ibi™ FOCUS®

Host Language Interface User Guide

Version 9.3.2 | November 2024

Copyright © 2021-2025. Cloud Software Group, Inc. All Rights Reserved.

Contents

Contents ... **2**

Introduction to HLI ... **8**
What is HLI? ... 8
Why the Host Language Interface? ... 8
How Can You Create HLI Applications? ... 9
Navigating Through a FOCUS Data Source ... 9
The Dummy SYSTEM Segment ... 10
Reading the Data Source ... 12
Test Relations ... 13
The File Communication Block (FCB) ... 15
Incorporating HLI Commands in the Application Program ... 15

Preparing HLI Work Areas ... **17**
Initializing the File Communication Block (FCB) ... 17
FCB Layout ... 17
NEX C from A ... 21
Definitions of FCB Terms ... 22
Defining the Record Work Area ... 22
Internal Data Representation Formats ... 22
Alignment of Data Offsets in the Work Area ... 24
Size of the Work Area ... 25
Defining the Work Area for a File With Descendant Segments ... 25
Declaring Multiple FCBs ... 26

Using HLI ... **28**
Writing HLI Programs ... 28
Properties of Master Files for Use With HLI ... 28

C Program Considerations 28
Include the HLI Header File in a C Program 29
Declare the Fcb Name Variable 29
Declare a Session Handle and Issue the Connection Call 29
Declare an HLI Handle and Issue the Connection Call 30
Close the HLI Connection 31
Call to Close the Session Connection 31
Sample HLI Programs 31
- Master File Used With the Sample Programs 32
- Sample C Program 32
- Sample FORTRAN Program 38
- Sample COBOL Program 42
- Sample PL/1 Program 46

Initializing the FCB 51
- Initializing the FCB in a C Program 51
- Initializing the FCB in a FORTRAN Program 51
- Initializing the FCB in a COBOL program 52
- Initializing the FCB in a PL/I program 52

Opening the FOCUS Data Source 53
Open a FOCUS Data Source (OPN) 53
- Opening a FOCUS Data Source in a C Program 54
- Opening a FOCUS Data Source in a FORTRAN Program 55
- Opening a FOCUS Data Source in a COBOL Program 55
- Opening a FOCUS Data Source in a PL/I Program 55

Data Offsets in the Work Area 55
Using a Show List 57
Select a List of Fields in a FOCUS Data Source (SHO) 57
- Using a Show List in a C Program 59
- Using a Show List in a FORTRAN Program 59
- Using a Show List in a COBOL Program 60
- Using a Show List in a PL/I Program 60

Locating Records 61
Expressing Test Relations 61
Expressing Test Relations in a C Program 61
Expressing Test Relations in a FORTRAN Program 62
Expressing Test Relations in a COBOL Program 62
Expressing Test Relations in a PL/I Program 63
Logical Reads 64
Move to the First Target Segment Instance Under the Anchor (FST) 64
Move to the Next Segment in a Logical Read (NEX) 66
Using Logical Reads in a C Program 68
Using Logical Reads in a FORTRAN Program 69
Using Logical Reads in a COBOL Program 69
Using Logical Reads in a PL/I Program 69
Physical Reads 69
Move to the First Physical Segment Instance (FSP) 70
Retrieve the Next Physical Occurrence of the Target Segment (NXP) 71
Indexed Reads 73
Retrieve a Segment Instance Using an Index (NXD) 74
Retrieving With a Backkey 75
Retrieve a Previous Target Segment Using a Backkey (NXK) 76
Altering the File 77
Including New Segments 78
Include New Segment Instances (INP) 78
Including New Segment Instances Using a C Program 80
Including New Segment Instances Using a FORTRAN Program 80
Including New Segment Instances Using a COBOL Program 80
Including New Segment Instances Using a PL/I Program 81
Changing Information in the File 81
Change Information in a Segment Instance (CHA) 81
Changing a Segment Instance in a C Program 83
Changing a Segment Instance in a FORTRAN Program 84

Changing a Segment Instance in a COBOL Program 84
Changing a Segment Instance in a PL/I Program 85
Deleting Segments From a File 86
Delete Segments From a File (DEL) 86
Deleting a Segment Instance in a C Program 87
Deleting a Segment Instance in a FORTRAN Program 88
Deleting a Segment Instance in a COBOL Program 88
Deleting a Segment Instance in a PL/I Program 89

Testing Status, Using Log Facilities, and Handling Errors 90
Testing Status 90
Using the Diagnostic Log Facility: ECHO and STAT 90
Log File Locations 91
Using the ECHO Log Facility 92
Using the STAT Log Facility 93
Error Handling 96

Creating an Executable HLI Program 97
Constructing a DLL Under ibi WebFOCUS 97
Compile and Link a C Program 98
Constructing a Load Module Under z/OS 99
Creating a Load Module Under z/OS 99
HLI Allocations 100
Sample HLI Batch Job 101

HLI and Simultaneous Usage of FOCUS Databases 102
Using the SU Profile 102
Multi-Threaded HLI/SU Reporting Facility (Mainframe FOCUS Only) 103
Multi-Threaded HLI/SU Reporting Under z/OS FOCUS 104
Preparing an FCB for Multi-Threaded HLI/SU Reporting Under z/OS FOCUS 105

HLI Command Summary 106
HLI Command Summary Chart 106

HLI Parameter Description Chart ... 107
Alphabetical List of HLI Commands ... 110
CHA (Change) Command ... 110
CLO (Close) Command ... 112
DEL (Delete) Command ... 112
FSP (First Physical) Command ... 113
FST (First) Command ... 114
INFO (Information) Command ... 115
INP (Input) Command ... 118
NEX (Next) Command ... 119
NXD (Next Through Index) Command ... 121
NXK (Next Through Backkey) Command ... 122
NXP (Next Physical) Command ... 124
OPN (Open) Command ... 125
SAV (Save) Command ... 127
SHO (Show) Command ... 128

HLI Status Return Codes ... 130
HLI Return Code Chart ... 130

Using the GENCPGM Build Tool ... 135
Using GENCPGM ... 135
USAGE Chart (Typical Syntax Plus Extended Options) ... 137
Compile and Link a Procedure ... 142
GENCPGM Usage Notes ... 144
Language and Platform Notes ... 145
Build Rules ... 146
- Generating a Subroutine Program From a C Source File ... 147
- Generating an HLI Program From a C Source File ... 147
- Generating a CALLPGM Program From a C Source File ... 147

Migrating CMS HLI Programs to UNIX or Linux ... 149

Changes Needed to the CMS HLI Program 149
Changes Needed to the FCB When Migrating a CMS HLI Program 151
Migrating Simultaneous Usage Applications From CMS to UNIX or Linux 152
SU Under CMS 152
SU Under UNIX or Linux 153

ibi Documentation and Support Services 154

Legal and Third-Party Notices 155

Introduction to HLI

The Host Language Interface (HLI) enables programs written in 3GL languages such as C, C++, FORTAN, COBOL, Assembler, RPG, or PL/1 to access FOCUS® or XFOCUS data sources. Once created, programs must be compiled and linked. HLI consists of a series of commands that you incorporate into your application programs.

Note: FOCUSand XFOCUS data sources can be accessed identically using HLI. In the remainder of this manual, all references to FOCUS data sources and all references to FOCUS data sources also apply to XFOCUS data sources.

What is HLI?

The Host Language Interface (HLI) enables you to access FOCUS databases using simple calls to the HLI from conventional programming languages. HLI commands allow the application program to open and close one or more FOCUS files and input, change, and delete data in these files. Records in the database can be located using logical, sequential, and indexed reads, as well as reads according to a record's specific address within the file.

Your program incorporates HLI commands as calls to a subroutine. Application programs using HLI can access both local FOCUS databases and FOCUS databases on a FOCUS Database Server (FDS, also called a sink machine). It is, therefore, possible for HLI programs to access FOCUS databases that are in use by other HLI programs as well as by FOCUS users who are updating those same databases using MODIFY or MAINTAIN or writing reports and generating graphs.

Why the Host Language Interface?

HLI makes it possible to have FOCUS or ibi™ WebFOCUS® applications that interface with specialized hardware and software, such as production control systems, in which data streams may be automated from bar code readers, digital scales, or other devices. HLI is also useful in third-generation applications that require an underlying database. There are several other reasons for using HLI:

- You can use an existing application with a FOCUS data source.
- You can create one application with access to both FOCUS and non-FOCUS data sources.
- You can update a FOCUS data source and report from it in one pass.

Caution: HLI can enter and alter FOCUS data sources outside of normal FOCUS data source processing and control. You must understand the structure and contents of your FOCUS data source before you use HLI. Otherwise, you may seriously damage the file. If you want more information about the structure of FOCUS files, consult the *Describing Data* manual.

How Can You Create HLI Applications?

To use HLI, you must create a Master File as you would for any FOCUS data source.

Your application program must do two things to use HLI:

- Define the File Communication Block (FCB) and work areas.
    - The FCB is a reserved area containing information about the FOCUS database, including its logical name (for example, on z/OS its ddname or, in UNIX, its filename). It monitors your current position within the FOCUS database and keeps track of the actions your application takes. HLI and your application both use the FCB.

      One FCB is required for each FOCUS file accessed in an application program. You can open a file several times concurrently by defining multiple FCBs for the file. You must issue a close (CLO) command for each opened FCB.
    - Work areas are defined spaces where retrieved segments are held (see Preparing HLI Work Areas).
- Incorporate HLI commands.

Navigating Through a FOCUS Data Source

Your HLI application navigates through a ibi™ FOCUS® data source using HLI commands. The basic command for moving from one segment instance to another is the NEX (Next)

command. Starting at an anchor segment representing the current position in the file, your application proceeds to a target segment. (The target segment can be above the anchor in the segment hierarchy.) Retrieved fields from the specified segments, including the anchor and target segments, are placed in a record work area that you must define in your HLI application. You can retrieve all fields in those segments or select fields to be retrieved by creating a show list.

Your HLI program can locate segment instances for retrieval by searching for an instance that satisfies selection tests (a qualified move) or by looking for the next segment instance without testing (an unqualified move).

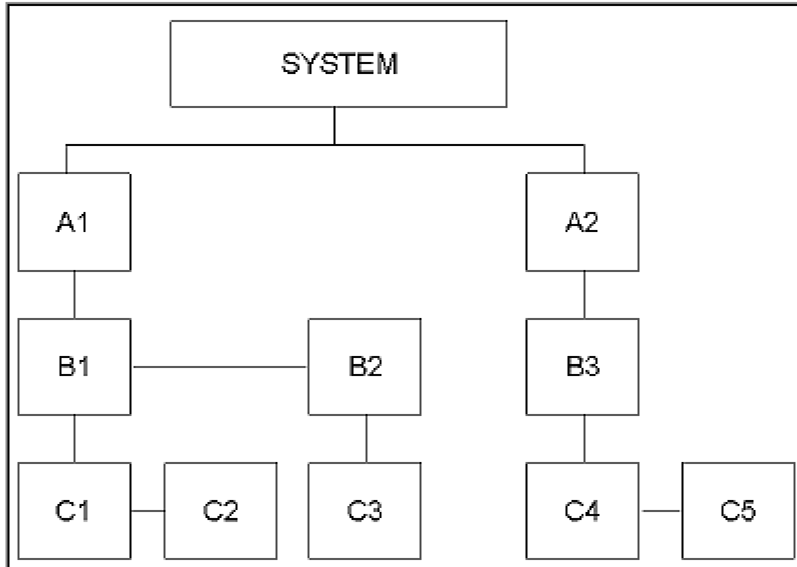
The Dummy SYSTEM Segment

The instances in the root segment form one chain considered descended from a dummy segment called the SYSTEM segment. The SYSTEM segment has a null value and is always considered current. It cannot be retrieved or modified. It can never be named as the target segment.

Consider the following Master File:

```
FILE=EXAMPLE1, SUFFIX=FOC,$
SEGNAME=A,$
 FIELD=.........,$
SEGNAME=B, PARENT=A, SEGTYPE=S1,$
 FIELD=.........,$
SEGNAME=C, PARENT=B, SEGTYPE=S1,$
 FIELD=.........,$
```

The following diagram represents the segment instances in the file, with the dummy SYSTEM segment added:



The following table illustrates how a series of NEX commands will search this file. The NEX command specifies the anchor segment (the position where your application starts), and the target segment (the segment to which you want to move), using FOCUS® pointers to move from segment to segment. The anchor segment determines the target segment, as shown by the following series of NEX commands.

Current Position	Target	Anchor	Explanation
`SYSTEM`	`NEX C`	`from SYSTEM`	Retrieves A1 B1 C1
`A1 B1 C1`	`NEX C`	`from A`	Retrieves A1 B1 C2
`A1 B1 C2`	`NEX C`	`from B`	Retrieves no segment
`A1 B1 C2`	`NEX B`	`from A`	Retrieves A1 B2
`A1 B2`	`NEX A`	`from`	Retrieves A2

Current Position	Target	Anchor	Explanation
		`SYSTEM`	
`A2`	`NEX B`	`from A`	Retrieves A2 B3
`A2 B3`	`NEX C`	`from B`	Retrieves A2 B3 C4
`A2 B3 C4`	`NEX C`	`from B`	Retrieves A2 B3 C5

Reading the Data Source

There are three ways to read a FOCUS data source with HLI:

- Logical reads.
- Physical reads.
- Indexed reads.

Logical Read Commands

The logical read commands are FST (First) and NEX (Next). They follow the logical pointers in the database to search the segments in their logical sequence (see Using HLI).

Physical Read Commands

The FSP (First Physical) and NXP (Next Physical) commands read FOCUS files as if they were sequential files. Segment instances are returned in the order in which they are stored, ignoring hierarchical links and links between siblings. Physical read commands provide faster access if the order of the segment instances returned is not significant.

The order in which data is stored is often quite different from the order that makes sense to a user. For example, if the database has been modified extensively, it may require many I/Os to retrieve the desired segments logically. In such cases, the physical read commands may improve the speed of your application program (see Using HLI).

Indexed Read Commands

The NXD (Next through Index) command retrieves values using the index of a specified field. This is particularly efficient when you are looking for a specific field value, since an index is an internally stored and maintained table of data values and locations that speeds retrieval (see Using HLI).

Test Relations

You can use test relations to control which instances you retrieve using HLI. The following test relations are supported:

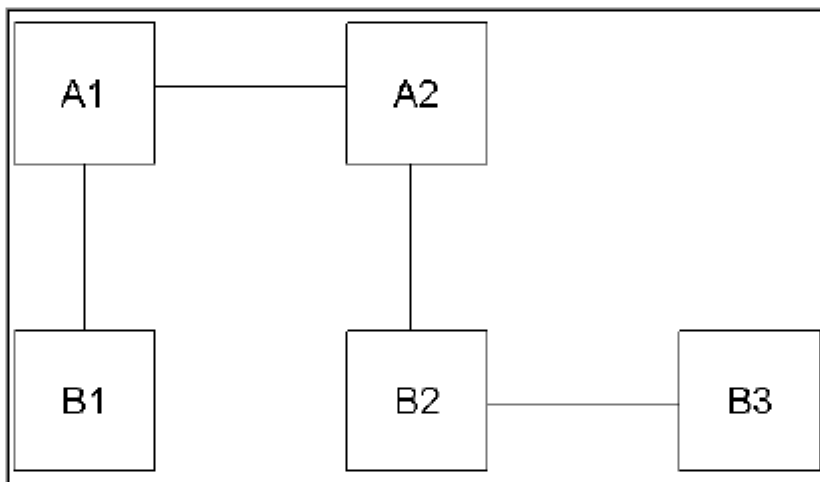
Test Relation	Meaning
`EQ`	Equal
`NE`	Not equal
`LT`	Less than
`GT`	Greater than
`LE`	Less than or equal
`GE`	Greater than or equal
`CO`	Contains
`OM`	Omits

Contains and Omits tests are useful when screening alphanumeric data fields.

