming different. Here, when the program restarts all the working storage variables are reallocated. That means we loose all the data which were populated during the last execution. To retain data across tasks we need to pass all the data that we need to the commarea of the returning transaction.

Let us look at an example. In the EMPLST program if we want to retain the input data, we will do as follows:

```
* Define the variables which we want to pass from one task to other
  01 COMMAREA-DATA.
     05 CA-EMPNO              PIC X(4).
     05 CA-EMPNAME            PIC X(15).

* Define DFHCOMMAREA for storing the above variables in the linkage section
  LINKAGE SECTION.
  01 DFHCOMMAREA              PIC X(19).
```

We populate these variables in the program. Then, when we issue a conditional return, we will use the commarea option as follows:

```
EXEC CICS RETURN
          TRANSID ('LIST')
          COMMAREA(COMMAREA-DATA)
          LENGTH  (19)
END-EXEC.
```

User then types the input and presses an AID key. When CICS starts the LIST transaction, the COMMAREA-DATA will be passed on to its DFHCOMMAREA. In the program, the following line can be executed to get the data back from DFHCOMMAREA to the working storage.

```
MOVE DFHCOMMAREA        TO COMMAREA-DATA.
```

Handling errors and exeptional conditions

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

- Conditions that are not normal from CICS's point of view but that are expected in the program.

  For example, when we issue a file read to read a particular record, if the record is not present we get the "not found" response. "mapfail" is a similar error. Errors in this category should be handled by explicit logic in the program.

- 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. For example, “division by zero”, “illegal character in numeric field”, “transaction id error”, “file not found”, etc. are errors of this kind. If the program does not handle these, CICS will ABEND the program. 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.

- 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 previous. Systems or operations personnel will still have to analyze the problem and fix it.

CICS provides mechanisms to detect errors and execptional conditions in CICS commands and handle them in your program. One of them is by using HANDLE CONDITION in your program and the other by using RESP option. These two methods detect errors only in CICS commands. Errors like “illegal character in numeric field” will cause CICS to terminate your program. These kinds of ABENDS also can be handled through the program using HANDLE ABEND command.

The HANDLE CONDITION command look like this:

```
EXEC CICS HANDLE CONDITION
      MAPFAIL (Y000-MAPFAIL-ERROR)
      PGMIDERR(Y100-PROGRAM-NOTFOUND-ERROR)
      NOTFND  (Y200-RECORD-NOTFOUND-ERROR)
      ERROR   (Y300-CATHALL-ERROR)
END-EXEC.
```

The above command can be written in the beginning of your program and whenever a MAPFAIL occurs, CICS automatically issues a

```
GOTO        Y000-MAPFAIL-ERROR.
```

This command is a “sticky” command. That means, once we define this command, it will be valid for every CICS command that follows this. After some time, if you want a different routine for mapfail, you need to redefine the HANDLE CONDITION. For any other errors other than “mapfail”, “pgmiderr”, and “notfnd”, CICS will execute the catch-all routine.
The second method is to use RESP option in all the CICS commands for which you expect an error or an exception. For example, you may expect a “mapfail” while receiving a map. To trap that code RESP option in RECEIVE command as follows:

```
EXEC CICS RECEIVE
          MAP   ('MENUMAP')
          MAPSET('MENUSET')
          RESP  (RSP-CODE)
END-EXEC.
```

```
IF RSP-CODE = 36 THEN PERFORM Y000-MAPFAIL-ERROR
ELSE IF RSP-CODE NOT= 0 THEN PERFORM Y300-CATCHALL-ERROR
     ELSE CONTINUE
     END-IF
END-IF.
```

Here RSP-CODE is a working storage variable which is defined in your program as follows:

```
01  RSP-CODE                PIC S9(9)   VALUE ZERO COMP.
```

The values corresponding to the response codes can be obtained from the system manual. The value 36 stands for MAPFAIL, value 0 stands for no error, and so on. But, normally you need not remember these values. We could use a copybook as shown below:

```
    01  RSP-CODE                PIC S9(9)     VALUE ZERO COMP SYNC.
         88  RSP-ALLOCERR                     VALUE 85.
         88  RSP-DISABLED                     VALUE 84.
         88  RSP-DUPKEY                       VALUE 15.
         88  RSP-DUPREC                       VALUE 14.
         88  RSP-END                          VALUE 83.
         88  RSP-ENDFILE                      VALUE 20.
         88  RSP-INVREQ                       VALUE 16.
         88  RSP-IOERR                        VALUE 17.
         88  RSP-ITEMERR                      VALUE 26.
         88  RSP-LENGERR                      VALUE 22.
         88  RSP-MAPFAIL                      VALUE 36.
         88  RSP-NORMAL                       VALUE ZERO.
         88  RSP-NOSPACE                      VALUE 18.
         88  RSP-NOTALLOC                     VALUE 61.
         88  RSP-NOTAUTH                      VALUE 70.
         88  RSP-NOTFND                       VALUE 13.
         88  RSP-NOTOPEN                      VALUE 19.
         88  RSP-OVERFLOW                     VALUE 40.
         88  RSP-PGMIDERR                     VALUE 27.
         88  RSP-QBUSY                        VALUE 25.
         88  RSP-QIDERR                       VALUE 44.
         88  RSP-QZERO                        VALUE 23.
         88  RSP-ROLLEDBACK                   VALUE 82.
         88  RSP-SYSIDERR                     VALUE 53.
         88  RSP-TERMIDERR                    VALUE 11.
         88  RSP-TRANSIDERR                   VALUE 28.
         88  RSP-FILE-UNAVAILABLE             VALUE 19, 53, 84.
```

Then we could use **`IF RSP-MAPFAIL ...`** and so on.

