Inside a CICS Program

Introduction 29
Conversations and Pseudo-conversations 30
The Communications Area 32
The COBOL Linkage Section 33
The Exec Interface Block 34
Commands to Access System Data 36
Translation Process 42
Logical Program Levels 43
Passing Data 44
Temporary Storage 51
Transient Data 52
Sample Program 54
Introduction

CICS programs are always subprograms.

This means that COBOL CICS programs always contain a LINKAGE SECTION. *Even if you don't use it*, CICS requires the LINKAGE SECTION for the Execution Interface Block (EIB).

Therefore, you can think of CICS as the *control* module.

![Diagram of CICS region with users, programs, maps, and files](CXC-05)
Conversations and Pseudo-conversations

When information is sent by an application program to a terminal, CICS provides the options, via the appropriate commands which will either:

- retain the current program in store and restart it at the next instruction (after the operator has responded)

or

- logically 'finish' the program, returning to CICS all the resources that it was using and specify which program is to be initiated next at that terminal when the operator subsequently responds.

The first type is a CONVERSATIONAL application, and the second is a PSEUDO-CONVERSATIONAL application.
When we compare operator response times to the time taken for a program to execute, we see that the PSEUDO-CONVERSATIONal approach imposes a smaller overhead to a CICS system. Resources are not held across terminal I/O and the programs tend to be smaller.

From the operator viewpoint, both types of application are indistinguishable. From the applications programming viewpoint, conversations are easier to write than pseudo-conversations as the instructions which process the response simply follow those which displayed the screen.

Most good practice guidelines recommend the pseudo-conversational techniques. However, the design of a pseudo-conversational application poses the question:

If all resources are released by the program which displays the screen, how does the next program obtain all the information which it requires to run?
The Communications Area

CICS provides several methods for passing data from one program to another. These include:

- hiding it on the screen (storing it with the dark attribute)
- in other forms of auxiliary storage available to CICS (Temporary Storage, Transient Data) – see later section
- in the 'COMMUNICATIONS AREA'

We are particularly interested in the third option here, since you will see it in almost all CICS programs. This is a special area which can be defined in COBOL in the LINKAGE SECTION and is passed from one program to another via CICS. It has the reserved COBOL name of **DFHCOMMAREA**.

Its length is determined by your requirements, but keep the following in mind:

- keep DFHCOMMAREA as small as possible
- check its limitations with your systems programmer or whoever has to define the maximum size in the CICS tables. This applies to each terminal. It is likely that the size will be the same for every terminal, but it need not be. *Do this before you design your programs.*
The COBOL Linkage Section

The DFHCOMMAREA will always be the first area coded in the LINKAGE SECTION of a CICS COBOL program. In the compilation listing it becomes the second area in the LINKAGE SECTION. This is because the Execution Interface Block will have been inserted by the CICS Translation phase.

It is possible to define other LINKAGE parameters in your program, but the requirements are more complex. The possible benefits arise from the following:

- when your program is compiled and linked, both the executable code and working storage are static and contribute to program size

- Linkage areas are allotted from CICS storage outside your program. This is done dynamically and is under your control. (The GETMAIN and FREEMAIN commands will be used.) It may be more efficient. However, these areas are indirectly addressed, which is a disadvantage.

Typical LINKAGE SECTION definitions might therefore appear as:

```cobol
LINKAGE SECTION.
  01  DFHCOMMAREA   PIC X(25).

LINKAGE SECTION.
  01  DFHCOMMAREA   PIC X(60).
  01  SCRN-AREA    PIC X(500).
```
The Exec Interface Block

The *Execution Interface Block* is copied into your program and provides a ready means of accessing system information simply by reference to the appropriate data-name rather than by the issue of a command.

The most important data elements which can be found there are:

- **EIBTIME**: the time the task started. *0hhmmss+ packed*
- **EIBDATE**: the current date. *01yyddd+ packed*
  
  00 for 1900-1999; 01 for 2000+

  the above are updated when a task is running by the command

  **EXEC CICS ASKTIME END-EXEC**

  You cannot otherwise update these or any other fields in the EIB.