Since the test relations are interdependent, a segment instance is rejected as soon as any test relation fails.

The tests are not necessarily performed in the order specified, but instead may be performed segment by segment. However, tests are performed in the specified order within the same segment. As soon as a test on a segment fails, the search continues with the next potential target segment. The descendants of a segment that did not meet the conditions are not examined.

When the target is above the level of the anchor in the hierarchy, the search is abandoned as soon as any test fails. For example, consider this diagram of segment instances:



Assume the current position is B1. From B1 (the anchor segment), the command

```
NEX A FROM B
```

will retrieve A1 (the target segment) if all conditions are met; if conditions are not met, the search will cease. A2 will not be considered, because even if it met the necessary qualifications, it would violate the definition of the current position because it is not linked to current B.

Note: You must use the FST (First) command when you are trying to retrieve the parent of a segment (see HLI Command Summary, for information on FST). If A1 were already our current position on segment A, the NEX command would be unsuccessful even if A1 met the qualifying conditions because there would be no change in the current position.

When the target segment is a descendant of the anchor segment, all segment instances along the anchor to target path are tested until one meets the conditions.

The File Communication Block (FCB)

The File Communication Block is a 200-byte structure that monitors the file on which you are working. One FCB is required for each FOCUS data source opened by an application program. More than one FCB may be attached to a single FOCUS data source.

Each FCB is unique and is identified by its address. This address becomes an internal token for manipulating multiple FCBs. You cannot copy an FCB to another location and use it to access a file. When finish using an FCB, you must issue a CLO (close) command for it.

See Preparing HLI Work Areas, for a discussion of initializing the FCB.

Incorporating HLI Commands in the Application Program

You incorporate HLI commands as calls from your host application program. For example, to use the HLI command INP (Input) on the segment named TOP, with an FCB declared as FCB1 and a work area declared as WKAREA, you would use the following syntax

In C:

```
edahliCall(hliHandle, 5, "INP ", &fcb, &wkarea, "TOP         ", 0);
```

In FORTRAN:

```
CALL FOCUS ('INP ', FCB1, WKAREA, 'TOP         ', 0)
```

In COBOL:

```
CALL 'FOCUS' USING INP FCB1 WKAREA TOP NUMB0.
```

In PL/1:

```
CALL FOCUS ('INP ', FCB1, WKAREA, 'TOP      ', 0);
```

HLI commands, except for the INFO command, are three characters long. If you use an argument in an HLI command as a literal string, as in the C, FORTRAN, and PL/1 examples, you must pad it with trailing blanks to make the argument the correct number of characters. For example, commands must be declared as 4 characters and segment names as 8 characters. If, instead, you supply an HLI argument as a variable, as is done in the COBOL example, the variable format establishes the length.

All the variables and parameters used with HLI must be declared and initialized following the conventions of the application language you are using.

Using HLI, illustrates HLI operations. It includes annotated examples of applications written in C, FORTRAN, COBOL, and PL/1 that use HLI commands, as well as sections that explain their syntax. HLI Command Summary, provides detailed instructions for using the HLI commands.

Preparing HLI Work Areas

Each HLI program requires at least one File Communication Block to monitor its operation and a work area in which retrieved segment instances can be placed and from which new data can be taken before saving it in the FOCUS data source.

Initializing the File Communication Block (FCB)

The File Communication Block is a 200-byte array that internally monitors the file in which you are working. It contains general file information, such as file name, as well as information about your current position within that file.

Each FCB is unique and is identified by its address. This address becomes an internal token for manipulating multiple FCBs.

You cannot copy an FCB to another location and use it to access a file.

When you are finished using an FCB, you must issue a CLO (close) command for it.

Initialize the FCB from byte 1 to 88 (which will contain file information) to blanks, file name, file type, mode, and, optionally, ECHO. All alphanumeric parameters, except the password, must be entered into the FCB using uppercase letters.

FCB Layout

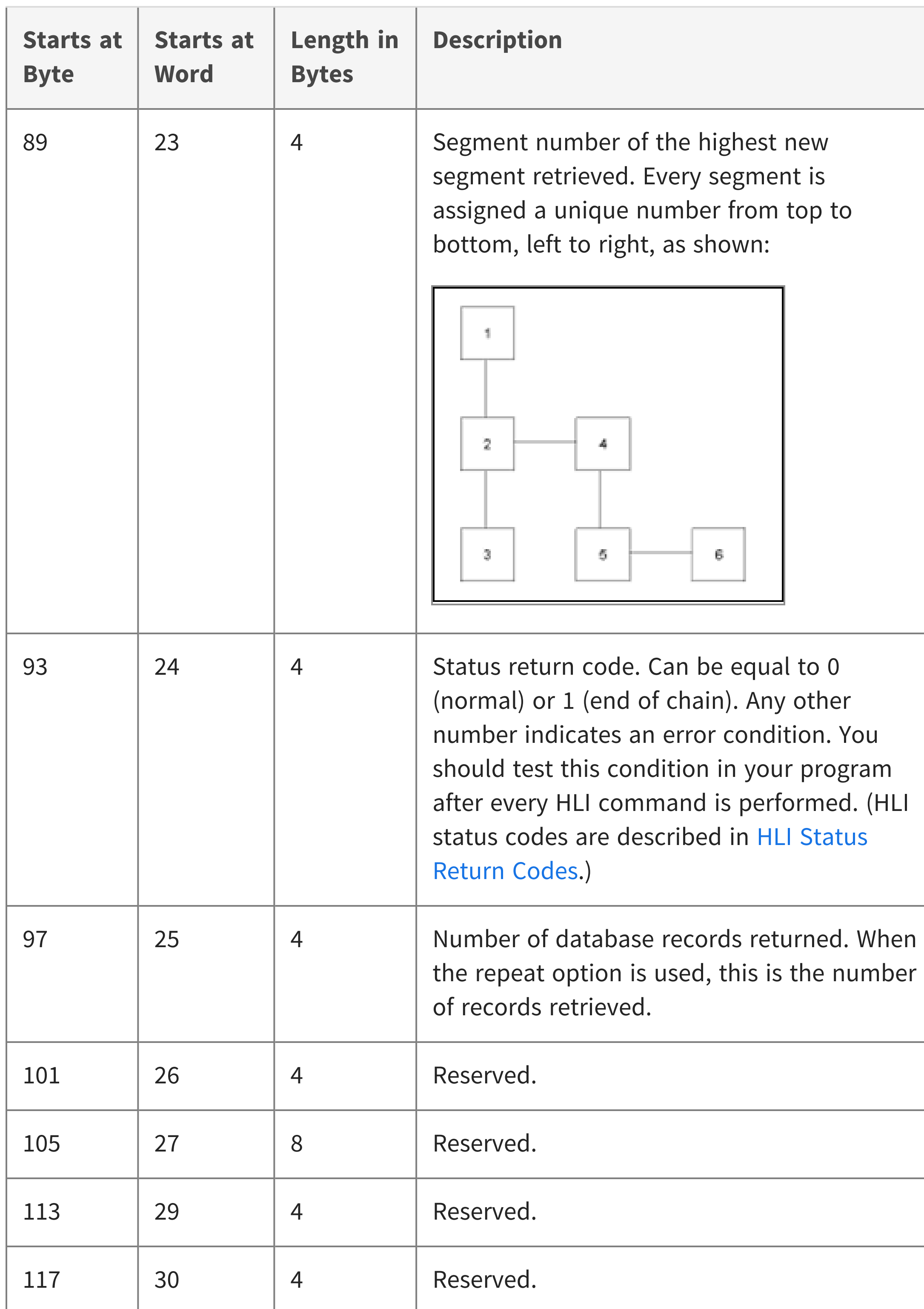
Each FCB is 200 bytes long and is organized as illustrated in the following diagram. Contents of the FCB are described in the chart immediately following this diagram. Not all parts of the FCB are used in all environments.

<table>
<tr><td>Filename</td><td>Filetype</td><td>File Mode</td><td>SU</td><td colspan="2">Procedure Name</td><td colspan="2">Sink Machine</td></tr>
<tr><td>Reserved</td><td>Reserved</td><td colspan="2">Backkey</td><td colspan="2">ECHO/STAT Logging</td><td colspan="2">Password</td></tr>
<tr><td>New Segment Name</td><td>Segment #</td><td>Status Code</td><td># of Rec. Ret.</td><td colspan="2">Reserved</td><td>Reserved</td><td>Reserved</td></tr>
<tr><td>Reserved</td><td>Reserved</td><td colspan="2">Reserved</td><td>Work Area Size</td><td>Work Area Length</td><td colspan="2">Reserved</td></tr>
<tr><td colspan="8"></td></tr>
</table>

The following chart describes the contents of the various bytes in the FCB layout. See Initializing the File Communication Block (FCB), for an explanation of alignment terms used in the FCB.

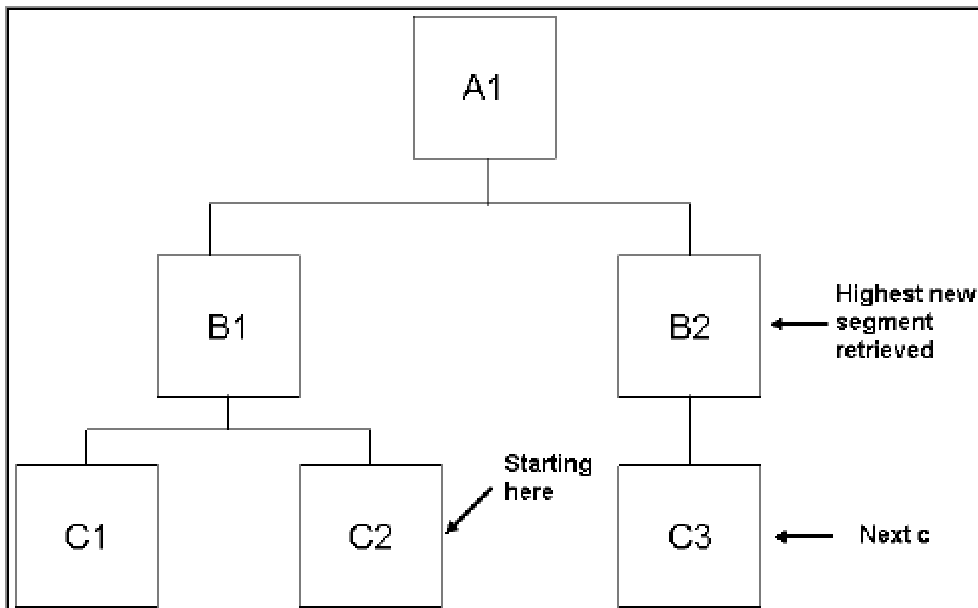
Starts at Byte	Starts at Word	Length in Bytes	Description
1	1	8	Logical name of the database (for example, ddname in z/OS).
9	3	8	File type (blank).
17	5	4	File mode (blank).
21	6	4	Simultaneous Usage flag. The value is • SU if running under a FOCUS Database Server (sink machine). • SULO if running multi-threaded (z/OS only) • Blank for direct access.
25	7	8	Procedure name. You can place information

Starts at Byte	Starts at Word	Length in Bytes	Description
			here that will appear in the PROCNAME column of the extended HLIPRINT trace (for more information, see Using HLI).
33	9	8	Name of FOCUS Database Server (for SU) on which the database is located. This machine must be up and running. On z/OS, this is the ddname of the sink machine's communication data set, which must be allocated. For the WebFOCUS® and WebFOCUS, this is the FDS node (typically FOCSU01 by default).
41	11	8	Reserved.
49	13	12	Reserved.
61	16	8	Backkey. Address a key of the target segment. This value changes each time you retrieve a target segment, and can be used later in the NXK command (for information, see Using HLI).
69	18	4	Logging option. If the value is ECHO or STAT, transaction is written to the HLIPRINT file (see Using HLI). If value is blank, nothing is written to HLIPRINT.
73	19	8	User access password. Required to access a file protected by the FOCUS DBA security restrictions.
81	21	8	New segment name. Highest new segment retrieved. (See Initializing the File Communication Block (FCB).)

Starts at Byte	Starts at Word	Length in Bytes	Description
89	23	4	Segment number of the highest new segment retrieved. Every segment is assigned a unique number from top to bottom, left to right, as shown: 
93	24	4	Status return code. Can be equal to 0 (normal) or 1 (end of chain). Any other number indicates an error condition. You should test this condition in your program after every HLI command is performed. (HLI status codes are described in HLI Status Return Codes.)
97	25	4	Number of database records returned. When the repeat option is used, this is the number of records retrieved.
101	26	4	Reserved.
105	27	8	Reserved.
113	29	4	Reserved.
117	30	4	Reserved.

Starts at Byte	Starts at Word	Length in Bytes	Description
121	31	4	Reserved.
125	32	4	Reserved.
129	33	4	Size of one returned work area (total length of fields requested).
133	34	4	Total length of the work area returned (if you are using the repeat option).
137	35	64	Reserved.

NEX C from A



Definitions of FCB Terms

The following definitions describe alignment terms as used in the FCB:

Term	Definition
`Word`	Refers to a 4-byte word.
`Word Alignment`	Means that the variable is on a full word boundary, which is a multiple of four bytes from the beginning of the data structure. The data structure must begin on a double word boundary.
`Double-Word Alignment`	Means that the variable is on a double word boundary, which is a multiple of eight bytes from the beginning of the data structure.

Defining the Record Work Area

The application program must provide an area into which a retrieved segment instance can be placed, or from which new data can be taken. The size and layout of this data structure are controlled by the data fields selected. In this document, this type of structure is variously referred to as the *work area*, the *change literals* array, the *inpliteral* array, or the *testliterals* array, depending on the way it is used. Fields can be selected by the SHO command.

The order in which the fields are named in the show list determines the order in which their data values are placed in the work area. If no SHO command is issued, the default show list is all the fields in the file. The usage formats of the data fields determine the size of this area.

Internal Data Representation Formats

All lengths are rounded up to multiples of 4 so that each data field starts on a full-word boundary in this area, or a double-word boundary for double-precision, floating-point data.