We can define HANDLE CONDITION and then use RESP in all commands with a NOHANDLE option. This will disable the previous handle condition and allow the response code to be used.

ABEND

An ABEND occurs when an unhandled error occurs in your program. In the event of an ABEND you may wish to transfer control to a general abend handling routine which displays some meaningful

messages and terminates. This can be achieved by HANDLE ABEND command which is as follows:

```
EXEC CICS HANDLE ABEND
          LABEL (Z999-ABEND-ROUTINE)
END-EXEC.
```

We could also transfer control to another error handling program by

```
EXEC CICS HANDLE ABEND
          PROGRAM ('ERRHANDL')
END-EXEC.
```

In the error handling paragraph we could use a forcible abend after moving proper abend code which results in a dump. The command to force an ABEND is

```
MOVE 'DB01'       TO ABEND-CODE.
EXEC CICS ABEND
          ABCODE  (ABEND-CODE)
END-EXEC.
```

where ABEND-CODE is an X(4) variable defined in the working storage to which any 4 letter code can be moved. The code should not start with letter 'A' because CICS abend codes always start with 'A'.

Creating a run unit

The CICS Translator

Because the compilers (and assemblers) cannot process CICS commands directly, an additional step is needed to convert your program into executable code. This step is called translation, and consists of converting CICS commands into the language in which the rest of the program is coded, so that the compiler (or assembler) can understand them.

CICS provides a translator program for each of the languages in which you may write, to handle both EXEC CICS and EXEC DLI statements. There are three steps: translation, compilation (assembly), and link-edit. The following Figure shows the process.

Note: If you use EXEC SQL, you need additional steps to translate the SQL statements and bind.