- **EIBTRNID**: the transaction identifier (the one which *initiated* the task)
- **EIBTASKN**: the task number
- **EIBTRMID**: the terminal id
- **EIBCPOSN**: the cursor position
- **EIBCALEN**: the length of the COMMAREA
- **EIBAID**: key last pressed by the user *(e.g. PFn, PAn, CLEAR, ENTER)*
- **EIBDS**: last data set accessed
- **EIBFN**: last CICS command *(code)*
- **EIBRCODE**: can be used to check last return code
  
  *(but still need HANDLE CONDITION or RESP to avoid dumps)*
- **EIBREQID**: interval control request code
- **EIBRSRCE**: last resource *(file, TS, TD, terminal)* used
EXEC INTERFACE BLOCK

EIBTIME      = +0164933
EIBDATE      = +0102072
EIBTRNID     = 'CECI'
EIBTASKN     = +0000076
EIBTRMID     = '0004'
EIBCPOSN     = +00004
EIBCALEN     = +00000
EIBAID       = X'7D'
EIBFN        = X'0202'          (ADDRESS)
EIBRCODE     = X'000000000000'
EIBDS        = '........'
EIBREQID     = '........'
EIBRSRCE     = '        '
EIBSYNC      = X'00'
EIBFREE      = X'00'
EIBRECV      = X'00'
EIBATT       = X'00'
EIBEOC       = X'00'
+  EIBFMH       = X'00'
+  EIBCOMPL     = X'00'
EIBSIG       = X'00'
EIBCONF      = X'00'
EIBERR       = X'00'
EIBERRCD     = X'00000000'
EIBSYNRB     = X'00'
EIBNODAT     = X'00'
EIBRESP      = +0000000000
EIBRESP2     = +0000000000
EIBRLDBK     = X'00'
Commands to Access System Data

You can access information in CICS storage blocks held outside of the program by coding the ADDRESS command. This provides the address of the area in question.

```
MOVE 'DEBSP01' TO FZ-PROGID.
MOVE EIBTRNID TO FZ-DCDE.
MOVE 'ADDRESSING PCI        ' TO FZ-MESSAGES.
EXEC CICS
  ADDRESS
  TCTUA (PCIBAR)
END-EXEC.
MOVE 'ADDRESSING TWA' TO FZ-MESSAGES.
EXEC CICS
  ADDRESS
  TWA (TWABAR)
END-EXEC.
```

Apart from the EIB, it is unusual to find the other control blocks in modern programs.
Whilst the `ADDRESS` command gives a pointer to an external data area, the `ASSIGN` command obtains an *external value*.

**ASSIGN places CICS values in program variables**

The values which can be obtained include:

*System area lengths*
- CWALENG
- TCTUALEN
- TWALENG

*Device characteristics*
- SCRNHT
- SCRNWD

*Operator information* (may be needed for creating unique names e.g. TS queues)
- OPID
- OPCLASS
- OPSECURITY

*Network information*
- APPLID
- NETNAME
- QNAME

For example: `EXEC CICS ASSIGN OPID(MYOPNM)
END-EXEC`
ASSIGN

STATUS:  COMMAND EXECUTION COMPLETE

EXEC CICS ASSIGN
< ABCode(' ') >
< ABDump(' ') >
< ABProgram(' ') >
< ALTSCRNHt(+00000) >
< ALTSCRNWd(+00000) >
< APLyd(' ') >
< APLText(' ') >
< APPlid('CICSP1 ') >

: < BRidge(' ') >
< BTrans(' ') >
+ < CMDsec('X') >
+ < Color('.' ) >
< CWaleng(+00512) >
< DEFSCRNHt(+00024) >
< DEFSCRNWd(+00080) >
< DELimiter(' ') >
< DESTCount(+00000) >
< DEStID(' ') >
< DEStIDLeng(+00000) >
< DS3270(' ') >