The following table describes the internal formats of data stored in FOCUS data sources and the representation of missing values for each format(in case your program needs to test for missing data):

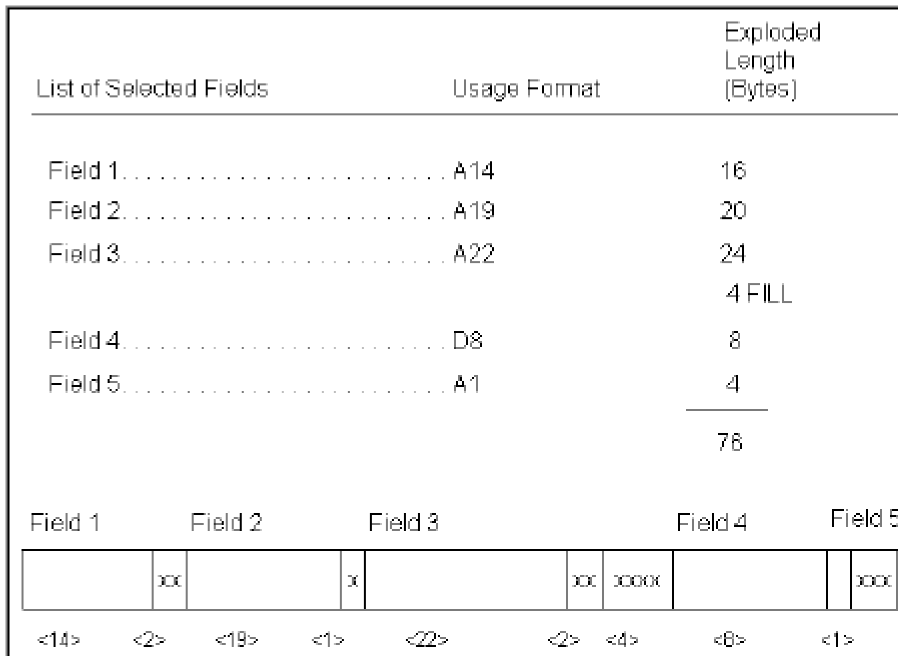
Format	Internal Representation	Missing value stored as
Alpha (A*n* or A*n*V)	Stores alphanumeric data as entered. Allocates one byte per character, rounded to a full word boundary. If the format is A*n*V, there are two additional binary bytes for the number of non-blank characters in the character string. For example, an A10V field with a value of ITALY would require 12 bytes, where the first 2 would have the value 5 in binary, and the remaining 10 characters would be `'ITALY     '` Performs no translation of uppercase and lowercase.	A left-justified period (.)
Integer (I*n*)	Standard binary integer format, 4 bytes.	-9998998
Floating Point (F*n*)	Full-word (4 bytes) floating point.	-9998998.
Floating PointDouble Word (D*n*)	Double word (8-byte) floating point.	-9998998.
Packed (P*n*)	Signed packed format. Size is: • 8 bytes for *n* = 1 through 15 • 16 bytes for *n* = 16 to 31.	-9998998.

Note:

- TX (text) fields are not supported for HLI.
- FOCUS data sources support several formats for storing dates. Dates can be defined as:
    - I, P, or A format with date display options, for example I8YYMD.
    - Date format, for example YMD. In this case, the 4-byte integer stored is the number of days from the base date of December 31, 1900.
    - 1. • Date-Time format, for example HYYMDm. They are stored as 8 bytes, 12 bytes if the time includes microseconds, or 16 bytes if the time includes nanoseconds.

Alignment of Data Offsets in the Work Area

The following diagram illustrates the alignment of data offsets in the work area:



Note:

- In PL/1, you should not rely on PL/1 to align the bytes, as the algorithm it uses may not be one that FOCUS uses.
- In the diagram, unused filler bytes are represented by x.

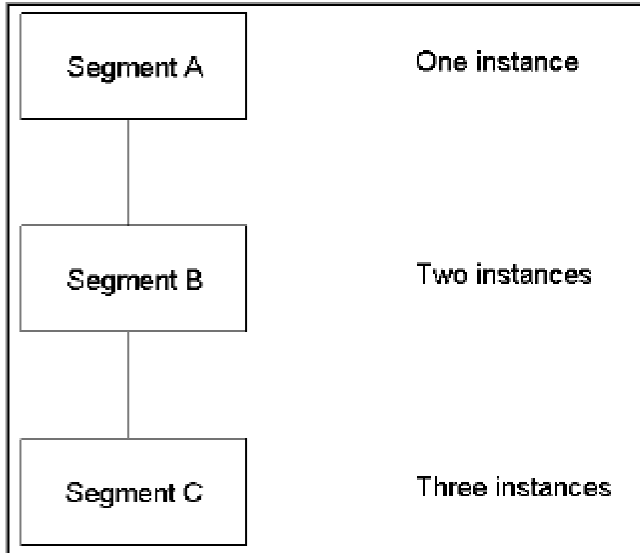
Size of the Work Area

The size of the work area depends on the number of selected fields in the file. The length of the standard work area is returned in the FCB, after a call to FOCUS with the OPN (Open) command.

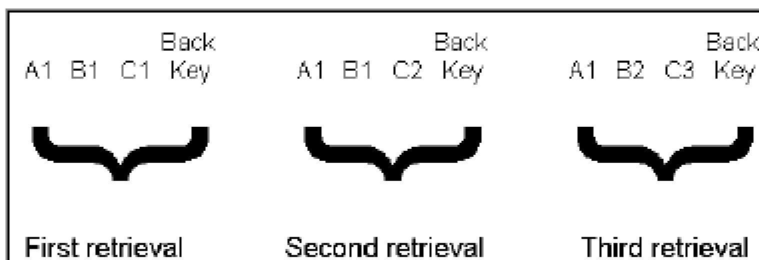
The length of the work area may differ from the size if the REPEAT option is used with the NEX and FST commands.

Defining the Work Area for a File With Descendant Segments

When you declare a work area, you must make it large enough to contain every field in the show list, no matter how many fields are retrieved. This is particularly important when using the REPEAT option to retrieve a root segment and its children. Issuing a REPEAT with a NEX or FST command is, in effect, equivalent to reissuing the retrieval command. All the fields in the show list, beginning with the first field, are retrieved. For example, assume you have a three-segment file:



If you use *nrepeat* in a NEX or FST command to retrieve all three instances of Segment C, HLI retrieves all three segments three times. You must allocate your work space as shown:



When a single record is retrieved with NEX or FST, the *backkey* (address of the data record) is placed in the FCB at word 16. However, when multiple records are retrieved in a single call, the backkey is present for each record returned, as shown in the diagram (the length of each back key is eight bytes).

Declaring Multiple FCBs

You can declare up to 4096 FCBs. Files that will be open simultaneously must have separate FCBs.

Multiple FCBs can be declared as elements in an array or as single arrays with unique names. Each FCB requires 200 bytes and is identifiable either as an element within an array, such as FCB(n), or by a unique name, such as FCB-*fn*.

```
FCBxxx(n)
```

where:

xxx

Is a number that uniquely identifies each separate FCB.

n

Represents the bytes in the array.

In COBOL, you can describe up to each FCB as a level 01 structure. You give each FCB a name in the format

```
name-fcb
```

where:

name

Is a user-assigned name.

In PL/1, you declare multiple FCBs as part of the same structure in the format

```
FCB(n)
```

where:

n

Is the number of FCBs.

You then refer to individual FCBs in the format:

```
FCB(n).fieldname
```

In C, you can assign any name to the FCB structure. You then refer to the individual FCBs in the format

```
fcbname.fieldname
```

For example, for the filename field of the FCB named *empfcb*:

```
empfcb.filename
```

Using HLI

This chapter illustrates the use of HLI with C, FORTRAN, COBOL, and PL/1.

HLI programs will generally work when the library path is adjusted to newer releases, or they are linked to the latest FOCUS load library. However, it is always a good idea to test thoroughly or consider rebuilding the application

Writing HLI Programs

Properties of Master Files for Use With HLI

- Field names can be a maximum of 12 characters.
- File and segment names can be a maximum of 8 characters.

C Program Considerations

If your HLI program is written in C, you must declare the variables required by the HLI Interface.

One session handle is required and one HLI handle for each machine that the program will access. For local databases, only one HLI handle is needed. For programs that access data on multiple FOCUS Database servers, one HLI handle is required for each of the servers.

In addition, your program must open the session connection and the HLI connection before issuing HLI commands, and must close those connections before ending. Before opening these connections, you must declare the following local variables:

- The fcb name.
- A session handle (pointer to the session connection).
- An HLI handle (pointer to the HLI engine connection). You then reference this handle

in all HLI calls.

Include the HLI Header File in a C Program

In addition to the standard header files, issue the following include command for the HLI header file:

```
#include "edahli.h"
```

Declare the Fcb Name Variable

```
t_edahli_fcb    fcb;
```

where:

fcb

Is the name that you assign to the fcb structure in your program. If your program uses multiple FCBs, you need one declaration for each.

Declare a Session Handle and Issue the Connection Call

Prior to issuing HLI commands, issue the following commands to declare the session handle, open the session connection, and check whether the connection was made successfully.

```
t_eda_handle    *sHandle = NULL;iwayHandle = edaOpen(trace_option);
     if(sHandle == NULL) {
printf("Failed to connect to session");
       return -1;
     }
```

where:

sHandle

Is the name that you assign to the session handle.

trace_option

Can be one of the following:

- **EDA_TRACE_AUTO.** Automatically determines whether the traces have been activated.
- **EDA_TRACE_ON.** Turns tracing on.
- **EDA_TRACE_OFF.** Turns tracing off.

Declare an HLI Handle and Issue the Connection Call

Prior to issuing HLI commands, and after opening a session connection, issue the following commands to declare an HLI handle, open a connection to the HLI engine, and check whether the connection was made successfully.

```
t_edahli_handle *hliHandle = NULL;hliHandle = edahliOpen(sHandle,
       ac > 1 ? EDAHLI_DESTINATION_SINK : EDAHLI_DESTINATION_LOCAL);
     if(hliHandle == NULL) {
       printf("Failed to connect to session/HLI engine");
       goto L_cleanup;
     }
```

where:

hliHandle

Is the name that you assign to the HLI handle. If your program uses multiple connections to HLI, you need one handle for each connection.

sHandle

Is the name assigned to the session handle.

L_cleanup

Is the part of the program that closes the WebFOCUS Reporting Server connection.

Close the HLI Connection

After issuing all HLI calls, and prior to closing the WebFOCUS Reporting Server connection, issue the following command to close the HLI connection:

```
edahliClose(hliHandle);
```

hliHandle

Is the name assigned to the HLI handle.

Call to Close the Session Connection

After closing all HLI connections, issue the following command to close the session connection:

```
edaClose(sHandle);
```

sHandle

Is the name assigned to the session handle.

Sample HLI Programs

Sample programs follow and are presented as simply as possible for ease of explanation. Numbers to the left of the lines of code refer to notes in the sections that describe the operations performed in the examples.

The sample programs perform the following basic HLI functions:

- Initializing the FCB.

- Opening the file.
- Using a show list.
- Locating records.
- Retrieving segments with a backkey.
- Changing, adding, and deleting data.
- Closing the file.

Master File Used With the Sample Programs

The Master File used in the examples follows:

```
FILENAME=EMP,SUFFIX=FOC
SEGNAME=ONE,SEGTYPE=S1
     FIELDNAME=EMPNO,ALIAS=EMPNUM,FORMAT=I5,$
     FIELDNAME=NAME,ALIAS=,FORMAT=A20,$
SEGNAME=TWO,PARENT=ONE,SEGTYPE=SH1
     FIELDNAME=DATE,ALIAS=,FORMAT=I6YMD,$
     FIELDNAME=SALARY,ALIAS=PAY,FORMAT=D12.2M,$
```