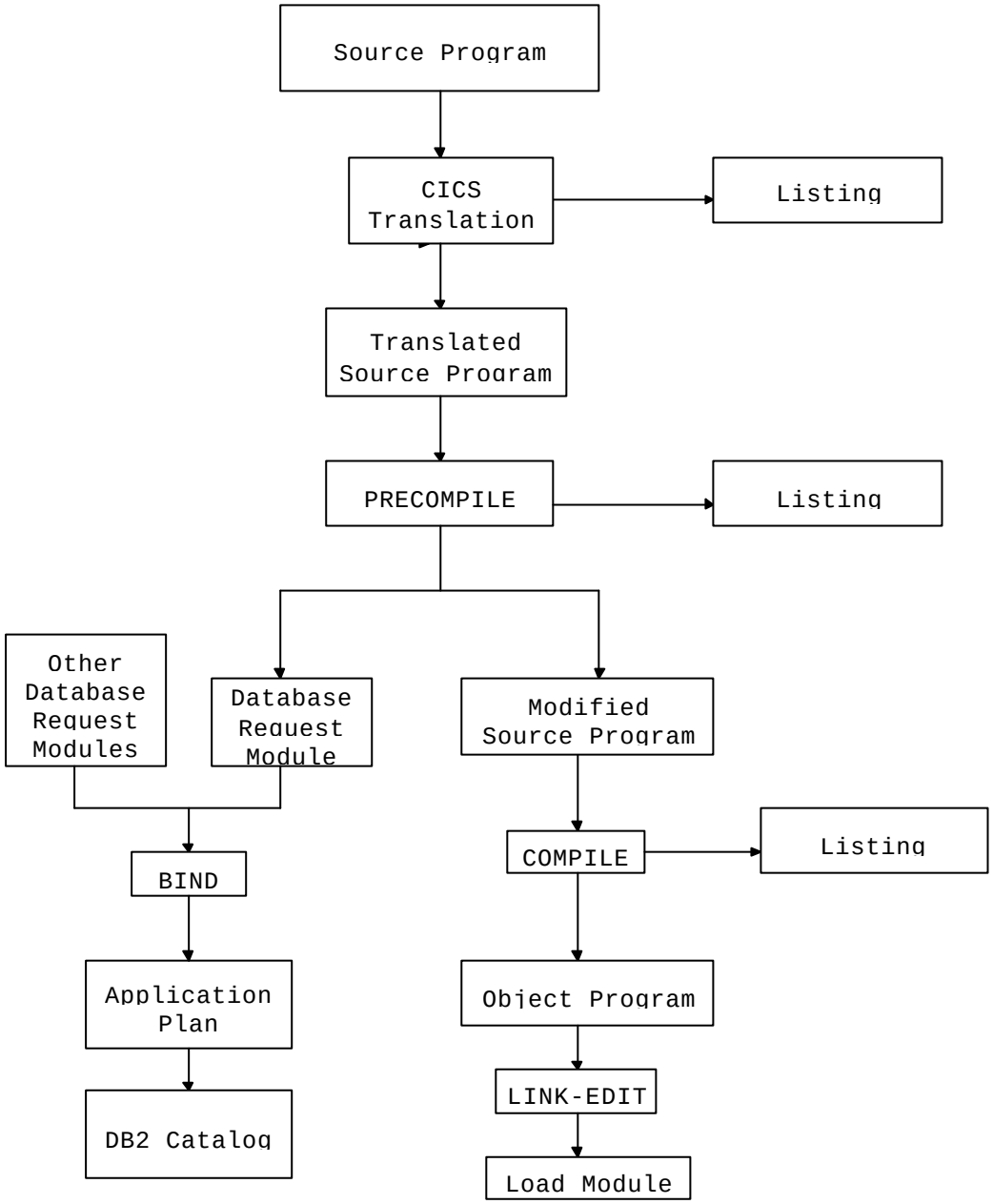


Figure 18. Preparing an application program

There are a number of options that you can specify for the translation process, and you may need to do this for certain types of programs.

The COBOL compiler reads the translated version of your program as input, rather than your original source. This affects what you see on your compiler listing. It also means that COPY statements in your source code must not bring in untranslated CICS commands, because it will be too late for the translator to convert them. (Copying in pretranslated code is possible, but not recommended; it may produce unpredictable results.)

EXEC commands are translated into CALL statements that invoke CICS interface modules. These modules get incorporated into your object module in the link-edit step, and you will see them in your link-edit output listing.

A language translator reads your source program and creates a new one; normal language statements remain unchanged, but CICS commands are translated into CALL statements of the form required by the language in which you are coding. The calls invoke CICS-provided "EXEC" interface modules, which later get link-edited into your load module, and these in turn invoke the requested services at execution time. The CICS commands that get translated still appear in the source, but as comments only. The non-CICS statements are unchanged.

The translators for all of the languages use one input and two output files:

- SYSIN (translator input) is the file that contains your source program.
- SYSPUNCH (translated source) is the translated version of your source code, which becomes the input to the compile (assemble) step. In this file, your source has been changed as follows:

  The EXEC interface block (EIB) structure has been inserted.

  EXEC CICS and EXEC DLI commands have been turned into CALL statements.

  CICS DFHRESP and DFHVALUE built-in functions have been processed.

  A data interchange block (DIB) structure and initialization call have been inserted if the program contains EXEC DLI statements.
- SYSPRINT (translator listing) shows the number of messages produced by the translator, and the highest severity code associated with any message. The options used in translating your program also appear, unless these have been suppressed with the NOOPTIONS option.

  For COBOL, C/370, and PL/I programs, SYSPRINT also contains the messages themselves. In addition, if you specify the SOURCE option of the translator, you will also get an annotated listing of the source in SYSPRINT. This listing contains almost the same information as the subsequent compilation listing, and therefore many installations elect to omit it (the NOSOURCE option). One item you may need from this listing that is not present in the compile listing, however, is the line numbers, if the translator is assigning them. Line numbers are one way to indicate points in the code when you debug with the execution diagnostic facility (EDF).

Translator options

Table 1. Translator options applicable to programming language

Translator option	VS COBOL II	OS/VS COBOL	C/370	PL/I	Assembler
ANSI85	X				

APOST|QUOTE	X	X				
CICS	X	X	X	X	X	
COBOL2	X					
DBCS	X					
DEBUG|NODEBUG	X	X	X	X		
DLI	X	X	X	X	X	
EDF|NOEDF	X	X	X	X	X	
EPILOG|NOEPILOG					X	
FE|NOFE	X	X	X	X	X	
FEPI|NOFEPI	X	X	X	X	X	
FLAG[(I|W|E|S)]	X	X	X	X		
GDS			X		X	
GRAPHIC				X		
LANGLVL(1)|LANGLVL(2)	X	X				
LINECOUNT(n)	X	X	X	X	X	
MARGINS(m,n)			X	X		
NATLANG	X	X	X	X	X	
NUM|NONUM	X	X				
OPMARGINS(m,n[,c])			X	X		
OPSEQUENCE(m,n)| NOOPSEQUENCE			X	X		
OPT|NOOPT	X	X				
OPTIONS|NOOPTIONS	X	X	X	X	X	
PROLOG|NOPROLOG					X	
QUOTE|APOST	X	X				
SEQ|NOSEQ	X	X				
SEQUENCE(m,n)|NOSEQUENCE			X	X		
SOURCE|NOSOURCE	X	X	X	X		
SP	X	X	X	X	X	

SPACE(1|2|3)	X	X			
SPIE|NOSPIE	X	X	X	X	X
SYSEIB	X	X	X	X	X
VBREF|NOVBREF	X	X	X	X	

4. Control Operations

In this unit you will learn:

- about task control - ENQUEUE, DEQUEUE, SUSPEND
- about program control - LINK, XCTL, RETURN, LOAD, RELEASE
- about storage control - GETMAIN, FREEMAIN
- about temorary storage control - WRITEQ, READQ, DELETEQ
- about transient data control - Intrapartition & extrapartition TDQ, its commands
- about interval control - ASKTIME, FORMATTIME, START, & others
- about terminal control - SEND, RECEIVE

This part of the material collects together several groups of operations that are not specifically database or data communication operations, but that control the execution of tasks within a CICS system. These groups of operations are as follows:

- Task control - comprising functions to temporarily relinquish control or to synchronize resource usage.
- Program control - comprising functions affecting the flow of control between application programs.
- Storage control - comprising functions to obtain and release areas of main storage.
- Temporary storage control - comprising functions for the temporary storage of data.
- Transient data control - comprising functions for the transfer of data between CICS tasks and between the CICS region and other regions.
- Interval control - comprising functions whose execution is dependent on time.
- Terminal control - comprising functions which provides communication between user-written application programs and terminals

Task control

The CICS task control program provides functions that synchronize task activity, or that control the use of resources.