+ < HiLight('.' ) >
< INITPARM(' ')
< INITPARMLen(+00000) >
< INPartn(' ') >
< INVokingprog('DFHECIP ') >

: < NATlanginuse('E') >
< NETname('TCP00001') >
+ < NEXTtransid('') >
+ < NUMtab('.' ) >
< OPClass('...') >
< OPid(' ' ) >
< ORgabcode(' ' ) >
< OUTline(' ' ) >
< PAGenum(+00000) >

< PROgram('DFHECID ') >
< PS('.' ) >
< Qname(' ') >
< RESSec('X') >
< RESTart('.' ) >
< RETurnprog('DFHECIP ') >
+ < SCR NHt(+00024) >
+ < SCR NWd(+00080) >
< Sigdata('.....') >
< Sosi(' ' ) >
< STARTcode('TD') >
< STATIONid(' ') >
< SYsid('PI') >
< TASKRIORITY(+00001) >
< Tctualeng(+00000) >
< Telerid(' ' ) >
< TERMCode('j2') >
< TERMPriority(+00000) >
< TEXTKybd('.' ) >
< TEXTPersist('.' ) >
< TRANpriority(+00001) >
< TWaleng(+00000) >
< UNattend('.' ) >
+ < USERId('CICSPUSER') >
+ < USERName(' ' ) >
< USERPriority(+00000) >
< Validation('.' ) >
Programs can also issue CEMT-style commands.

Using the INQUIRE and SET commands, it is possible to virtually duplicate the facilities of the master terminal operator by enabling an applications program to inquire about or change information relating to CICS resources such as:

- Datasets
- CICS itself
- Transactions
- Terminals

Use INQUIRE to obtain values from CICS table entries

It is possible for example to open and close datasets or just inquire on their status, e.g.

```
EXEC CICS INQUIRE FILE ('OAKMAST')
  OPENSTATUS (WS-FILE)
END-EXEC
```

and the CICS VALUE DATA AREAS (cvda) can be used to compare the result e.g.

```
IF WS-FILE = DFHVALUE (CLOSED)
  GO TO END-P1.
```

Further commands allow read/write access to the spool queues.
Language syntax

A CICS instruction in an application program may be equated to a COBOL or PL/1 statement; for COBOL, a single command may form a sentence terminated by a period, or several commands may be combined together with other COBOL statements to form a sentence.

For COBOL these may only appear in the PROCEDURE DIVISION of an application program and for PL/1 these are procedure statements only.

The general format of a command in COBOL is:

```
EXECUTE CICS function  option (argument)
  option (argument)
  .
  .
  .
  option (argument)
END-EXEC
```

The PL/1 equivalent simply replaces the END-EXEC with a semicolon.

Every CICS command returns a response value in the field EIBRESP.

Each command can be coded with the NOHANDLE option or with the RESP option.
Some options may require no argument, and the EXECUTE is commonly abbreviated to EXEC. The END-EXEC is mandatory for COBOL. The next example reads a record from a file:

```
EXEC CICS READ  FILE ('PAYMSTA')
    INTO (PAYMST)
    RIDFLD (EMPNO)
    NOHANDLE

END-EXEC
```

Commas may be used between options as an alternative to blanks.

A typical valid COBOL sentence with two commands might be

```
IF VAL = 'P14' OR VAL = 'P16'
    EXEC CICS READ  FILE ('PAYMMT2')
        INTO (PAYAREA)
        RIDFLD (ACCN)
        UPDATE
        NOHANDLE

    END-EXEC
    MOVE UP-N-FS TO ACCTNOA
    PERFORM UPDATE-REC
ELSE
    EXEC CICS RETURN END-EXEC
END-IF
```

Whilst in PL/1 we might have:

```
SELECT (VAL);
    WHEN('P14') DO;
        EXEC CICS READ FILE ('PAYMMT2')
            INTO (PAYAREA)
            RIDFLD (ACCN) UPDATE;
        ACCTNOA=UP_NFS;
        CALL UPDATE_REC;
        END;
    WHEN('P16') DO;
        EXEC CICS RETURN;
        END;
```

The Translation Process

Preparing a CICS COBOL or CICS PL/1 program is a two-stage process. The source program is pre-processed by the appropriate COMMAND LANGUAGE TRANSLATOR, which produces a listing and a punch deck to be passed to the compiler for compilation (and subsequently that output is link edited).