Sample C Program

```
1.    #include <stdlib.h>
2.    #include <stdio.h>
3.    #include <string.h>
4.    #include "edahli.h"
      /**
       * Database record, as per MFD. SALARY data type has to be aligned
       * to 8-byte boundary, hence padding member has to be inserted
       * between date and salary fields.
      */
5.    typedef struct {
6.       int    empno;
7.       char   name[20];
8.       int    date;
9.       char   padding[4];
10.      double salary;
11.   } t_record;
```

```
       /**
       * Symbolic definitions for database segment names: blank-padded
8-
       * char long.
       */
12.    #define SEG_SYS "SYSTEM  "
13.    #define SEG_ONE "ONE     "
14.    #define SEG_TWO "TWO     "
      /**
      * Symbolic definitions of the database field names: blank-padded
      * 12-char long.
      */
15.   #define FLD_EMPNO  "EMPNO       "
16.   #define FLD_NAME   "NAME        "
17.   #define FLD_DATE   "DATE        "
18.   #define FLD_SALARY "SALARY      "
      /**
      * Local function prototypes.
      */
19.   static void doInsert(t_edahli_handle *hli, t_edahli_fcb *pFcb);
20.   static void doList(t_edahli_handle *hli, t_edahli_fcb *pFcb);
21.   static void doDelete(t_edahli_handle *hli, t_edahli_fcb *pFcb);
22.   static void doChange(t_edahli_handle *hli, t_edahli_fcb *pFcb);
23.   int main(int ac, char **av) {
      /* Fieldname list for SHO command */
24.      char *names =  FLD_EMPNO FLD_NAME FLD_DATE FLD_SALARY;
25.      int namesL  = 4; /* How many fields we pass to SHO */
```

```
26.      t_eda_handle *edah;
27.      t_edahli_handle *hli;
28.      t_edahli_fcb fcb;
29.      int rc;
        /* Standard initialization of the FCB */
30.      memset(&fcb, '\0', sizeof(fcb));
31.      memset(&fcb, ' ', EDAHLI_FCB_INIT_SIZE);
         /**
         * Put file name into FCB. filetype and
         * filemode should be left blank.
         */
32.      memcpy(fcb.filename, "EMP", 3);
33.      memcpy(fcb.filetype, "     ", 5);
34.      memcpy(fcb.filemode, " ", 1);
         /**
         * Open session handle
         */
35.      edah = edaOpen(EDA_TRACE_AUTO);
36.      if(edah == NULL) {
```

```
37.         printf("Failed to open eda handle\n");
38.         return -1;
39.      }
         /**
         * Connect to the HLI engine.
         */
40.      hli = edahliOpen(edah, EDAHLI_DESTINATION_LOCAL);
41.      if(hli == NULL) {
42.         printf("Failed to open hli handle\n");
43.         edaClose(edah);
44.         return -1;
45.      }
         /**
         * Open file and check status.
         */
46.      rc = edahliCall(hli, 2, "OPN ", &fcb);
47.      if(rc != 0 || fcb.status != 0) {
48.         printf("OPEN ERROR: rc=%ld, status=%ld\n", rc, fcb.status);
49.         edahliClose(hli);
50.         edaClose(edah);
51.         return -1;
52.      }
```

```
         /**
         * Set up show field names.
         */
53.      rc = edahliCall(hli, 4, "SHO ", &fcb, names, &namesL);
54.      if(rc != 0 || fcb.status != 0) {
55.         printf("SHOW ERROR: rc=%ld, status=%ld\n", rc, fcb.status);
56.         edahliCall(hli, 2, "CLO ", &fcb);
57.         edahliClose(hli);
58.         edaClose(edah);
59.         return -1;
60.      }
         /**
         * Main option menu.
         */
61.      while(1) {
62.         int ians = 0;
63.         printf("Enter 1 to add a new date and salary\n");
64.         printf("Enter 2 to change a salary\n");
65.         printf("Enter 3 to delete a salary\n");
66.         printf("Enter 4 to print salary information\n");
67.         printf("Enter 0 to exit\n");
68.         scanf("%d", &ians);
69.         if(ians < 0 || ians > 4) {
70.            printf("A bad code of %d was given - try again\n", ians);
```

```
71.             continue;
72.          }
73.          if(ians == 0) break;
             /**
             * Enter employee number and check for existence which also
             * establishes position in the database
             */
74.          while(1) {
75.             t_record testlt, wkarea;
76.             char testrl[4 * 4]; /* 4 bytes per SHO field */
77.             int testL = 1;      /* single test pass to FST */
78.             printf("Enter the employee number or 0 for menu\n");
79.             scanf("%d", &testlt.empno);
80.             if(testlt.empno == 0) break;
81.             memset(&testrl[0], ' ', sizeof(testrl));
82.             memcpy(&testrl[0], "EQ   ", 4);
```

```
83.             rc = edahliCall(hli, 8, "FST ", &fcb,
84.                  &wkarea, SEG_ONE, SEG_SYS, &testL, testrl, &testlt);
85.             if(rc == 0 && fcb.status == 0) {
                   /**
                   * Branch according to option selected
                   */
86.                printf("Employee name is %-10.10s\n", wkarea.name);
87.                switch(ians) {
88.                case 1:
89.                   doInsert(hli, &fcb);
90.                   break;
91.                case 2:
92.                   doChange(hli, &fcb);
93.                   break;
94.                case 3:
95.                   doDelete(hli, &fcb);
96.                   break;
97.                case 4:
98.                   doList(hli, &fcb);
99.                   break;
100.               }
101.               break;
102.            }
103.            printf("Employee %d not found\n", testlt.empno);
104.         }
105.      }
          /* Cleanup */
106.      edahliCall(hli, 2, "CLO ", &fcb);
107.      edahliClose(hli);
108.      edaClose(edah);
```

```
109.  }
       /**
       * Add a new date and salary.
       */
110.  static void doInsert(t_edahli_handle *hli, t_edahli_fcb *pFcb) {
111.     t_record wkarea;
112.     int c__2 = 2; /* Option 2 makes INP reject duplicate keys */
113.     int rc;
114.     while(1) {
115.        printf("Enter date in form YYMMDD or 0 for menu\n");
116.        scanf("%d", &wkarea.date);
117.        if(wkarea.date == 0) break;
118.        printf("Enter salary with decimal\n");
119.        scanf("%lf9.2", &wkarea.salary);
```

```
120.        rc = edahliCall(hli, 3, "INP ", pFcb, wkarea, "TWO     ",
121.                         &c__2);
122.        if(rc == 0 && pFcb->status == 0) {
123.           break;
124.        }
125.        printf("***** ERROR ***** Date of %6d already exists\n");
126.     }
127.  }
      /**
      * Change a salary but first display existing salary.
      */
128.  static void doChange(t_edahli_handle *hli, t_edahli_fcb *pFcb) {
129.     t_record testlt, wkarea, chalit;
130.     char testrl[4 * 4]; /* 4 bytes per SHO field */
131.     int testL = 1; /* just single test passed to NEX */
132.     char charel[4 * 4]; /* 4 bytes per SHO field */
133.     int chaL = 1; /* changing just one field */
134.     int rc;
135.     while(1) {
136.       printf("Enter date in form YYMMDD or 0 for menu\n");
137.       scanf("%d", &testlt.date);
138.       if(testlt.date == 0) return;
139.        memset(testrl, ' ', sizeof(testrl));
140.        memcpy(&testrl[4 * 2], "EQ  ", 4);
141.        rc = edahliCall(hli, 8, "NEX ", pFcb, &wkarea,
142.                        SEG_TWO, SEG_ONE, &testL, testrl, &testlt);
143.        if(rc == 0 && pFcb->status == 0) {
144.           break;
145.        }
146.        printf("***** ERROR ***** Date of %6d not found\n");
147.     }
148.     printf("Existing salary is %12.2f. Enter new salary",
```

```
149.            wkarea.salary);
150.      scanf("%9.2lf", &chalit.salary);
151.      memset(charel, ' ', sizeof(charel));
152.      memcpy(&charel[16], "EQ  ", 4);
```

```
153.      rc = edahliCall(hli, 7, "CHA ", pFcb,
154.                      &wkarea, SEG_TWO, SEG_ONE, charel, &chalit);
155.      if(rc != 0 || pFcb->status != 0) {
156.         printf("Error in change: rc=%d, status=%d\n", rc,
157.                pFcb->status);
158.      }
159.   }
       /**
       * Delete a segment TWO instance given a date.
       */
160.   static void doDelete(t_edahli_handle *hli, t_edahli_fcb *pFcb) {
161.      t_record teslit, wkarea;
162.      char testrl[4 * 4]; /* 4 bytes per SHO field */
163.      int testL = 1; /* just single test passed to NEX */
164.      int rc;
165.      while(1) {
166.         printf("Enter date in form YYMMDD or 0 for menu\n");
167.         scanf("%d", &teslit.date);
168.         if(teslit.date == 0) return;
169.         memset(testrl, ' ', sizeof(testrl));
170.         memcpy(&testrl[8], "EQ   ", 4);
171.         rc = edahliCall(hli, 8, "NEX ", pFcb, &wkarea,
172.                         SEG_TWO, SEG_ONE, &testL, testrl, &teslit);
173.         if(rc == 0 && pFcb->status == 0) {
174.            break;
175.         }
176.         printf("***** ERROR ***** Date of %6d not found\n");
177.      }
178.      rc = edahliCall(hli, 3, "DEL", pFcb, SEG_TWO);
179.      if(rc != 0 || pFcb->status != 0) {
180.         printf("***** ERROR ***** Segment not deleted: rc=%d,
181.                status=%d\n", rc, pFcb->status);
182.      } else {
183.         printf("Segment deleted\n");
184.      }
185.   }
```

```
       /**
       * Report by nexting thru segment TWO
       */
186.   static void doList(t_edahli_handle *hli, t_edahli_fcb *pFcb) {
```

```
187.     t_record wkarea;
188.     int c__0 = 0;
189.     int rc;
190.     while(1) {
191.        rc = edahliCall(hli, 5, "NEX ", pFcb, wkarea, SEG_TWO,
192.                          SEG_ONE, &c__0);
193.        if(rc != 0 || pFcb->status != 0) {
194.           break;
195.        }
196.        printf(" %6d  %12.2f\n", wkarea.date, wkarea.salary);
197.     }
198.  }
```

Sample FORTRAN Program

```
1.          IMPLICIT INTEGER (A-Z)
2.          DIMENSION FCB(50)
3.          REAL*8 FN,FT,SALARY,SALNEW
4.          INTEGER*4 STATUS, DATE, NAME(5)
5.          EQUIVALENCE (FCB(1),FN), (FCB(3),FT), (FCB(5),FM),
6.          *           (FCB(24),STATUS)
7.          DATA FCB /'EMP    ', '         ', '    ', 17*' ', 28*0/
  C
  C         SET UP NAMES AREA
  C
8.          DIMENSION NAMES (3,4)
9.          DATA NAMES /9*'    '/
10.         DATA NAMES(1,1)/'EMPN'/,NAMES(2,1)/'0   '/,NAMES(3,1)/'   '/,
11.        *     NAMES(1,2)/'NAME'/,NAMES(2,2)/'    '/,NAMES(3,2)/'   '/,
12.        *     NAMES(1,3)/'DATE'/,NAMES(2,3)/'    '/,NAMES(3,3)/'   '/,
13.        *     NAMES(1,4)/'SALA'/,NAMES(2,4)/'RY  '/,NAMES(3,4)/'   '/
   C
   C        SET UP WORK AREA
   C
14.         DIMENSION WKAREA(10)
15.         EQUIVALENCE (WKAREA(2),NAME(1)), (WKAREA(7),DATE),
16.        * (WKAREA(9),SALARY)
   C
   C        SET UP TEST ARRAY
   C
17.         DIMENSION TESTRL(4),TESTLT(10),CHAREL(4),CHALIT(10)
18.         EQUIVALENCE (CHALIT(9),SALNEW)
19.         DATA TESTRL /4*'    '/, CHAREL /4*'    '/
```

```
   C
   C        SET UP HLI COMMANDS
   C
20.         DATA FST/'FST '/, NEXT/'NEX '/, EQ /'EQ  '/, BLANK/'    '/
21.         DATA OPN/'OPN '/, SHO /'SHO '/, INP/'INP '/, CHA /'CHA '/
22.         DATA DEL/'DEL '/, CLO /'CLO '/
   C
   C        SET UP SEGMENT NAMES
   C
23.         REAL*8 ONE/'ONE     '/, TWO/'TWO     '/, SYSTEM/'SYSTEM  '/
   C
   C        OPEN FILE AND CHECK STATUS
   C
24.         CALL FOCUS (OPN,FCB)
25.         IF (STATUS.EQ.0) GO TO 20
26.         WRITE (6,10) STATUS
27.     10  FORMAT (1X, 'OPEN ERROR', I5)
28.         RETURN
   C
   C        SET UP SHOW FIELD NAMES
   C
29.     20  CALL FOCUS (SHO,FCB,NAMES,4)
30.         IF (STATUS.EQ.0) GO TO 40
31.         WRITE (6,30) STATUS
32.     30  FORMAT (1X, 'SHOW ERROR', I5)
33.         RETURN
```

```
   C
   C        MAIN OPTION MENU
   C
34.     40  WRITE (6,50)
35.     50  FORMAT (/1X,'ENTER 1 TO ADD A NEW DATE AND SALARY'/,
36.        1         1X,'ENTER 2 TO CHANGE A SALARY',/
37.        2         1X,'ENTER 3 TO DELETE A SALARY',/
38.        3         1X,'ENTER 4 TO PRINT SALARY INFORMATION',/
39.        4         1X,'ENTER <RETURN> TO EXIT')
40.         READ (5,60,END=999) IANS
41.     60  FORMAT (I1)
42.         IF(IANS.LT.1)IANS=5
43.         IF(IANS.GT.4)IANS=5
44.         GOTO(80,80,80,80,65),IANS
45.     65  WRITE (6,70) IANS
46.     70  FORMAT (1X,'A BAD CODE WAS GIVEN - TRY AGAIN')
47.         GO TO 40
   C
```

```
   C        ENTER EMPLOYEE NUMBER AND CHECK FOR EXISTENCE WHICH ALSO
   C        ESTABLISHES POSITION IN THE DATABASE
   C
48.      80  WRITE (6,90)
49.      90  FORMAT (/,1X,'ENTER THE EMPLOYEE NUMBER OR 0 FOR MENU')
50.          READ (5,100,END=40) TESTLT(1)
51.     100  FORMAT (I5)
52.          IF (TESTLT(1).EQ.0) GO TO 40
53.          TESTRL(1)=EQ
54.          CALL FOCUS (FST, FCB, WKAREA, ONE,SYSTEM , 1, TESTRL,
TESTLT)
55.          IF (STATUS.EQ.0) GO TO 120
56.          WRITE (6,110) TESTLT(1)
57.     110  FORMAT (/,1X,'EMPLOYEE ',I5,' NOT FOUND')
58.          GOTO 80
   C
   C        BRANCH ACCORDING TO OPTION SELECTED
   C
59.     120  WRITE(6,130) NAME
60.     130  FORMAT (/1X, 'EMPLOYEE NAME IS ',5A4)
61.          GO TO (1000,2000,3000,4000),IANS
```

```
   C
   C        ADD A NEW DATE AND SALARY
   C
62.     1000 WRITE (6,1100)
63.     1100 FORMAT (/,1X,'ENTER DATE IN FORM YYMMDD OR 0 FOR MENU')
64.          READ (5,1200) DATE
65.     1200 FORMAT(I6)
66.          IF (DATE.EQ.0) GO TO 40
67.          WRITE (6,1300)
68.     1300 FORMAT (1X,'ENTER SALARY WITH DECIMAL')
69.          READ (5,1400) SALARY
70.     1400 FORMAT(F9.2)
71.          CALL FOCUS (INP,FCB,WKAREA,TWO,2)
72.          IF (STATUS.EQ.0) GO TO 1000
73.          WRITE (6,1500) DATE
74.     1500 FORMAT (1X,'***** ERROR ***** DATE OF ',I6,' ALREADY
EXISTS')
75.          GO TO 1000
   C
   C        CHANGE A SALARY BUT FIRST DISPLAY EXISTING SALARY
   C
76.     2000 WRITE (6,1100)
77.          READ (5,1200) TESTLT(7)
78.          IF (TESTLT(7).EQ.0) GO TO 40
```

```
79.         TESTRL(1)=BLANK
80.         TESTRL(3)=EQ
81.         CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,1,TESTRL,TESTLT)
82.         TESTRL(3)=BLANK
83.         IF (STATUS.EQ.0) GO TO 2200
84.         WRITE (6,2100)TESTLT(7)
85.    2100 FORMAT (1X, ' ***** ERROR ***** DATE OF ',I6, ' NOT FOUND')
86.         GO TO 2000
87.    2200 WRITE (6,2300) SALARY
88.    2300 FORMAT (1X,'EXISTING SALARY IS ',F12.2/,' ENTER NEW SALARY')
89.         READ (5,1400) SALNEW
90.         CHAREL(4)=EQ
91.         CALL FOCUS (CHA,FCB,WKAREA,TWO,ONE,1,CHAREL,CHALIT)
92.         IF (STATUS.EQ.0) GO TO 80
93.         WRITE (6,2400) STATUS
94.    2400 FORMAT (1X,'ERROR IN CHANGE STATUS IS ',I6)
95.         GO TO 80
```

```
   C
   C        DELETE A SEGMENT TWO INSTANCE GIVEN A DATE
   C
96.    3000 WRITE (6,1100)
97.         READ (5,1200) TESTLT(7)
98.         IF (TESTLT(7).EQ.0) GO TO 40
99.         TESTRL(1)=BLANK
100.        TESTRL(3)=EQ
101.        CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,1,TESTRL,TESTLT)
102.        TESTRL(3)=BLANK
103.        IF (STATUS.EQ.0) GO TO 3100
104.        WRITE (6,2100)TESTLT(7)
105.        GO TO 3000
106.   3100 CALL FOCUS (DEL,FCB,TWO)
107.        IF (STATUS.EQ.0) GOTO 3300
108.        WRITE (6,3200) STATUS
109.   3200 FORMAT (1X,'**** ERROR **** SEGMENT NOT DELETED STATUS=',I6)
110.        GO TO 3000
111.   3300 WRITE (6,3400)
112.   3400 FORMAT (1X,'SEGMENT DELETED')
113.        GO TO 3000
    C
    C       REPORT BY NEXTING THROUGH SEGMENT TWO
    C
114.   4000 CALL FOCUS (NEXT, FCB, WKAREA,TWO,ONE,0)
115.        IF (STATUS.NE.0) GOTO 80
116.        WRITE (6,4100) DATE, SALARY
117.   4100 FORMAT (/,1X,I6,2X,F12.2)
```

```
118.        GOTO 4000
    C
    C       EXIT PROGRAM AND CLOSE DATABASE
    C
119.    999  CALL FOCUS (CLO,FCB)
120.         IF(STATUS.EQ.0)GOTO 9999
121.         WRITE(6,9990) STATUS
122.    9990 FORMAT(1X,'ERROR IN CLOSE IS ',I6)
123.    9999 RETURN
124.         END
```