CICS processes tasks according to priorities assigned by the system programmer. Control of the processor is given to the highest priority task that is ready to be processed and is returned to the operating system when no further work can be done by CICS or by user-written application programs.

Task control commands are provided to:

- Suspend a task (SUSPEND).
- Schedule the use of a resource by a task (ENQ and DEQ).

Suspend a task (SUSPEND)

A task can issue the SUSPEND command to relinquish control and allow tasks higher on the active chain to proceed. This facility can be used to prevent processor-intensive tasks from monopolizing the processor. As soon as no other task higher on the active chain is waiting to be processed, control is returned to the issuing task; that is, the task remains dispatchable.

This command is used to relinquish control to a task of higher dispatching priority. Control is returned to the task issuing the command as soon as no other task of a higher priority is ready to be processed.

```
+-------------------------------------------------------------------+
|                                                                   |
|   SUSPEND                                                         |
|                                                                   |
+-------------------------------------------------------------------+
```

Schedule use of a resource by a task (ENQ and DEQ)

Scheduling the use of a resource by a task is sometimes useful in order to protect the resource from concurrent use by more than one task, that is, to make the resource serially reusable. Each task that is to use the resource issues an ENQ (enqueue) command. The first task to do so has the use of the resource immediately, but subsequent ENQ commands for the resource, issued by other tasks, result in those tasks being suspended until the resource is available. Each task using the resource should issue a DEQ (dequeue) command when it has finished with it. The resource then becomes available and the next task to have issued an ENQ command is resumed and given use of the resource. The other tasks obtain the resource in turn, in the order in which they enqueued upon it.

```
+-------------------------------------------------------------------+
|                                                                   |
|   ENQ                                                             |
|   RESOURCE(data-area)                                             |
|   [LENGTH(data-value)]                                            |
|   [NOSUSPEND]                                                     |
|                                                                   |
|   Conditions: ENQBUSY, LENGERR                                    |
|                                                                   |
+-------------------------------------------------------------------+
```

```
+-------------------------------------------------------------------+
|                                                                   |
|   DEQ                                                             |
|   RESOURCE(data-area)                                             |
|   [LENGTH(data-value)]                                            |
|                                                                   |
|   Conditions: LENGERR                                             |
|                                                                   |
+-------------------------------------------------------------------+
```

The ENQ and DEQ commands can be used to enqueue upon and dequeue from a resource that is to be protected from concurrent use by more than one task.

The ENQ command causes further execution of the task issuing the ENQ command to be synchronized with the availability of the specified resource; control is returned to the task when the resource is available.

The ENQBUSY condition allows a conditional ENQ to be used. If a resource is not available when enqueued, the ENQBUSY condition is raised. The execution of a HANDLE CONDITION ENQBUSY command will return control to the task at the ENQBUSY label, without waiting for the resource to become available.

The DEQ command causes a resource currently enqueued upon by the task to be released for use by other tasks. If a task enqueues upon a resource but does not dequeue from it, CICS automatically releases the resource during sync point processing or when the task is terminated.

If more than one ENQ command is issued for the same resource by a given task, the resource remains owned by that task until the task issues a matching number of DEQ commands.

Program control

As we explained earlier, a transaction (even task) may execute several programs in the course of completing its work.

Associating programs and transactions

The installed program definition contains one entry for every program used by any application in the CICS system in PPT. 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 in PCT. 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 PCT.

3. This tells CICS which program to invoke first.

4. CICS looks up this program in PPT, finds out where it is, and loads it if it is not 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.

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.