For COBOL, the translator:

- expands CICS commands to a series of MOVE statements and a CALL, then changes the original command to comments
- inserts a number of working storage fields for its own exclusive use, and includes an area for application program reference to CICS system information into the LINKAGE SECTION. This is the EXEC INTERFACE BLOCK.

The Enterprise COBOL compiler will do the translation as well as the compilation.
Logical Program Levels

When a program passes control to another program, and expects control to be returned to the point of transfer, then that program is considered to be at a higher level than the called program. The command which produces such a transfer of control is LINK.

Where CICS differs from the batch environment is that the XCTL command can transfer control to another program with no anticipated return to the program issuing the XCTL. Both programs are considered to be at the same logical level.

In the case of LINK, both programs are retained in storage, and the lower level module is deleted when control is returned to the higher level program.

Notice that when OAK6 issues a return, control goes back to OAK3 - even though there have been several programs intervening and OAK3 originally LINKed to OAK4.
Passing Data

There are several methods of transferring data between programs:

- DFHCOMMAREA
- on the screen (dark protected, perhaps)
- TEMPORARY STORAGE
- TRANSIENT DATA
- Other CICS storage areas
- Containers and channels (from CICS TS V3)
Using a COMMAREA

The EIB field \textit{EIBCALEN} gives the length of the data which has been passed. It is common to check this for zero to establish whether this is the first initiation of the program in the dialogue. i.e. \textit{is it first-time-in}?

\begin{center}
\textbf{Does the COMMAREA exist?}
\end{center}

Creating a COMMAREA

1. \textit{Return to CICS, saving COMMAREA}
The first program terminates and its data areas are deleted. However, the COMMAREA is preserved by CICS and is associated with the user’s terminal.

2. Program areas deleted, COMMAREA in CICS

When the user’s work resumes with the next task, the program checks EIBCALEN and uses its LINKAGE SECTION to access the COMMAREA contents.

3. New program uses LINKAGE to access COMMAREA
The program can copy the contents of the COMMAREA to working storage if required, and it can pass the COMMAREA contents to subsequent programs. Ideally, if you have a set of programs which communicate with each other, the COMMAREA should be a standard size and layout for each program.

4. Optional: copy COMMAREA to W-S

The theoretical maximum size of a COMMAREA is 32k (VTAM restriction), but to be on the safe side, keep it less than 2048 bytes.

The COMMAREA is a flexible method of passing data between transactions in multiple CICS regions and CICSplexes. In the latter case, the contents must not include storage addresses. This avoids inter-transaction affinities which force two transactions to run in the same CICS region, thus limiting the scope for balancing the workload across regions.
The second method of passing data between programs is not recommended except where the volume of data is very small (and the network is local).

Data is sent to the screen and returns to CICS when the user responds.

IBM does not recommend this method. If the user presses CLEAR, the data disappears!

Only character data should be used since hex data causes unpredictable results on 3270s.
A further method of passing data uses various CICS areas. These were used in old CICS programs, but are not now generally recommended.

ADDRESS enables a program to access external storage areas

CECI ADDRESS
STATUS: COMMAND EXECUTION COMPLETE
EXEC CICS ADDRESS
  < Acee( X'FF000000' ) >
  < Commarea( X'0BD001A8' ) >
  < Cwa( X'000C1000' ) >
  < Eib( X'001400D0' ) >
  < TCTua( X'FF000000' ) >
  < Twa( X'FF000000' ) >

Programs which use the addresses of these control blocks cause transaction affinities if they pass the addresses to other programs. This affects flexibility in a CICSplex.
Temporary Storage

This CICS facility allows the program to store data in a *queue* either in main storage or on auxiliary (disk) storage.