Sample COBOL Program

```
1.   IDENTIFICATION DIVISION. 37.       05  OUT-REST.
38.           10  OUT-DATE      PIC X(6).
39.           10  FILLER        PIC XX.
40.           10  OUT-SALARY    PIC ZZZ,ZZZ,ZZZ,ZZZ.99.
41.           10  FILLER        PIC XX.
42.           10  OLD-SAL       PIC ZZZ,ZZZ,ZZZ,ZZZ.99.
43.           10  FILLER        PIC XX.
44.           10  OUT-MSG       PIC X(50).
   ***************************************************
   *                 HLI WORK AREAS                  *
   ***************************************************
45.   01  LIST-OF-SEGNAMES.
46.       02  SYSSEG    PIC X(8) VALUE 'SYSTEM'.
47.       02  ONESEG    PIC X(8) VALUE 'ONE    '.
48.       02  TWOSEG    PIC X(8) VALUE 'TWO    '.
49.   01  FCB.
50.       02  FCB-FN     PIC X(8) VALUE 'EMP     '.
51.       02  FCB-FT     PIC X(8) VALUE '        '.
52.       02  FCB-FM     PIC X(4) VALUE '  '.
53.       02  FILLER     PIC X(48) VALUE SPACES.
54.       02  FCB-ECHO   PIC X(4) VALUE 'ECHO'.
55.       02  FILLER     PIC X(20) VALUE SPACES.
56.       02  FCB-STATUS PIC S9(5) COMP.
57.       02  FILLER     PIC X(104) VALUE SPACES.
58.   01  CMDS.
59.       02  OPNCMD     PIC X(4) VALUE 'OPN '.
60.       02  CLOCMD     PIC X(4) VALUE 'CLO '.
61.       02  CHACMD     PIC X(4) VALUE 'CHA '.
62.       02  NEXCMD     PIC X(4) VALUE 'NEX '.
63.       02  FSTCMD     PIC X(4) VALUE 'FST '.
64.       02  DELCMD     PIC X(4) VALUE 'DEL '.
```

```
65.       02  SAVCMD     PIC X(4) VALUE 'SAV '.
66.       02  SHOCMD     PIC X(4) VALUE 'SHO '.
67.       02  INPCMD     PIC X(4) VALUE 'INP '.
68.   01  NAMES-AREA.
69.       02  FIELD-0001  PIC X(12) VALUE 'EMPNO       '.
70.       02  FIELD-0002  PIC X(12) VALUE 'NAME        '.
71.       02  FIELD-0003  PIC X(12) VALUE 'DATE        '.
72.       02  FIELD-0004  PIC X(12) VALUE 'SALARY      '.
73.   01  NUMB0            PIC S9(9) COMP VALUE 0.
74.   01  NUMB1            PIC S9(9) COMP VALUE 1.
75.   01  NUMB2            PIC S9(9) COMP VALUE 2.
76.   01  NUMB3            PIC S9(9) COMP VALUE 3.
77.   01  NUMB4            PIC S9(9) COMP VALUE 4.
78.   01  NULL             PIC X(8) VALUE SPACES.
79.   01  EOF-FLAG         PIC S9 COMP VALUE 0.
```

```
80.   01  WRKAREA.
81.       02  EMPNO        PIC S9(9) COMP.
82.       02  EMPNAME      PIC X(20) VALUE SPACES.
83.       02  EMPDATE      PIC S9(6) COMP.
84.       02  FILLER       PIC X(4).
85.       02  SALARY       COMP-2.
86.   01  TESTREL.
87.       02  REL-NO             PIC X(04) VALUE SPACES.
88.       02  REL-NAME           PIC X(04) VALUE SPACES.
89.       02  REL-DATE           PIC X(04) VALUE SPACES.
90.       02  REL-SAL            PIC X(04) VALUE SPACES.
91.   01  TESTLIT.
92.       02  LIT-EMPNO       PIC S9(9) COMP.
93.       02  LIT-NAME        PIC X(20) VALUE SPACES.
94.       02  LIT-DATE        PIC S9(6) COMP.
95.       02  FILLER          PIC X(4).
96.       02  LIT-SALARY      COMP-2.
97.   PROCEDURE DIVISION.
98.   AAAA-MAIN-PROGRAM.
99.       PERFORM A000-INIT THRU A999-EXIT.
100.       IF FCB-STATUS EQUAL 0
101.          PERFORM B100-READ THRU B999-EXIT
102.          UNTIL EOF-FLAG EQUAL 9.
103.       PERFORM ZZZZ-CLOSES.
104.       STOP RUN.
105.   A000-INIT.
106.       OPEN INPUT TRANS OUTPUT OUTFI.
107.       CALL 'FOCUS' USING OPNCMD FCB NUMB0.
108.       IF FCB-STATUS NOT EQUAL 0
109.          MOVE SPACES TO OUT-REC
110.          MOVE 'ERROR IN OPEN' TO OUT-MSG
```

```
111.          WRITE OUTREC FROM OUT-REC
112.          GO TO A999-EXIT.
113.       CALL 'FOCUS' USING SHOCMD FCB NAMES-AREA NUMB4.
114.       IF FCB-STATUS NOT EQUAL 0
115.          MOVE SPACES TO OUT-REC
116.          MOVE 'ERROR IN SHOW' TO OUT-MSG
117.          WRITE OUTREC FROM OUT-REC
118.          GO TO A999-EXIT.
119.   A999-EXIT. EXIT.
```

```
120.   B100-READ.
121.       READ TRANS AT END MOVE 9 TO EOF-FLAG GO TO B999-EXIT.
122.       IF TRANS-TYPE LESS THAN 1 OR TRANS-TYPE GREATER 4
123.          MOVE SPACES TO OUT-REC
124.          MOVE TRANS-REC TO OUT-REC
125.          MOVE 'ERROR IN TRANSACTION CODE' TO OUT-MSG
126.          WRITE OUTREC FROM OUT-REC
127.          GO TO B999-EXIT.
128.       MOVE SPACES TO OUT-REC MOVE TR-EMPNO TO OUT-EMPNO.
129.       MOVE SPACES TO TESTREL. MOVE 'EQ' TO REL-NO.
130.       MOVE TR-EMPNO TO LIT-EMPNO.
131.       CALL 'FOCUS' USING FSTCMD FCB WRKAREA ONESEG SYSSEG
132.          NUMB1 TESTREL TESTLIT.
133.       IF FCB-STATUS EQUAL 1
134.          MOVE 'NOT ON FILE' TO OUT-MSG
135.          WRITE OUTREC FROM OUT-REC
136.          GO TO B999-EXIT.
137.       IF FCB-STATUS GREATER 0
138.          MOVE 'ERROR IN READ' TO OUT-MSG
139.          WRITE OUTREC FROM OUT-REC
140.          MOVE ZERO TO FCB-STATUS
141.          GO TO B999-EXIT.
142.       MOVE LIT-NAME TO OUT-NAME.
143.       IF TRANS-TYPE EQUAL 1
144.          PERFORM C100-ADD THRU C999-EXIT
145.          GO TO B999-EXIT
146.       ELSE IF TRANS-TYPE EQUAL 2
147.          PERFORM D100-CHG THRU D999-EXIT
148.          GO TO B999-EXIT
149.       ELSE IF TRANS-TYPE EQUAL 3
150.          PERFORM E100-DEL THRU E999-EXIT
151.          GO TO B999-EXIT
152.       ELSE PERFORM F100-PRT THRU F999-EXIT UNTIL FCB-STATUS
153.          GREATER 0 MOVE 0 TO FCB-STATUS.
154.   B999-EXIT. EXIT.
```

```
155.   C100-ADD.
156.       MOVE SPACES TO TESTLIT MOVE TR-EMP-DATE TO EMPDATE.
157.       MOVE TR-EMP-DATE TO OUT-DATE.
158.       MOVE TR-SALARY TO SALARY.
159.       MOVE TR-SALARY TO OUT-SALARY.
160.       CALL 'FOCUS' USING INPCMD FCB WRKAREA TWOSEG NUMB2.
161.       IF FCB-STATUS NOT EQUAL 0
162.          MOVE 'DATE ALREADY IN DATABASE' TO OUT-MSG
163.          ELSE MOVE 'RECORD INCLUDED' TO OUT-MSG.
164.       WRITE OUTREC FROM OUT-REC.
165.   C999-EXIT. EXIT.
166.   D100-CHG.
167.       PERFORM G100-GET-DATE THRU G999-EXIT.
168.       MOVE SALARY TO OLD-SAL.
169.       MOVE TR-SALARY TO LIT-SALARY. MOVE 'EQ' TO REL-SAL.
170.       MOVE SPACES TO REL-DATE.
171.       CALL 'FOCUS' USING CHACMD FCB WRKAREA TWOSEG NULL NUMB1
172.          TESTREL TESTLIT.
173.       IF FCB-STATUS NOT EQUAL 0
174.          MOVE 'ERROR CHANGING DATE' TO OUT-MSG
175.          ELSE MOVE 'RECORD CHANGED' TO OUT-MSG.
176.       WRITE OUTREC FROM OUT-REC.
177.   D999-EXIT. EXIT.
178.   E100-DEL.
179.       PERFORM G100-GET-DATE THRU G999-EXIT.
180.       MOVE SALARY TO OLD-SAL.
181.       CALL 'FOCUS' USING DELCMD FCB TWOSEG.
182.       IF FCB-STATUS NOT EQUAL 0
183.          MOVE 'ERROR CHANGING DATE' TO OUT-MSG
184.          ELSE MOVE 'SEGMENT DELETED' TO OUT-MSG.
185.       WRITE OUTREC FROM OUT-REC.
186.   E999-EXIT. EXIT.
```

```
187.   F100-PRT.
188.       CALL 'FOCUS' USING NEXCMD FCB WRKAREA TWOSEG ONESEG NUMB0.
189.       IF FCB-STATUS GREATER 1
190.          MOVE 'ERROR READING FILE' TO OUT-MSG
191.          WRITE OUTREC FROM OUT-REC
192.          GO TO F999-EXIT.
193.       IF FCB-STATUS EQUAL 1
194.          GO TO F999-EXIT.
195.       MOVE EMPDATE TO OUT-DATE. MOVE SALARY TO OUT-SALARY.
196.       WRITE OUTREC FROM OUT-REC.
197.   F999-EXIT. EXIT.
198.   G100-GET-DATE.
199.       MOVE SPACES TO TESTLIT MOVE TR-EMP-DATE TO LIT-DATE.
200.       MOVE TR-EMP-DATE TO OUT-DATE.
```

```
201.       MOVE TR-SALARY TO OUT-SALARY.
202.       MOVE SPACES TO TESTREL MOVE 'EQ' TO REL-DATE.
203.       CALL 'FOCUS' USING FSTCMD FCB WRKAREA TWOSEG ONESEG
204.          NUMB1 TESTREL TESTLIT.
205.       IF FCB-STATUS NOT EQUAL 0
206.          MOVE 'DATE NOT IN DATABASE' TO OUT-MSG
207.          WRITE OUTREC FROM OUT-REC GO TO D999-EXIT.
208.   G999-EXIT. EXIT.
209.   ZZZZ-CLOSES.
210.       CALL 'FOCUS' USING CLOCMD FCB.
211.       IF FCB-STATUS NOT EQUAL 0
212.          MOVE SPACES TO OUT-REC
213.          MOVE 'ERROR IN CLOSE' TO OUT-MSG
214.          WRITE OUTREC FROM OUT-REC.
215.       CLOSE TRANS OUTFI.
```

Sample PL/1 Program

```
1.   BOBPLI: PROCEDURE OPTIONS(MAIN); 26.  DCL 1 TESTLT STATIC,
27.         2 EMPTEST   FIXED BIN(31),
28.         2 NAMETEST  CHAR(20),
29.         2 DATETEST  FIXED BIN(31),
30.         2 DUMMYT    FIXED BIN(31),
31.         2 SALTEST   FLOAT BIN(53);
```

```
32.  DCL 1 TESTRL STATIC,
33.         2 EMPREL    CHAR(4) INIT(' '),
34.         2 NAMEREL   CHAR(4) INIT(' '),
35.         2 DATEREL   CHAR(4) INIT(' '),
36.         2 SALREL    CHAR(4) INIT(' ');
37.  DCL 1 CHALIT STATIC,
38.         2 CEMPTEST  FIXED BIN(31),
39.         2 CNAMETEST CHAR(20),
40.         2 CDATETEST FIXED BIN(31),
41.         2 CDUMMYT   FIXED BIN(31),
42.         2 CSALTEST  FLOAT BIN(53);
43.  DCL 1 CHAREL STATIC,
44.         2 CEMPREL   CHAR(4) INIT(' '),
45.         2 CNAMEREL  CHAR(4) INIT(' '),
46.         2 CDATEREL  CHAR(4) INIT(' '),
47.         2 CSALREL   CHAR(4) INIT(' ');
     /*
```

```
            SET UP FOCUS COMMANDS
      */
48.   DCL    FST   CHAR(4) INIT('FST'),
49.          NEXT  CHAR(4) INIT('NEX'),
50.          EQ    CHAR(4) INIT('EQ'),
51.          BLANK CHAR(4) INIT(' '),
52.          OPN   CHAR(4) INIT('OPN'),
53.          SHO   CHAR(4) INIT('SHO'),
54.          INP   CHAR(4) INIT('INP'),
55.          CHA   CHAR(4) INIT('CHA'),
56.          DEL   CHAR(4) INIT('DEL'),
57.          CLO   CHAR(4) INIT('CLO');
```

```
  /*
            SET UP SEGMENT NAMES
      */
58.   DCL    ONE         CHAR(8) INIT('ONE'),
59.          TWO         CHAR(8) INIT('TWO'),
60.          SYSTEM      CHAR(8) INIT('SYSTEM');
      /*
            SET UP OTHER VARIABLES
      */
61.   DCL    FLAG        BIT(1) INIT('1'B),
62.          IANS        FIXED BIN(15),
63.          NUMBER      FIXED BIN(31);
      /*
            OPEN FILE AND CHECK STATUS
      */
64.      CALL FOCUS (OPN, FCB);
65.      IF STATUS ]= 0 THEN DO;
66.         PUT LIST ('OPEN ERROR ',STATUS);
67.         GO TO CLOSE;
68.      END;
      /*
            SET UP SHOW FIELD NAMES
      */
69.      NUMBER=4;
70.      CALL FOCUS (SHO, FCB, NAMES, NUMBER);
71.      IF STATUS ]= 0 THEN DO;
72.         PUT LIST ('SHOW ERROR ',STATUS);
73.         GOTO CLOSE;
74.      END;
      /*
            MAIN OPTION MENU
      */
75.   DO WHILE(FLAG);
```

```
76.  MENU:
77.    PUT EDIT ('ENTER 1 TO ADD A NEW DATE AND SALARY',
78.              'ENTER 2 TO CHANGE A SALARY',
79.              'ENTER 3 TO DELETE A SALARY',
80.              'ENTER 4 TO PRINT SALARY INFORMATION
81.              'ENTER <RETURN> TO EXIT')
82.             (SKIP(0),A(36),SKIP(0),A(26),SKIP(0),A(26),SKIP(0),
83.             A(35),SKIP(0),A(22));
84.    GET EDIT (IANS) (F(1));
85.    IF IANS = 0 THEN GOTO CLOSE;
86.    IF IANS < 1 [ IANS  4 THEN DO;
87.       PUT LIST ('A BAD CODE OF', IANS, ' WAS GIVEN - TRY AGAIN');
88.       GOTO MENU;
89.    END;
```

```
 /*
          ENTER EMPLOYEE NUMBER AND CHECK FOR EXISTENCE WHICH ALSO
          ESTABLISHES POSITION IN THE DATABASE
     */
90.   EMPLOYEE:
91.     PUT SKIP LIST ('ENTER THE EMPLOYEE NUMBER OR 0 FOR MENU');
92.     GET EDIT (EMPTEST) (F(5));
93.     IF EMPTEST = 0 THEN GOTO MENU;
94.     EMPREL=EQ;
95.     NUMBER=1;
96.     CALL FOCUS ( FST,FCB,WKAREA,ONE,SYSTEM,NUMBER,TESTRL,TESTLT );
97.     IF STATUS ]= 0 THEN DO;
98.        PUT SKIP LIST ('EMPLOYEE',EMPTEST,'NOT FOUND');
99.        GOTO EMPLOYEE;
100.       END;
     /*
           BRANCH ACCORDING TO OPTION SELECTED
     */
101.    PUT LIST ('EMPLOYEE NAME IS ',NAME);
102.    IF IANS = 1 THEN CALL ADD;
103.     ELSE IF IANS = 2 THEN CALL CHANGE;
104.      ELSE IF IANS = 3 THEN CALL DELETE;
105.      ELSE IF IANS = 4 THEN CALL PRINT;
106.    END;
     /*
          EXIT PROGRAM AND CLOSE DATABASE
     */
107.  CLOSE: CALL FOCUS (CLO, FCB);
108.    IF STATUS ]= 0 THEN DO;
109.       PUT LIST ('ERROR IN CLOSE IS', STATUS);
110.       END;
```

```
      /*
            PROCEDURE TO ADD A NEW DATE AND SALARY
      */
111.  ADD: PROC;
112.   NUMBER=2;
113.   DO WHILE (FLAG);
114.     PUT SKIP LIST ('ENTER DATE IN FORM YYMMDD OR 0 FOR MENU');
115.     GET EDIT (DATE) (F(6));
116.     IF DATE = 0 THEN RETURN;
117.     PUT LIST ('ENTER SALARY WITH DECIMAL');
118.     GET EDIT (SALARY) (F(9,2));
119.     CALL FOCUS (INP, FCB, WKAREA, TWO, NUMBER);
120.     IF STATUS ]= 0 THEN DO;
121.        PUT LIST (' ***** ERROR ***** DATE ALREADY EXISTS', DATE);
122.        END;
123.     END;
124.     RETURN;
125.   END ADD;
```

```
 /*
         PROCEDURE TO CHANGE A SALARY BUT FIRST DISPLAY EXISTING SALARY
      */
126.   CHANGE: PROC;
127.   NUMBER=1;
128.   PUT SKIP LIST ('ENTER DATE IN FORM YYMMDD OR 0 FOR NEW
EMPLOYEE');
129.     GET EDIT (DATETEST) (F(6));
130.     IF DATETEST = 0 THEN RETURN;
131.     EMPREL = BLANK;
132.     DATEREL=EQ;
133.     CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,NUMBER,TESTRL,TESTLT);
134.     DATEREL=BLANK;
135.     IF STATUS = 0 THEN DO;
136.        PUT EDIT ('EXISTING SALARY IS ',SALARY,'ENTER NEW SALARY')
137.        (A(19),F(9,2),SKIP(0),A(16));
138.        GET EDIT (CSALTEST) (F(9,2));
139.        CSALREL=EQ;
140.        CALL FOCUS (CHA,FCB,WKAREA,TWO,ONE,NUMBER,CHAREL,CHALIT);
141.           IF STATUS ]= 0 THEN DO;
142.              PUT LIST ('ERROR IN CHANGE STATUS IS ',STATUS);
143.              END;
144.        END;
145.   ELSE DO;
146.        PUT LIST ('***** ERROR ***** DATE NOT FOUND',DATETEST);
147.        END;
148.   RETURN;
```

```
149.  END CHANGE;
```

```
 /*
             PROCEDURE TO DELETE A SEGMENT TWO INSTANCE GIVEN A DATE
      */
150.   DELETE: PROC;
151.    NUMBER=1;
152.   DO WHILE (FLAG);
153.   PUT SKIP LIST ('ENTER DATE IN FORM YYMMDD OR 0 FOR NEW
EMPLOYEE');
154.     GET EDIT (DATETEST) (F(6));
155.     IF DATETEST = 0 THEN RETURN;
156.     EMPREL=BLANK;
157.     DATEREL=EQ;
158.     CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,NUMBER,TESTRL,TESTLT);
159.     DATEREL=BLANK;
160.     IF STATUS = 0 THEN DO;
161.        CALL FOCUS (DEL,FCB,TWO);
162.         IF STATUS ]= 0 THEN PUT LIST ('SEGMENT NOT DELETED
STATUS=',
163.                                         STATUS); ELSE
164.           PUT LIST ('SEGMENT DELETED');
165.        END;
166.     ELSE DO;
167.        PUT LIST ('***** ERROR DATE NOT FOUND',DATETEST);
168.     END;
169.    END;
170.    RETURN;
171.   END DELETE;
      /*
          PROCEDURE TO REPORT BY NEXTING THROUGH SEGMENT TWO
      */
172.   PRINT: PROC;
173.    NUMBER=0;
174.     DO WHILE (FLAG);
175.         CALL FOCUS (NEXT, FCB, WKAREA, TWO, ONE, NUMBER);
176.         IF STATUS ]= 0 THEN RETURN;
177.         PUT EDIT (DATE,SALARY) (SKIP(0),F(6),X(2),F(12,2));
178.     END;
179.   END PRINT;
      /*
             END MAIN PROGRAM
      */
180.   END BOBPLI;
```