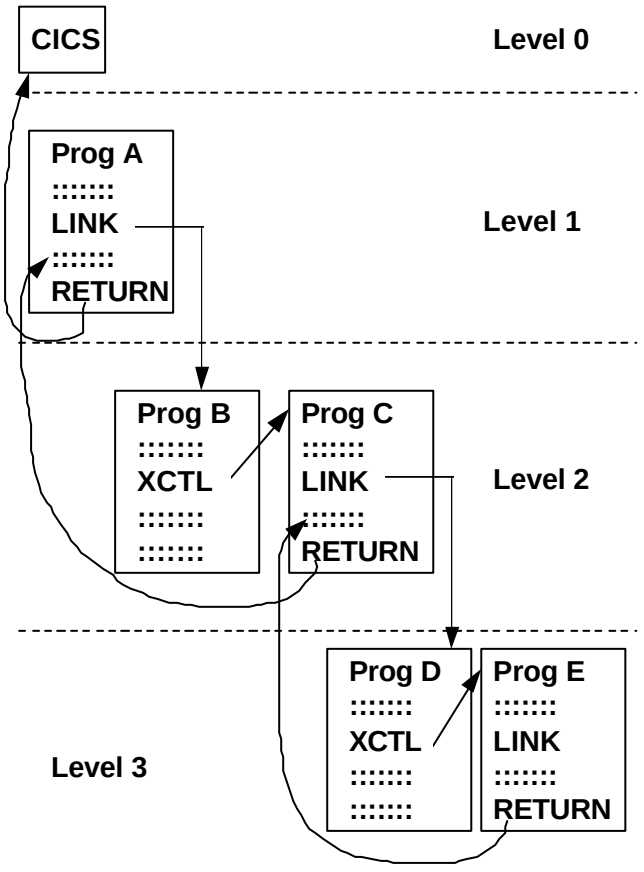


Figure 19. Transferring control between programs (normal returns)

In contrast, when one program transfers control to another using XCTL, 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. See Figure 19.

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 do not use it. Like the length parameter in other commands, it must be a halfword binary value.

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.                                                           |
|                                                                       |
```

```
+------------------------------------------------------------------+
```

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 is 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 is 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.

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. There are some observations that has to be made. For example:

- A CALLed program remains in its last-used state after it returns 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 programs 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.

Always use XCTL if it will do, of course, rather than LINK. That is just a program logic issue; you either need control back or you do not. In our example, as you will see, we have broken our own rule and used a LINK (rather than an XCTL) to the error-handling program.

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.

Another main issue is that a LINKed program can not be pseudo-conversational where as an XCTLed program can be.

Passing control and data between programs and transactions

Now that we have 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 us code a few program control commands for the example application.

In EMPMNT, when we need to do a numeric validation for EMPNO, we transfer control to the general-purpose numeric validation program, NUMCHK. We pass two items of information to NUMCHK:

1. The EMPNO value
2. The return flag which indicates whether the passed variable is numeric or not.

The following is EMPMNT's working storage which is passed onto NUMCHK:

```
01 COMMAREA-FOR-NUMCHK.
   05 NUMCHK-EMPNO          PIC X(4).
   05 NUMCHK-EMPNO-OK       PIC X.
```

Before transfering control to NUMCHK program we should move the employee number which is received from EMPMNT map and move value 'Y' to the flag. If the employee number is not numeric then the NUMCHK program will set the flag to 'N' and return control:

Here is the code in EMPMNT to pass control to NUMCHK is:

```
 MOVE EMPNOMI     TO NUMCHK-EMPNO
 MOVE 'Y'         TO NUMCHK-EMPNO-OK
* here empnomi is the dsect variable of empno field
* now we will link to the numchk routine

 EXEC CICS LINK
          PROGRAM('NUMCHK')
          COMMAREA(COMMAREA-FOR-NUMCHK)
          LENGTH(5)
 END-EXEC.
```

The program receiving control, NUMCHK in this case, defines this same area in its Linkage Section, as shown below.

```
LINKAGE SECTION.
01 DFHCOMMAREA.
   05 NUMCHKRTN-EMPNO         PIC X(4).
   05 NUMCHKRTN-EMPNO-OK      PIC X.
```

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, validate the employee number, move 'N' to NUMCHKRTN-EMPNO-OK in case of error, and then RETURN to the next statement in EMPMNT program. In case of error, after returning, you can see that NUMCHK-EMPNO-OK variable of EMPMNT program will contain 'N'.

Communicating between transactions in the example application

There are several different types of return to CICS. The simplest occurs in program MAINMNU, after the user has indicated a wish to exit from the application by pressing PF3 or CLEAR key. 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 all the programs, in contrast, we need to end the transaction temporarily after sending the map. In MAINMNU program, after sending the map, the RETURN command is written:

```
EXEC CICS RETURN
          TRANSID('MENU')
END-EXEC.
```

But if we want to retain some data across pseudo-conversational invocations of the same program, we need to pass data to it as well. In our EMPMNT program we may want to store EMPNO, EMPNAME, and EMPADDRESS in the commarea. Then the communications area in Working-Storage looks like this:

```
01 COMMAREA-FOR-EMPMNT.
   05 CA-EMPNO                PIC S9(4) COMP.
```

```
   05 CA-EMPNAME              PIC X(15).
   05 CA-EMPADDREAA           PIC X(25).
```

And the code needed is:

```
EXEC CICS RETURN
          TRANSID('EMPMNT')
          COMMAREA(COMMAREA-FOR-EMPMNT)
          LENGTH(42)
END-EXEC.
```

When program EMPMNT is invoked after the pseudo-conversation, 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.

Abending a transaction

In addition to the normal return sequences that we have described, there is another command that you use in abnormal circumstances. This is the ABEND command. It returns control to CICS directly. 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.

Other program control commands

There are two other program control commands that we will 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 does not give it control. This is useful for tables of the type that are assembled and stored in a program library, but that do not contain executable code.

The **RELEASE** command tells CICS that you have finished using such a "program".

Storage control

The CICS storage control program controls requests for main storage to provide intermediate work areas and any other main storage not provided automatically by CICS but needed to process a transaction. You can initialize the acquired main storage to any bit configuration; for example, zeros or EBCDIC blanks.

Storage control commands are provided to:

- Get and initialize main storage (GETMAIN).
- Release main storage (FREEMAIN).

CICS releases all main storage associated with a task when the task is terminated normally or abnormally. This includes any storage acquired, and not subsequently released, by your application program. If there is insufficient main storage to satisfy a GETMAIN command, the NOSTG exceptional condition will occur.

Temporary Storage Control

The need for scratchpad and queuing facilities

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 will cover in a moment. Others are listed below.

- 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. The COMMAREA is the recommended scratchpad area.
- The Common Work Area (known as the CWA). Any transaction can access the CWA, and since there is 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 is not commonly used in command level programs.

CICS provides two queuing facilities: temporary storage and transient data.

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 is not 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.

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.

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.

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.

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.
```

Notice that you cannot delete individual items from a temporary storage queue; you have to delete the whole queue.

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 do not 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.

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 (we could use EIBTRMID). 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. In our EMPLIST program, we will define our queue name as follows:

```
01 EMP-QUEUE-ID.
```

```
05 EMP-TERMINAL-ID              PIC X(4).
05 EMP-QUEUE-NAME               PIC X(4) VALUE 'EMPL'.
```

We will subsequently move EIBTRMID to EMP-TERMINAL-ID before issuing any of the commands.

Using temporary storage in the example application

Let us see how we will use temporary storage in the example application for our scratchpad requirements. In program EMPLIST we will use TSQ as follows:

When we open the employee data file for browsing, we will read 42 records at a time and populate the TSQ. The we will show the first 14 records on the screen. Whenever the user presses PF8 for viewing next page, we will read from the TSQ instead of the file. But when the user moves beyond 3rd page we need to rebuild the TSQ for the next 42 records. (Note that this number '3 pages at a time' is not any special number. It is decided while designing the program. In our example, it just happened to be 3.)

Errors on temporary storage commands

You can experience six different types of error on the temporary storage commands that we have 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).

- **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 is not 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 do not, 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.

- **QIDERR** means that the queue that you have 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.

Transient Storage Control

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 every 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 add 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 Read 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 does not start until both the trigger level is reached and the terminal in question is available. This is known as Automatic Task Initiation (ATI).

- Transient data queues are always written to a file; there is no counterpart to the MAIN option that is used in temporary storage commands.

- 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 does not apply to extrapartition queues.

In the example application, we will not be using TDQs.

Interval Control

The CICS interval control program, in conjunction with a time-of-day clock maintained by CICS, provides functions that can be performed at the correct time; such functions are called time-controlled functions. The time of day is obtained from the operating system at intervals whose frequency, and thus the accuracy of the time-of-day clock, depends on the task mix and the frequency of task switching operations.

Using interval control commands you can:

- Request the current date and time of day (ASKTIME)
- Select the format of date and time (FORMATTIME)
- Delay the processing of a task (DELAY)
- Request notification when specified time has expired (POST)
- Wait for an event to occur (WAIT EVENT)
- Start a task and store data for the task (START)
- Retrieve data stored (by a START command) for a task (RETRIEVE)
- Cancel the effect of previous interval control commands (CANCEL).

Request identifiers

As a means of identifying the request and any data associated with it, a unique request identifier is assigned by CICS to each DELAY, POST, or START command. You can specify your own request identifier by means of the REQID option; if you do not, CICS assigns (for POST and START only) a unique request identifier and places it in field EIBREQID in the EXEC interface block (EIB). Specify a request identifier if you want the request to be canceled at some later time by a CANCEL command.

Request current date and time of day (ASKTIME)

```
+-----------------------------------------------------------------+
|                                                                 |
|   ASKTIME                                                       |
|   [ABSTIME(data-area)]                                          |
|                                                                 |
+-----------------------------------------------------------------+
```

You use this command to update the date and CICS time-of-day clock, and the fields EIBDATE and EIBTIME in the EIB. These two fields contain initially the date and time when the task started. The command returns the current time in the form of the number of milliseconds since 0000 hours on January 1, 1900.

Select the format of date and time (FORMATTIME)

```
FORMATTIME
ABSTIME(data-value)
[YYDDD(data-area)]
[YYMMDD(data-area)]
[YYDDMM(data-area)]
[DDMMYY(data-area)]
[MMDDYY(data-area)]
[DATE(data-area)]
[DATEFORM(data-area)]
[DATESEP[(data-area)]]
[YEAR(data-area)]
[TIME(data-area)
[TIMESEP[(data-area)]]
```

Start a task (START)

You use the START command to start a task, on a local or remote system, at a specified time. The starting task may pass data to the started task and may also specify a terminal to be used by the started task as its principal facility. The TRANSID, TERMID, and FROM options specify the transaction to be executed, the terminal to be used, and the data to be used, respectively.