**TS queues are disk or main storage**

Access to this data is normally *sequential*, but it can be made *random* by specifying an ITEM number in the relevant command. There are commands for writing, reading, and updating items, and a queue can also be deleted if it is no longer required.

Temporary storage queues are set up dynamically when the first WRITE operation is performed. They are explicitly deleted under program control. **Do this as soon as possible** for efficiency. It is common to encounter WRITE errors as a result of failing to delete a queue at the appropriate point.

A queue is identified by an **eight** character name *[16 characters from CICS TS 1.3]*, and may be accessed by any task which *knows* that name. For this reason (and because CICS programs are re-entrant and shareable) it is usually necessary to construct a unique queue name for each task or dialogue. This is frequently done by extracting unique information from the EIB such as task number, terminal id, and operator id.

TS queues can be browsed using the CEBR transaction. You can also use Xpediter to view their contents.
Transient Data

This CICS *scratchpad* facility may be used as an alternative to TEMPORARY STORAGE. The main differences are:

- Transient Data queues *must* be predefined to CICS by the systems programmer in the CSD (CICS System Definition) file.

- These queues are *strictly sequential*, no random access is available. Records may only be read (from the appropriate type of device) *in the same order as they were written* (FIFO). Their main use is for exchange of batch data, either between CICS applications, or between the CICS partition and other partitions. The destination device may be disk, tape, or even line printer. Transient data queues are frequently used for printing.

- TD queues are *read-destructive*. You cannot therefore re-read an item.

- The TD queues have *four-character* names

**TD queues are pre-defined to CICS**

![Diagram showing the relationship between TS queue, TD queue, and DCT]

(CFB1/2;2005.01) © Oakwood Computing Ltd

North American Distribution Rights – Computer Education Techniques, Inc.
TD queues can be defined to be CICS only, or they may be defined to be written in CICS and read by other regions e.g. batch.

- those within CICS are described as INTRAPARTITION. These are disk only.

- those outside CICS are described as EXTRAPARTITION. They may be on disk or on tape or 'slow' devices.

**INTRAPARTITION** TD queues may be used to initiate *asynchronous tasks*. Normally, tasks are initiated and controlled via terminals. However, a TD queue may be defined with a *trigger-level*: when the number of items on the queue reaches the specified *trigger*, a separate task is initiated. You should be aware that jobs done by that task need not have anything to do with the data in the queue!

Typical applications might be:

- writing a transaction audit trail (*why occupy a terminal?*)

- print tasks - invoices etc.
Sample Program

The following sample program OAK1 has an associated transid of OAKT. It is invoked after a map PPMSM01 has been displayed and the operator has responded.

The program verifies that the operator pressed the 'ENTER' key and selected a menu option '1' or '2'. If neither of these were selected, OAK1 redisplay the screen with an error message. Otherwise, for option '1' it transfers control to program OAK2, and for '2', control is passed to OAK3.

An employee name is also keyed onto the screen, and it is passed to OAK2 and OAK3. OAK1 is pseudo-conversational, and if it needs to redisplay the screen, the next program to be invoked when the operator responds is, in fact, itself. No data is passed (to itself) in this case.

Do not worry about the commands at this stage - study the layout.
COBOL example

IDENTIFICATION DIVISION.
PROGRAM-ID. OAK1.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
COPY DFHBMSCA.
COPY DFHAID.
COPY PPMSM01.
01 PPMSM01I.
   02 FILLER PIC X(12).
   02 EMPNL PIC S9(4) COMP.
   02 EMPNF PIC X.
   02 FILLER REDEFINES EMPNF.
      03 EMPNA PIC X.
      02 EMPNI PIC X(30).
      02 OPTL PIC S9(4) COMP.
      02 OPTF PIC X.
      02 FILLER REDEFINES OPTF.
         03 OPTA PIC X.
         02 OPTI PIC 9.
         02 MESL PIC S9(4) COMP.
         02 MESF PIC X.
         02 FILLER REDEFINES MESF.
            03 MESA PIC X.
            02 MESI PIC X(50).
01 PPMSM01O REDEFINES PPMSM01I.
   02 FILLER PIC X(12).
   02 FILLER PIC XXX.
   02 EMPNO PIC X(30).
   02 FILLER PIC XXX.
   02 OPTO PIC 9.
   02 FILLER PIC XXX.
   02 MESO PIC X(50).