Initializing the FCB

HLI must have the location of the File Communication Block before it can open the file.

Your program must:

1. Define a 200-byte area of memory for the FCB.
2. Store the needed values in the correct bytes. For example, the file name must be in bytes 1 to 8. If you want ECHO logging, you must place the characters 'ECHO' in bytes 69 through 72.

For more information about FCB layout and contents, see Initializing the File Communication Block (FCB).

Initializing the FCB in a C Program

The following code (taken from Lines 30 through 34 of the C program shown in Writing HLI Programs) declares a File Communication Block. The FCB structure is defined in the header file edahli.h. The program initializes the file name to 'EMP'. No logging is requested and the file is not on a sink machine, so those bytes are blank:

```
memset(&fcb, '\0', sizeof(fcb));
memset(&fcb, ' ', EDAHLI_FCB_INIT_SIZE);
/**
* Put file name into FCB. filetype and
* filemode should be left blank.
*/
memcpy(fcb.filename, "EMP", 3);
memcpy(fcb.filetype, " ", 5);
memcpy(fcb.filemode, " ", 1);
```

Initializing the FCB in a FORTRAN Program

The following code (taken from Lines 2 through 7 of the FORTRAN program shown in Writing HLI Programs) declares a File Communication Block. The FCB is declared as an array of 50 four-byte words. The bytes that need to be initialized by the program are also given individual names, FN for the file name and STATUS for the status code. The file name

is initialized to 'EMP'. No logging is requested and the file is not on a sink machine, so those bytes are blank:

```
      DIMENSION FCB(50)
      REAL*8 FN,FT,SALARY,SALNEW
      INTEGER*4 STATUS, DATE, NAME(5)
      EQUIVALENCE (FCB(1),FN), (FCB(24),STATUS)

      DATA FCB /'EMP     ', '        ', '    ', 17*' ', 28*0/
```

Initializing the FCB in a COBOL program

The following code (taken from Lines 49 through 57 of the COBOL program shown in Writing HLI Programs) declares a File Communication Block. The FCB is declared as a structure . The file name (FCB-FN) is initialized to 'EMP'. ECHO logging is requested (FCB-ECHO). The status bytes (FCB-STATUS) are binary and not initialized. The file is not on a sink machine, so all other bytes are initialized to blanks:

```
01  FCB.
    02  FCB-FN     PIC X(8) VALUE 'EMP     '.
    02  FCB-FT     PIC X(8) VALUE '   '.
    02  FCB-FM     PIC X(4) VALUE ' '.
    02  FILLER     PIC X(48) VALUE SPACES.
    02  FCB-ECHO   PIC X(4) VALUE 'ECHO'.
    02  FILLER     PIC X(20) VALUE SPACES.
    02  FCB-STATUS PIC S9(5) COMP.
    02  FILLER     PIC X(104) VALUE SPACES.
```

Initializing the FCB in a PL/I program

The following code (taken from Lines 2 through 13 of the PL/I program shown in Sample PL/1 Program) declares a File Communication Block. The FCB is declared as a structure . The file name (FN) is initialized to 'EMP'. The segment number, status, and error number bytes (SEGNUM, STATUS, and ERRORNUM) are binary and not initialized. No logging is requested, and the file is not on a sink machine, so those bytes are initialized to blanks:

```
DCL 1 FCB STATIC,
       2 FN         CHAR(8) INIT ('EMP'),
```

```
        2 FT          CHAR(8) INIT ('     '),
        2 FM          CHAR(4) INIT ('  '),
        2 FILL1       CHAR(48) INIT (' '),
        2 ECHO        CHAR(4) INIT (' '),1
        2 PASSCTL     CHAR(8),
        2 NEWSEG      CHAR(8),
        2 SEGNUM      FIXED BIN(31),
        2 STATUS      FIXED BIN(31),
        2 ERRORNUM    FIXED BIN(31),
        2 FILL2       CHAR(100);
```

Opening the FOCUS Data Source

Before you use any HLI functions, you must open the FOCUS data source using the OPN command. If your FOCUS file is protected by DBA security, you must have read/write access in order to use HLI. Cross-referenced files are opened with read-only access.

Open a FOCUS Data Source (OPN)

In C:

```
edahliCall(hliHandle, 2, opn, &fcb)
CALL FOCUS (opn, fcb, numb)
CALL 'FOCUS' USING opnfcbnumb.
CALL FOCUS (opn, fcb, numb);
```

In FORTRAN:

```
CALL FOCUS (opn, fcb, numb)
```

In COBOL:

```
CALL 'FOCUS' USING opnfcbnumb.
```

In PL/1:

```
CALL FOCUS (opn, fcb, numb);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI Interface. For instructions on creating the HLI handle, see Writing HLI Programs.

2

Is the number of parameters remaining in the call sequence, 2 for the OPN command.

opn

(Open) may be a variable containing the OPN command, or, in languages that allow it, may be the literal string 'OPN' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

numb

The database must have been created before executing the program, and the value of numb should be 0.

This must be the first call to FOCUS to open each FOCUS file, or if multiple positions to the same file are required, for each position.

When the file is opened, no data from the database has yet been retrieved. The Master File has been read and interpreted (consult the *Describing Data* manual for more information about Master Files for FOCUS databases).

Opening a FOCUS Data Source in a C Program

The following code (taken from Line 46 of the C program shown in Writing HLI Programs) opens the FOCUS database named EMP. The file name was initialized to 'EMP' in the filename field of the FCB:

```
rc = edahliCall(hli, 2, "OPN ", &fcb);
```

Opening a FOCUS Data Source in a FORTRAN Program

The following code (taken from Line 24 of the FORTRAN program shown in Writing HLI Programs) opens the FOCUS database named EMP . The file name was initialized to 'EMP' in the FN field of the FCB:

```
          CALL FOCUS (OPN,FCB)
```

Opening a FOCUS Data Source in a COBOL Program

The following code (taken from Line 107 of the COBOL program shown in Writing HLI Programs) opens the FOCUS database named EMP. The file name was initialized to 'EMP' in the FN field of the FCB:

```
         CALL 'FOCUS' USING OPNCMD FCB NUMB0.
```

Opening a FOCUS Data Source in a PL/I Program

The following code (taken from Line 64 of the PL/I program shown in Sample PL/1 Program) opens the FOCUS database named EMP. The file name was initialized to 'EMP' in the FN field of the FCB:

```
      CALL FOCUS (OPN, FCB);
```

Data Offsets in the Work Area

The work area is an area of memory for storing the FOCUS data source fields. The data offset is the number of bytes from the beginning of the work area to the place where the data for each of the fields will be located.

You can use the INFO command to obtain the offsets for each field (see HLI Command Summary, for a description of INFO). However, if you want to calculate them yourself, you must first determine the internal length of each field in the work area.

For example, in the Master File shown in Writing HLI Programs, EMPNO has a format of I5; NAME, A20; DATE, I6; and SALARY, D12.2. The lengths are determined as follows:

I5

Is an integer, and integers take four bytes of internal storage. The 5 expresses the display length.

A20

Is a character. Since one byte is allocated for each character, 20 bytes of storage are allocated.

I6

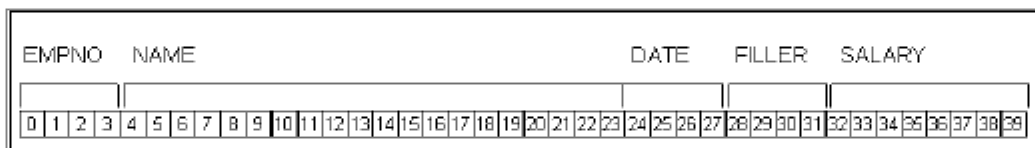
Is an integer, and integers take four bytes of internal storage. The 6 expresses the display length.

D12.2

Is a real number that uses eight bytes for storage. Again, the 12.2 is the display format.

Floating, integer, and alphanumeric fields always have to start on a word (4-byte) boundary; 8-byte real numbers must start on a double-word (8-byte) boundary.

The byte offset of a field is the byte on which the field begins. In our example, the byte offset of EMPNO is 0, since it starts on the first byte. EMPNO takes up four bytes; thus, the byte offset of NAME is 4. NAME takes up 20 bytes, which means that the next available byte on which to start a field is at offset 24. DATE is of type I, and must start on a word boundary. It takes up four bytes, so byte 27 is the next available byte for a field to begin on. SALARY, however, is of type D, which must begin on a double-word boundary. The next available double-word boundary is at offset 32; thus, SALARY must begin there. You must supply a filler field for the skipped bytes:



Note:

- In C, these filler fields can be supplied in a work area structure. The work area can be a structure or a character buffer.
- In FORTRAN, if you define your work area as an array, you do not need to supply any field locations to the compiler. Thus, the filler fields will automatically be included.
- In COBOL, these filler fields must be supplied in your FD statements or you can use JUSTIFY.
- In PL/1, you should not depend on PL/1 to adjust your fields for you. This is because the algorithm used may not be one that FOCUS uses.

Using a Show List

By default, all the fields in the database are available when the file is open. If your application program refers to a small number of fields, the SHO command selects the fields you want to use.

Select a List of Fields in a FOCUS Data Source (SHO)

In C:

```
edahliCall(hlihandle, 4, sho, &fcb, showlist, &numb);
CALL FOCUS (sho, fcb, showlist, numb)
CALL 'FOCUS' USING shofcbshowlistnumb.
CALL FOCUS (sho, fcb, showlist, numb);
```

In FORTRAN:

```
CALL FOCUS (sho, fcb, showlist, numb)
```

In COBOL:

```
CALL 'FOCUS' USING shofcbshowlistnumb.
```

In PL/1:

```
CALL FOCUS (sho, fcb, showlist, numb);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI interface. For instructions on creating the HLI handle, see Writing HLI Programs.

4

Is the number of parameters remaining in the call sequence, 4 for the SHO command.

sho

(Show) may be a variable with the value SHO or, in languages that allow it, may be the literal string ' SHO' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program. An OPN command must have been issued for this FCB.

showlist

Is the name of the array where the names of the desired fields are stored. Each entry in this array contains one 12-byte field name, padded with blanks if necessary.

numb

Is the number of fields in the show list.

If you do not issue a SHO command before the first retrieval command, FOCUS retrieves all fields in the path from the anchor segment to the target. However, because SHO can be used to organize your offsets to satisfy requirements such as beginning on a double word boundary. Also, if additional fields are added to the file that are not required for this program, using SHO isolates the program from such changes, so its use is recommended. Multiple SHOs for the same file/FCB may also be issued depending on the needs of the application.

INFO, a related command, returns Master File information such as segment names, field names, and formats. The field names are returned in the order of the show list. See HLI Command Summary, for information on the INFO command.