The syntax of the command is as follows:

```
START
[INTERVAL(hhmmss)_|TIME(hhmmss)]
TRANSID(name)
[REQID(name)]
[FROM(data-area)
[LENGTH(data-value)]
[TERMID(name)]
[SYSID(name)]
[RTRANSID(name)]
[RTERMID(name)]
[QUEUE(name)]
```

Further data may be passed to the started task in the FROM, RTRANSID, RTERMID, and QUEUE options. For example, one task can start a second task passing it a transaction name and a terminal name to be used when the second task starts a third task; the first task may also pass the name of a queue to be accessed by the second task.

One or more constraints have to be satisfied before the transaction to be executed can be started, as follows:

1. The specified interval must have elapsed or the specified expiration time must have been reached. The INTERVAL option should be specified when a transaction is to be executed on a remote system; this avoids complications arising when the local and remote systems are in different time zones.

2. If the TERMID option is specified, the named terminal must be available.

Starting tasks without terminals

If the task to be started is not associated with a terminal, each START command results in a separate task being started. This happens regardless of whether or not data is passed to the started task.

The following example shows how to start a specified task, not associated with a terminal, in one hour:

```
EXEC CICS START
       TRANSID('LIST')
       INTERVAL(10000)
END-EXEC.
```

Starting tasks with terminals and data

Data is passed to a started task if one or more of the FROM, RTRANSID, RTERMID, and QUEUE options is specified. Such data is accessed by the started task through execution of a RETRIEVE command as described later in the topic.

It is possible to pass many data records to a new task by issuing several START commands, each specifying the same transaction and terminal.

Execution of the first START command ultimately causes the new task to be started and allows it to retrieve the data specified on the command. The new task is also able to retrieve data specified on subsequently executed START commands that expire before the new task is terminated.

The following example shows how to start a task associated with a terminal and pass data to it:

```
01 DATA-FOR-EMPLIST          PIC X(5) VALUE 'HELLO'.
```

```
EXEC CICS START
       TRANSID('LIST')
       TERMID(EIBTRMID)
       FROM(DATA-FOR-EMPLIST)
       LENGTH(5)
END-EXEC.
```

Retrieve data stored for a task (RETRIEVE)

```
+-------------------------------------------------------------------------+
¦   RETRIEVE                                                              ¦
¦   [INTO(data-area)|SET(ptr-ref)]                                        ¦
¦   [LENGTH(data-area)]                                                   ¦
¦   [RTRANSID(data-area)]                                                 ¦
¦   [RTERMID(data-area)]                                                  ¦
¦   [QUEUE(data-area)]                                                    ¦
¦                                                                         ¦
+-------------------------------------------------------------------------+
```

The following example shows how to retrieve data stored by a START command for the task, and store it in the user provided data area called DATA-FROM-MENU.

```
01 DATA-FROM-MENU             PIC X(5).
01 DATA-FROM-MENU-LEN         PIC S9(4) COMP.
EXEC CICS RETRIEVE
       INTO(DATA-FROM-MENU)
       LENGTH(DATA-FROM-MENU-LEN)
END-EXEC.
```

You use this command to retrieve data stored by expired START commands. It is the only method available for accessing such data.

The INTO option specifies the area into which the data is to be placed.

You must specify, in the LENGTH option, a data area that contains the maximum length of record that the application program will accept. If the record length exceeds the specified maximum, it is truncated and the LENGERR condition occurs. After the retrieval operation, the data area specified in the LENGTH option is set to the record length (before any truncation occurred).

A task that is not associated with a terminal can access only the single data record associated with the original START command; it does so by issuing a RETRIEVE command. The storage occupied by the data associated with the task is released upon execution of the RETRIEVE command, or upon termination of the task if no RETRIEVE command is executed prior to termination. A task that is associated with a terminal can access all data records associated with all expired START commands having the same transaction identifier and terminal identifier as the START command that started the task; it does so by issuing consecutive RETRIEVE commands.

Errors on the START and RETRIEVE commands

A number of different problems may arise in connection with the START and RETRIEVE commands that we have described.

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 (Terminal Control Table).

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.

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 different 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 POST command, through the use of the CANCEL command.

- Assign a name to the data to be passed from the originating task to the started task, through the use of the REQID option on the START and RETRIEVE commands.

Terminal control

The CICS terminal control program provides for communication between user-written application programs and terminals and logical units, by means of terminal control commands. Terminal control uses the standard access methods available with the host operating system, as follows:

- BTAM (Basic Telecommunications Access Method) is used by CICS for most start-stop and BSC terminals. Optionally, TCAM (Telecommunications Access Method) can be specified.

- SAM (Sequential Access Method) is used where keyboard terminals are simulated by sequential devices such as card readers and line printers.

- VTAM (Virtual Telecommunications Access Method) or TCAM (Telecommunications Access Method) is used for systems network architecture (SNA) terminal systems.

Terminal control polls terminals to see if they are ready to transmit or receive data. Terminal control handles code translation, transaction identification, synchronization of input and output operations, and the line control necessary to read from or write to a terminal.

The application program is freed from having to physically control terminals. During processing, an application program is connected to one terminal for one task and the terminal control program monitors which task is associated with which terminal.

Terminal control is used for communication with terminals. In SNA systems, however, it is used also to control communication with logical units or with another CICS system.

Terminal control commands are provided to request the following services that are applicable to most, or all, of the types of terminal or logical unit supported by CICS:

- Read data from a terminal or logical unit (RECEIVE).