LINKAGE SECTION.
PROCEDURE DIVISION.

EXEC CICS HANDLE AID ENTER (OPT-OK)
   ANYKEY (INVAL-OPT)
END-EXEC.

EXEC CICS HANDLE CONDITION ERROR (DUMP1)
END-EXEC.

MOVE LOW-VALUES TO PPMSM01I.
EXEC CICS RECEIVE MAP ('PPMSM01')
END-EXEC.

OPT-OK.
   IF OPTI NOT NUMERIC
      GO TO INVAL-OPT.
   IF OPTI = 1
      EXEC CICS XCTL PROGRAM ('OAK2')
         COMMAREA (EMPNI)
         LENGTH (30)
   END-EXEC
ELSE
   IF OPTI = 2
      EXEC CICS XCTL PROGRAM ('OAK3')
         COMMAREA (EMPNI)
         LENGTH (30)
   END-EXEC.

INVAL-OPT.
   MOVE 'WRONG OPTION OR KEY USED' TO MESO.
   MOVE 'Z' TO MESA.
* PROT ALPHA BRIGHT TO ATTRIBUTE FOR MESSAGE FIELD
* POSITION CURSOR AT EMPLOYEE NAME FIELD
   EXEC CICS SEND MAP ('PPMSM01')
      FROM (PPMSM01I)
      ERASE
     _CURSOR
      FREEKB
   END-EXEC.

EXEC CICS RETURN TRANSID ('OAKT')
END-EXEC.

DUMP1.
   EXEC CICS ABEND
END-EXEC.

* ABEND THE TASK ENSURING WE GET DUMP
GOBACK.
PROCEDURE DIVISION.

A100-FIRST SECTION.
MOVE LOW-VALUES TO PPMSM01I
EXEC CICS RECEIVE MAP ('PPMSM01')
NOHANDLE
END-EXEC
IF EIBRESP NOT = DFHRESP(NORMAL)
PERFORM Z999-DUMP1
END-IF
IF EIBAID = DFHENTER PERFORM A200-OPT-OK
ELSE                 PERFORM A300-INVAL-OPT
END-IF.

A200-OPT-OK SECTION.
IF  OPTI NOT NUMERIC
PERFORM A300-INVAL-OPT
END-IF
EVALUATE OPTI
WHEN 1    EXEC CICS XCTL  PROGRAM ('OAK2')
COMMAREA (EMPNI)
LENGTH (30)
NOHANDLE
END-EXEC
WHEN 2    EXEC CICS XCTL PROGRAM ('OAK3')
COMMAREA (EMPNI)
LENGTH (30)
NOHANDLE
END-EXEC
END-EVALUATE
IF EIBRESP NOT = DFHRESP(NORMAL)
PERFORM Z999-DUMP1
END-IF.

A300-INVAL-OPT SECTION.
MOVE 'WRONG OPTION OR KEY USED' TO MESO
MOVE DFHPROTI TO MESA
* PROT ALPHA BRIGHT TO ATTRIBUTE FOR MESSAGE FIELD
MOVE -1 to EMPNL
* POSITION CURSOR AT EMPLOYEE NAME FIELD
EXEC CICS SEND MAP ('PPMSM01')
FROM (PPMSM01I)
ERASE
CURSOR
FREEKB
END-EXEC
EXEC CICS RETURN TRANSID ('OAKT')
END-EXEC.

Z999-DUMP1 SECTION.
EXEC CICS ABEND
END-EXEC
* ABEND THE TASK ENSURING WE GET DUMP
GOBACK.