Using a Show List in a C Program

The following code (taken from Line 53 of the C program shown in Writing HLI Programs) issues the SHO command:

```
    rc = edahliCall(hli, 4, "SHO ", &fcb, names, &namesL);
```

The names list contains the names of the fields that are to be made available. The following code, taken from Lines 15 to 18 and 24 of the sample C program initialize this list with the field names EMPNO, NAME, DATE, and SALARY:

```
 #define FLD_EMPNO  "EMPNO       "
 #define FLD_NAME   "NAME        "
 #define FLD_DATE   "DATE        "
 #define FLD_SALARY "SALARY      "
    char *names =  FLD_EMPNO FLD_NAME FLD_DATE FLD_SALARY;
```

Using a Show List in a FORTRAN Program

The following code (taken from Line 29 of the FORTRAN program shown in Writing HLI Programs) issues the SHO command:

```
      20  CALL FOCUS (SHO,FCB,NAMES,4)
```

The NAMES array contains the names of the fields that are to be made available. The following code, taken from Lines 8 to 13 of the sample FORTRAN program initialize this array with the field names EMPNO, NAME, DATE, and SALARY:

```
          DIMENSION NAMES (3,4)
          DATA NAMES /9*'    '/
          DATA NAMES(1,1)/'EMPN'/,NAMES(2,1)/'O  '/,NAMES(3,1)/'  '/,
         *     NAMES(1,2)/'NAME'/,NAMES(2,2)/'   '/,NAMES(3,2)/'  '/,
```

```
      *     NAMES(1,3)/'DATE'/,NAMES(2,3)/'   '/,NAMES(3,3)/'  '/,
      *     NAMES(1,4)/'SALA'/,NAMES(2,4)/'RY '/,NAMES(3,4)/'  '/
```

Using a Show List in a COBOL Program

The following code (taken from Line 113 of the COBOL program shown in Writing HLI Programs) issues the SHO command:

```
         CALL 'FOCUS' USING SHOCMD FCB NAMES-AREA NUMB4.
```

The NAMES-AREA structure contains the names of the fields that are to be made available. The following code, taken from Lines 68 to 72 of the sample COBOL program initialize this array with the field names EMPNO, NAME, DATE, and SALARY:

```
     01  NAMES-AREA.
         02  FIELD-0001  PIC X(12) VALUE 'EMPNO       '.
         02  FIELD-0002  PIC X(12) VALUE 'NAME        '.
         02  FIELD-0003  PIC X(12) VALUE 'DATE        '.
         02  FIELD-0004  PIC X(12) VALUE 'SALARY      '.
```

Using a Show List in a PL/I Program

The following code (taken from Line 70 of the PL/I program shown in Sample PL/1 Program) issues the SHO command:

```
       CALL FOCUS (SHO, FCB, NAMES, NUMBER);
```

The NAMES-AREA structure contains the names of the fields that are to be made available. The following code, taken from Lines 15 to 19 of the sample PL/I program initialize this array with the field names EMPNO, NAME, DATE, and SALARY:

```
  DCL 1 NAMES STATIC,
        2 NAME1 CHAR(12) INIT('EMPNO'),
        2       NAME2   CHAR(12) INIT('NAME'),
        2 NAME3 CHAR(12) INIT('DATE'),
        2 NAME4 CHAR(12) INIT('SALARY');
```

Locating Records

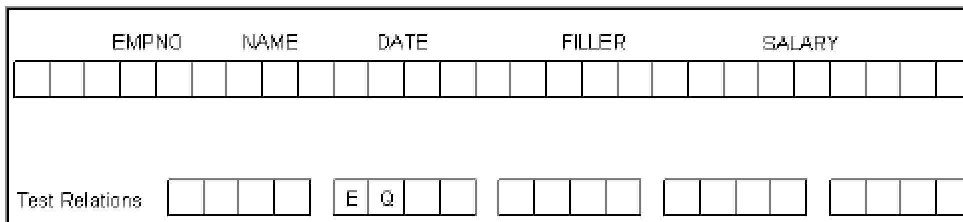
Now that the file is open, and a work area has been established, the program can search for a segment instance.

HLI offers three ways to read FOCUS files:

- In logical sequence through the pointers.
- In physical sequence as if it were a flat, sequential file.
- Through values in an index that you specify.

Expressing Test Relations

The work area is a series of bytes allocated to each field (see Opening the FOCUS Data Source, for information on byte offsets). There are also 4 bytes per field allocated for expressing test conditions, as shown below:



You can identify the field to be tested by marking the appropriate full-word in the test relations area. For example, suppose the test relation applies to the DATE field. You must mark the third full-word in the test relation array and identify the condition to apply to the field. Do this by entering eight blanks to indicate you want to skip the first eight bytes (which apply to the first two fields) in the array. Then place the relation in the third full-word (for the third field). Since this is only a two-byte relation (EQ), pad the second two bytes with blanks. Alternatively, establish the test relations as an array of 4-byte fields, and move the required relation value into the applicable element in the array.

Expressing Test Relations in a C Program

The following code (taken from Lines 75 to 77 in the program shown in Writing HLI Programs) creates the array for the test relations and the structure for the test literals:

```
        t_record testlt, wkarea;
        char testrl[4 * 4]; /* 4 bytes per SHO field */
        int testL = 1;      /* single test pass to FST */
```

Expressing Test Relations in a FORTRAN Program

The following code (taken from Lines 17 to 19 in the program shown in Writing HLI Programs) creates the arrays for the test relations and test literals:

```
        DIMENSION TESTRL(4),TESTLT(10),CHAREL(4),CHALIT(10)
        EQUIVALENCE (CHALIT(9),SALNEW)
        DATA TESTRL /4*'    '/, CHAREL /4*'    '/
```

The following code (taken from Lines 48 to 53) prompts the user to enter the test literal (the employee number for the record to be retrieved) and sets the test relation to 'EQ':

```
    80  WRITE (6,90)
    90  FORMAT (/,1X,'ENTER THE EMPLOYEE NUMBER OR 0 FOR MENU')
        READ (5,100,END=40) TESTLT(1)
   100  FORMAT (I5)
        IF (TESTLT(1).EQ.0) GO TO 40
        TESTRL(1)=EQ
```

Additional examples of test relations are on Lines 76 to 80 and 96 to 100.

Expressing Test Relations in a COBOL Program

The following code (taken from Lines 86 to 96 in the program shown in Writing HLI Programs) creates the structures for the test relations and test literals:

```
  01  TESTREL.
      02  REL-NO             PIC X(04) VALUE SPACES.
      02  REL-NAME           PIC X(04) VALUE SPACES.
      02  REL-DATE           PIC X(04) VALUE SPACES.
      02  REL-SAL            PIC X(04) VALUE SPACES.
  01  TESTLIT.
      02  LIT-EMPNO       PIC S9(9) COMP.
      02  LIT-NAME        PIC X(20) VALUE SPACES.
```

```
       02  LIT-DATE        PIC S9(6) COMP.
       02  FILLER          PIC X(4).
       02  LIT-SALARY      COMP-2.
```

The following code (taken from Lines 120 to 130) reads the test literal (the employee number for the record to be retrieved) and sets the test relation to 'EQ':

```
   B100-READ.
       READ TRANS AT END MOVE 9 TO EOF-FLAG GO TO B999-EXIT.
       IF TRANS-TYPE LESS THAN 1 OR TRANS-TYPE GREATER 4
          MOVE SPACES TO OUT-REC
          MOVE TRANS-REC TO OUT-REC
          MOVE 'ERROR IN TRANSACTION CODE' TO OUT-MSG
          WRITE OUTREC FROM OUT-REC
          GO TO B999-EXIT.
       MOVE SPACES TO OUT-REC MOVE TR-EMPNO TO OUT-EMPNO.
       MOVE SPACES TO TESTREL. MOVE 'EQ' TO REL-NO.
       MOVE TR-EMPNO TO LIT-EMPNO.
```

Additional examples of test relations are on Lines 169 and 202.

Expressing Test Relations in a PL/I Program

The following code (taken from Lines 26 to 36 in the program shown in Sample PL/1 Program) creates the structures for the test relations and test literals:

```
  DCL 1 TESTLT STATIC,
         2 EMPTEST   FIXED BIN(31),
         2 NAMETEST  CHAR(20),
         2 DATETEST  FIXED BIN(31),
         2 DUMMYT    FIXED BIN(31),
         2 SALTEST   FLOAT BIN(53);
  DCL 1 TESTRL STATIC,
         2 EMPREL    CHAR(4) INIT(' '),
         2 NAMEREL   CHAR(4) INIT(' '),
         2 DATEREL   CHAR(4) INIT(' '),
         2 SALREL    CHAR(4) INIT(' ');
```

The following code (taken from Lines 90 to 95) prompts the user to enter the test literal (the employee number for the record to be retrieved) and sets the test relation to 'EQ':

```
EMPLOYEE:
    PUT SKIP LIST ('ENTER THE EMPLOYEE NUMBER OR 0 FOR MENU');
    GET EDIT (EMPTEST) (F(5));
    IF EMPTEST = 0 THEN GOTO MENU;
    EMPREL=EQ;
    NUMBER=1;
```

Additional examples of test relations are on Lines 128 to 132 and 153 to 157.

Logical Reads

Logical reads use the pointers in the FOCUS file as the means of advancing from segment instance to segment instance.

The FST command is used to advance to the position of the first target segment instance under the anchor. If the anchor is SYSTEM, the FST command advances to the position of the first target segment instance in the file.

You use the FST command after you have issued several logical reads and want to return to the first segment instance or when you are trying to move to a parent segment from a child segment.

From within the file, issue NEX to get the next occurrence of a target segment.

Move to the First Target Segment Instance Under the Anchor (FST)

In C:

```
edahliCall(hliHandle, 10, fst, &fcb, &workarea, target, anchor, ntest,
testrelations, testliterals, null, nrepeat);
```

In FORTRAN:

```
CALL FOCUS (fst, fcb, workarea, target, anchor, ntest, testrelations,
testliterals, null, nrepeat)
```

In COBOL:

```
CALL 'FOCUS' USING fstfcbworkareatargetanchorntesttestrelations
testliteralsnullnrepeat.
```

In PL/1:

```
CALL FOCUS (fst, fcb, workarea, target, anchor, ntest, testrelations,
testliterals, null, nrepeat);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI interface. For instructions on creating the HLI handle, see Writing HLI Programs.

10

Is the number of parameters remaining in the call sequence, 10 for the FST command.

fst

(First) may be a variable that contains the FST command, or, in languages that allow it, may be the literal string, 'FST' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

workarea

Is a location where the data retrieved from the FOCUS file will be placed, if data is found. An OPN command must have been issued for this FCB.

target

Is the name of the segment to be retrieved. Notice that the segment name must take up the full eight characters allocated to it, even if it is fewer than eight characters long. In the sample programs, segment names have been declared as variables in order to avoid padding them with blanks.

anchor

Is the segment from which you start reading. In our example, SYSTEM is an imaginary segment that sits above all segments. Internally, SYSTEM is represented as segment #0.

ntest

Is the number of test conditions to be applied.

testrelations

Specifies the test condition to be applied. (See testliterals.)

testliterals

Is the value that you are searching for. Notice that in the sample programs four blanks are provided before the EQ to mark the byte position of the field to be tested. The offsets in the testliteral must match the offsets in the work area.

null

(Optional, only required if nrepeat is specified.) Is an 8-byte field used for positioning.

nrepeat

(Optional.) Specifies the number of records satisfying the test conditions that you want returned to you.

Move to the Next Segment in a Logical Read (NEX)

The NEX command specifies that you want to look for the next segment in a logical read. You can use the NEX command as the first logical read in your file.

In C:

```
edahliCall(hliHandle, 10, nex, &fcb, &workarea, target, anchor, ntest,
testrelations, testliterals, null, nrepeat);
```

In FORTRAN:

```
CALL FOCUS (nex, fcb, workarea, target, anchor, ntest, testrelations,
testliterals, null, nrepeat)
```

In COBOL:

```
CALL 'FOCUS' USING nexfcbworkareatargetanchorntesttestrelations
testliteralsnullnrepeat.
```

In PL/1:

```
CALL FOCUS (nex, fcb, workarea, target, anchor, ntest, testrelations,
testliterals, null, nrepeat);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI Interface. For instructions on creating the HLI handle, see Writing HLI Programs.

10

Is the number of parameters remaining in the call sequence, 10 for the NEX command.

nex

May be a variable that contains the NEX command, or, in languages that allow it, may be the literal string 'NEX ' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

workarea

Is a location where the data retrieved from the FOCUS file will be placed, if data is found. An OPN command must have been issued for this FCB.

target

Is the name of the segment to be retrieved. Notice that the segment name must take up the full eight characters allocated to it, even if it is fewer than eight characters long. In the sample programs, segment names have been declared as variables in order to avoid padding them with blanks.

anchor

Is the segment from which you start reading. In our example, SYSTEM is an imaginary segment that sits above all segments. Internally, SYSTEM is represented as segment #0.

ntest

Is the number of test conditions to be applied.

testrelations

Specifies the test condition to be applied. (See testliterals.)

testliterals

Is the value that you are searching for. Notice that in the sample programs four blanks are provided before the EQ to mark the byte position of the field to be tested. The offsets in the testliteral must match the offsets in the work area.

null

(Optional, only required if nrepeat is specified.) Is an 8-byte field used for positioning.

nrepeat

(Optional.) Specifies the number of records satisfying the test conditions that you want returned to you. If not specified, one instance will be returned.

If literals have been coded as constants in a call, you must pad each alphanumeric field with blanks to fix the byte offsets. Numeric constants cannot usually be hard coded in a call. If you declare your literals as variables instead, the variables must be defined with the correct byte offset.

Using Logical Reads in a C Program

The following code (taken from Lines 83 to 84 in the program shown in Writing HLI Programs) finds the first segment instance in segment ONE starting from SYSTEM that matches the employee number read in as the test literal in Locating Records:

```
          rc = edahliCall(hli, 8, "FST ", &fcb,
               &wkarea, SEG_ONE, SEG_SYS, &testL, testrl, &testlt);
```

Additional examples of using logical reads are on Lines 141, 171, and 191.

Using Logical Reads in a FORTRAN Program

The following code (taken from Line 54 in the program shown in Writing HLI Programs) finds the first segment instance in segment ONE starting from SYSTEM that matches the employee number read in as the test literal in Locating Records:

```
        CALL FOCUS (FST, FCB, WKAREA, ONE,SYSTEM , 1, TESTRL, TESTLT)
```

Additional examples of using logical reads are on Lines 81, 101, and 114.

Using Logical Reads in a COBOL Program

The following code (taken from Lines 131 and 132 in the program shown in Writing HLI Programs) finds the first segment instance in segment ONE starting from SYSTEM that matches the employee number read in as the test literal in Locating Records:

```
       CALL 'FOCUS' USING FSTCMD FCB WRKAREA ONESEG SYSSEG
          NUMB1 TESTREL TESTLIT.
```

Additional examples of using logical reads are on Lines 188 and 203.

Using Logical Reads in a PL/I Program

The following code (taken from Line 96 in the program shown in Sample PL/1 Program) finds the first segment instance in segment ONE starting from SYSTEM that matches the employee number read in as the test literal in Expressing Test Relations in a PL/I Program:

```
     CALL FOCUS ( FST,FCB,WKAREA,ONE,SYSTEM,NUMBER,TESTRL,TESTLT );
```

Additional examples of using logical reads are on LInes 133, 158, and 175.

Physical Reads

You can also use HLI to read segment instances in their stored sequence (when included in the file, regardless of pointer) using the FSP command. The syntax for a physical read is

very similar to that for a logical read, except you do not provide an anchor segment. It goes to the first physical segment instance in the file that matches the test conditions (if any), without using the logical pointers stored in the file.