- Write data to a terminal or logical unit (SEND).

- Converse with a terminal or logical unit (CONVERSE).

- Synchronize terminal input/output for a transaction (WAIT TERMINAL).

- Send an asynchronous interrupt (ISSUE SIGNAL).

- Relinquish use of a communication line (ISSUE RESET).

- Disconnect a switched line or terminate a session with a logical unit (ISSUE DISCONNECT).

Read from Terminal (RECEIVE)

The RECEIVE command is used to read data from a terminal or logical unit. The INTO option is used to specify the area into which the data is to be placed.

Write to Terminal (SEND)

The SEND command is used to write data to a terminal or logical unit. The options FROM and LENGTH specify respectively the data area from which the data is to be taken and the length (in bytes) of the data.

5. Handling files

In this unit you will learn:

- different Access methods - VSAM, BDAM
- VSAM considerations - ESDS, KSDS, RRDS, Recoverable files
- different file handling commands - READ, WRITE, REWRITE, DELETE, UNLOCK, STARTBR, READNEXT, READPREV, RESETBR, ENDBR
- what are common exception or error conditions
- Committing and rollingback file updations
- what is an LUW

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 are not 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 will cover only VSAM key-sequenced data sets, accessed by key, in this Book. 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 will cover in the coming section.

Before describing how you read and write files, we should explain briefly about an important CICS table, the File Control Table (FCT) which we discussed in unit 1. This table contains one entry for each file used in any application in the system. 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 our employee details example, we will use the symbolic file name "EMPDATA".

Reading a file record

Let us see how we can read an employee file record in EMPMNT program. The user types in employee number and presses enter to see the employee details. The program first moves this employee number to the key field and then do a file read.

```
MOVE EMPNOMI            TO FEMPNO.

EXEC CICS READ
          FILE('EMPDATA')
          RIDFLD(FEMPNO)
          INTO(EMPREC)
          LENGTH(EMPREC-LEN)
          RESP(RSP-CODE)
END-EXEC.
```

Here EMPNOMI is where we have stored the employee number taken from the map, and EMPREC-LEN is a constant in working storage defined as the expected length of a record in the employee file. EMPREC is the working storage employee record definition.

```
01 EMPREC-LEN           PIC S9(4) COMP VALUE +42.
01 EMPREC.
   05 FEMPNO            PIC S9(4) COMP.
   05 FEMPNAME          PIC X(15).
   05 FEMPADDRESS       PIC X(25).
```

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 Book.

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 are covering in this Book, 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. 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.

Browsing a file

In program EMPLIST, if we are searching by employee number, 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.

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 is not allowed; **file browsing is strictly a read-only operation**.

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 have 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. Do not change it, either, because you will 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. The RESETBR command allows you to reset your starting point in the middle of a browse.

Reading the previous record

The READPREV command is almost like READNEXT, except that it lets you proceed backward through a data set instead of forward.

```
EXEC CICS READPREV
          FILE(filename)
          INTO(recarea)
          LENGTH(length)
          RIDFLD(fdbkarea)
END-EXEC.
```

Finishing the browse operation

When you have finished reading a file sequentially, you terminate the browse with the ENDBR command:

```
EXEC CICS ENDBR
          FILE(filename)
END-EXEC.
```

It tells CICS which browse is being terminated, and it must name a file for which a STARTBR has been issued earlier.

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.

Rewriting a file record

The REWRITE command updates the record you have just read. **You can use it only after you have 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.

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.

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 have read it for update. In this case, you must omit RIDFLD.

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 have 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.

ENDFILE means that you have 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.

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.

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 was not 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.

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.

- 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 is no space in the file to fit the record you have just tried to put there with a WRITE or REWRITE command. This does not mean that there is 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.

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. NOTFND may result from a user error, may be expected by the program, or may indicate an error in the program logic.

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.

Other file services

Before leaving the topic of file commands, we will list some of the other facilities that are available.

- You can use relative-record VSAM files (RRDS) as well as key-sequenced 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 have read for update if you decide not to update after all. The UNLOCK command is the means of doing this.

- You can access records without moving them into your program by using the SET option on the READ command.

- You can delete a whole block of adjacent records in a VSAM file with a single command (using the "generic delete" option).

- You can insert a whole block of records at once into a VSAM file ("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 are not 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.

Committing and Rollingback file updations

All the file updations can be committed by using the following CICS command:

```
EXEC CICS SYNCPOINT
END-EXEC.
```

On the above command, CICS will commit all file updations, issue SQL COMMIT to DB2 if we are using SQL commands in our program and make sure that the CICS journal (or log) will be in sync with the database log.

To rollback any modification issue

```
EXEC CICS SYNCPOINT
          ROLLBACK
END-EXEC.
```

If we do not issue either of these commands, at the end of the task, CICS issues an auto SYNCPOINT on successful completion and a ROLLBACK on an ABEND.

Logical Unit of Work (LUW)

As we mentioned in unit1, a task can further be split into several LUWs. An LUW is defined as follows: it starts when the first CICS command is executed in the present task and ends when a SYNCPOINT (with or with out rollback) is executed. In our EMPMNT example, whenever we add an employee record we will issue a SYNCPOINT.