Move to the First Physical Segment Instance (FSP)

In C:

```
edahliCall(hliHandle, 7, fsp, &fcb, &workarea, target, ntest,
testrelations, testliterals);
```

In FORTRAN:

```
CALL FOCUS (fsp, fcb, workarea, target, ntest, testrelations,
testliterals)
```

In COBOL:

```
CALL 'FOCUS' USING fspfcbworkareatargetntesttestrelationstestliterals.
```

In PL/1:

```
CALL FOCUS (fsp, fcb, workarea, target, ntest, testrelations,
testliterals);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI Interface. For instructions on creating the HLI handle, see Writing HLI Programs.

7

Is the number of parameters remaining in the call sequence, 7 for the FSP command.

fsp

(First Physical) may be a variable that contains the FSP command, or, in languages that allow it, may be the literal string 'FSP' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

workarea

Is a location where the data retrieved from the FOCUS file will be placed, if data is found. An OPN command must have been issued for this FCB.

target

Is the name of the segment to be retrieved. Notice that the segment name must take up the full eight characters allocated to it, even if it is fewer than eight characters long. In the sample programs, segment names have been declared as variables in order to avoid padding them with blanks.

ntest

Is the number of test conditions to be applied.

testrelations

Specifies the test condition to be applied. (See testliterals.)

testliterals

Is the value that you are searching for. Notice that in the sample programs four blanks are provided before the EQ to mark the byte position of the field to be tested. The offsets in the testliteral must match the offsets in the work area.

Retrieve the Next Physical Occurrence of the Target Segment (NXP)

The NXP command is used to retrieve the next physically adjacent occurrence of the target segment.

In C:

```
edahliCall(hliHandle, 7, nxp, &fcb, &workarea, target, ntest,
testrelations, testliterals);
```

In FORTRAN:

```
CALL FOCUS (nxp, fcb, workarea, target, ntest, testrelations,
testliterals)
```

In COBOL:

```
CALL 'FOCUS' USING nxpfcbworkareatargetntesttestrelationstestliterals.
```

In PL/1:

```
CALL FOCUS (nxp, fcb, workarea, target, ntest, testrelations,
testliterals);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI Interface. For instructions on creating the HLI handle, see Writing HLI Programs.

7

Is the number of parameters remaining in the call sequence, 7 for the NXP command.

nxp

(Next Physical) may be a variable that contains the NXP command, or, in languages that allow it, may be the literal string 'NXP' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

workarea

Is a location where the data retrieved from the FOCUS file will be placed, if data is found. An OPN command must have been issued for this FCB.

target

Is the name of the segment to be retrieved. Notice that the segment name must take up the full eight characters allocated to it, even if it is less than eight characters long. In the sample programs, segment names have been declared as variables in order to avoid padding them with blanks.

anchor

Is the segment from which you start reading. In our example, SYSTEM is an imaginary segment that sits above all segments. Internally, SYSTEM is represented as segment #0.

ntest

Is the number of test conditions to be applied.

testrelations

Specifies the test condition to be applied. (See testliterals.)

testliterals

Is the value that you are searching for. Notice that in the sample programs four blanks are provided before the EQ to mark the byte position of the field to be tested. The offsets in the testliteral must match the offsets in the work area.

Note: Physical searches of the database do not retrieve logically related segments. Thus, you will receive an error status code if you attempt to delete an instance that has been retrieved physically, as all instances of a given parent must be logically chained together. After retrieving an instance physically, the program may use logical retrieval commands to retrieve the parent instance, and then re-retrieve the instance to be deleted.

Indexed Reads

HLI supports retrieval through a FOCUS file index for a specific key value. The NXD command locates the first instance containing a key value of a specified indexed field.

Retrieve a Segment Instance Using an Index (NXD)

In C:

```
edahliCall(hliHandle, 9, nxd, &fcb, &workarea, target, field, ntest,
testrelations,
```

In FORTRAN:

```
CALL FOCUS (nxd, fcb, workarea, target, field, ntest,
testrelations,testliterals, savearea)
```

In COBOL:

```
CALL 'FOCUS' USING nxdfcbworkareatargetfieldntesttestrelations
testliteralssavearea.
```

In PL/1:

```
CALL FOCUS (nxd, fcb, workarea, target, field, ntest,
testrelations,testliterals, savearea);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI Interface. For instructions on creating the HLI handle, see Writing HLI Programs.

9

Is the number of parameters remaining in the call sequence, 9 for the NXD command.

nxd

(Next through Index) may be a variable containing the NXD command, or, in languages that allow it, may be the literal string 'NXD' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

workarea

Is a location where the data retrieved from the FOCUS file will be placed, if the data is found.

target

Is the segment to be retrieved using the index of the specified field.

field

Names the field whose index will be used to retrieve the target segment. The field must be declared in the Master File Description with the FIELDTYPE=I.

ntest

Is the number of test conditions to be applied.

testrelations

Specifies the test conditions to be applied.

testliterals

Is the value you are searching for.

savearea

Is a 32-byte area used to save information when the retrieval is requested. This should be set to binary zeros for the first occurrence of a specific value, but left unchanged for subsequent retrieval of instances with the same value for the indexed field.

Retrieving With a Backkey

The NXK command works through a direct address in the FCB (bytes 61-68), and retrieves a previous target segment you have earmarked for this purpose. Following the retrieval, the backkey in the FCB contains the address. This may be saved for future use.

Retrieve a Previous Target Segment Using a Backkey (NXK)

In C:

```
edahliCall(hliHandle, 5, nxk, &fcb, &workarea, target,
```

In FORTRAN:

```
CALL FOCUS (nxk, fcb, workarea, target, backkey)
```

In COBOL:

```
CALL 'FOCUS' USING nxkfcbworkareatargetbackkey.
```

In PL/1:

```
CALL FOCUS (nxk, fcb, workarea, target, backkey);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI interface. For instructions on creating the HLI handle, see Writing HLI Programs.

5

Is the number of parameters remaining in the call sequence, 5 for the NXK command.

nxk

Next through the Key may be a variable containing the NXK command, or, in languages that allow it, may be the literal string 'NXK' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

workarea

Is a location where the data retrieved from the FOCUS file, if any is found, will be placed.

target

Is the name of the segment you want to retrieve.

backkey

Is an address into the file that lets you move back to a previously retrieved segment. The current segment address begins in byte 61 of the FCB. When multiple instances were retrieved into an array with a single logical retrieval, the address of each instance retrieved is in the 8-byte field designated in the work area for multiple retrievals. For additional information, see Defining the Record Work Area

Altering the File

Once the segments are retrieved, you can:

- Include new segment instances.
- Change the information in segment instances.
- Delete segment instances.

Note: You must have a logical position in the file to add or delete segments.

All changes to the database are written to a buffer. They are not permanently saved in the database until the program either issues a SAV command or issues a CLO command.

When maintaining a file under the control of a sink machine, before accepting a maintenance command, verification is made by the sink process that the instance has not changed by another user since it was retrieved. If it was, the command is not processed, and a non-zero return code is returned.

Including New Segments

You use the INP command to include new segment instances. The INP command creates new segment instances that include the information contained in the *inpliteral* variable.

Include New Segment Instances (INP)

In C:

```
edahliCall(hliHandle, 5, inp, &fcb,  inpliteral, target, n);
```

In FORTRAN:

```
CALL FOCUS (inp, fcb, inpliteral, target, n)
```

In COBOL:

```
CALL 'FOCUS' USING inpfcbinpliteraltargetn.
```

In PL/1:

```
CALL FOCUS (inp, fcb, inpliteral, target, n);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI Interface. For instructions on creating the HLI handle, see Writing HLI Programs.

5

Is the number of parameters remaining in the call sequence, 5 for the INP command.

inp

(Input) may be a variable that contains the INP command, or, in languages that allow it, may be the literal string 'INP' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

inpliteral

Is an array that contains the new information you want to enter. The data to be included must be assembled in a data structure that can be passed to HLI. This data structure is contained in the inpliteral variable. The format of the inpliteral structure is the same as the format of the work area.

target

Is the name of the segment to be added. It must be padded to eight characters. The target is located using your current position in the file, established through logical reads.

n

Is a numeric code to control how the new segment is added. If the segment type is S0 or blank, the possible values of *n* are:

0

Inserts the new segment after the current segment, providing the key fields are the same.

1

Inserts the new segment before the current segment, providing the key fields are the same.

If the segment type is S*n* or SH*n*, the value of *n* must be:

0

Duplicate keys are allowed. This is not supported when the file is under the control of a sink machine.

2

Rejects the new segment instance if the key exists.

Including New Segment Instances Using a C Program

The following code (taken from Lines 120 and 121 in the program shown in Writing HLI Programs) inputs a new segment TWO instance after reading the new date and salary values into the work area. Because the SEGTYPE of segment TWO is SHI, the value 2 for the last argument specifies that the transaction is rejected if the key (date value) exists in the data source:

```
      rc = edahliCall(hli, 3, "INP ", pFcb, wkarea, "TWO     ",
                           &c__2);
```

Including New Segment Instances Using a FORTRAN Program

The following code (taken from Line 71 in the program shown in Writing HLI Programs) inputs a new segment TWO instance after reading the new date and salary values into the work area. Because the SEGTYPE of segment TWO is SHI, the value 2 for the last argument specifies that the transaction is rejected if the key (date value) exists in the data source:

```
         CALL FOCUS (INP,FCB,WKAREA,TWO,2)
```

Including New Segment Instances Using a COBOL Program

The following code (taken from Line 160 in the program shown in Writing HLI Programs) inputs a new segment TWO instance after reading the new date and salary values and placing them in the work area. Because the SEGTYPE of segment TWO is SHI, the value 2 for the last argument specifies that the transaction is rejected if the key (date value) exists in the data source:

```
        CALL 'FOCUS' USING INPCMD FCB WRKAREA TWOSEG NUMB2.
```

Including New Segment Instances Using a PL/I Program

The following code (taken from Line 119 in the program shown in Sample PL/1 Program) inputs a new segment TWO instance after reading the new date and salary values into the work area. Because the SEGTYPE of segment TWO is SHI, the value 2 for the last argument specifies that the transaction is rejected if the key (date value) exists in the data source:

```
      CALL FOCUS (INP, FCB, WKAREA, TWO, NUMBER);
```

Changing Information in the File

Use the CHA command to change information in a segment. All the new information must be loaded into an array before the CHA command is issued. Then the CHA command specifies the byte offset of the field that you want to change. The *changelist* variable indicates what fields are to be changed.

Change Information in a Segment Instance (CHA)

In C:

```
edahliCall(hliHandle, 7, cha, &fcb, &workarea, target, anchor, n,
changelist, changeliterals);
```

In FORTRAN:

```
CALL FOCUS (cha, fcb, workarea, target, anchor, n, changelist,
changeliterals)
```

In COBOL:

```
CALL 'FOCUS' USING chafcbworkareatargetanchornchangelistchangeliterals.
```

In PL/1:

```
CALL FOCUS (cha, fcb, workarea, target, anchor, n, changelist,
changeliterals);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI interface. For instructions on creating the HLI handle, see Writing HLI Programs.

8

Is the number of parameters remaining in the call sequence, 8 for the CHA command.

cha

(Change) may be a variable that contains the CHA command or, in languages that allow it, may be the literal string 'CHA' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

workarea

Is the previously defined work area.

target

Is the target segment on which you want the changes performed. The target instance changed is the one at your current position in the file, established through logical reads.

anchor

Is the anchor segment. Although this parameter is required in the syntax, it is not used and can be a null value (eight-byte field of blank spaces).

n

Is the number of fields that you want to change.

changelist

Indicates which fields are to be changed. Its format is identical to that of *testrelations*. The data fields whose corresponding *changelist* values are set to EQ are changed to the

values specified by the *changeliterals* value. A copy of the changed segment instance is returned to the work area. The *changeliterals* area must correspond to the current SHO command field layout. there are four bytes in the changelist area for each field in the showlist. Only those fields that are in the target segment are changed.

changeliterals

Is the set of new values for the fields (declared by their byte offsets in the same format that the work area is declared) that you want to change.

Changing a Segment Instance in a C Program

The following code (taken from Lines 129 and 132 to 133 in the program shown in Writing HLI Programs) establishes the *charel* array and *chalit* structure, which will contain the field values to change (based on byte offsets in the work area):

```
t_record testlt, wkarea, chalit;
char charel[4 * 4]; /* 4 bytes per SHO field */
int chaL = 1; /* changing just one field */
```

The following code (taken from Lines 141 and 142) establishes the current position of segment TWO using the logical read command NEX:

```
   rc = edahliCall(hli, 8, "NEX ", pFcb, &wkarea,
                   SEG_TWO, SEG_ONE, &testL, testrl, &testlt);
```

The following code (taken from Lines 150 to 154) reads the new salary into *chalit*, places the value EQ in the fourth word of a *charel* array (which indicates that the one field to be replaced is the SALARY field in the current instance of segment TWO), and issues the CHA command:

```
scanf("%9.2lf", &chalit.salary);
memset(charel, ' ', sizeof(charel));
memcpy(&charel[16], "EQ  ", 4);
rc = edahliCall(hli, 7, "CHA ", pFcb,
                &wkarea, SEG_TWO, SEG_ONE, charel, &chalit);
```

Changing a Segment Instance in a FORTRAN Program

The following code (taken from Lines 17 to 19 in the program shown in Writing HLI Programs) establishes the CHAREL and CHALIT arrays, which will contain the field values to change (based on byte offsets in the work area). CHALIT(9), where a new salary value will be placed, corresponds to WKAREA(9):

```
      DIMENSION TESTRL(4),TESTLT(10),CHAREL(4),CHALIT(10)
      EQUIVALENCE (CHALIT(9),SALNEW)
      DATA TESTRL /4*'    '/, CHAREL /4*'    '/
```

The following code (taken from Line 81) establishes the current position of segment TWO using the logical read command NEX:

```
      CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,1,TESTRL,TESTLT)
```

The following code (taken from Lines 89 to 91) reads the new salary into the CHALIT array, places the value EQ in the fourth word of CHAREL array (which indicates that the one field to be replaced is the SALARY field in the current instance of segment TWO), and issues the CHA command:

```
      READ (5,1400) SALNEW
      CHAREL(4)=EQ
      CALL FOCUS (CHA,FCB,WKAREA,TWO,ONE,1,CHAREL,CHALIT)
```

Changing a Segment Instance in a COBOL Program

The following code (taken from Lines 86 to 96 in the program shown in Writing HLI Programs) establishes the TESTREL and TESTLIT structures, which will contain the field values to change (based on byte offsets in the work area). LIT-SALARY, where a new salary value will be placed, corresponds to SALARY in WRKAREA:

```
  01  TESTREL.
      02  REL-NO              PIC X(04) VALUE SPACES.
      02  REL-NAME            PIC X(04) VALUE SPACES.
      02  REL-DATE            PIC X(04) VALUE SPACES.
```

```
        02  REL-SAL            PIC X(04) VALUE SPACES.
    01  TESTLIT.
        02  LIT-EMPNO       PIC S9(9) COMP.
        02  LIT-NAME        PIC X(20) VALUE SPACES.
        02  LIT-DATE        PIC S9(6) COMP.
        02  FILLER          PIC X(4).
        02  LIT-SALARY      COMP-2.
```

The following code (taken from Lines 199 to 204) establishes the current position of segment TWO using the logical read command FST:

```
       MOVE SPACES TO TESTLIT MOVE TR-EMP-DATE TO LIT-DATE.
       MOVE TR-EMP-DATE TO OUT-DATE.
       MOVE TR-SALARY TO OUT-SALARY.
       MOVE SPACES TO TESTREL MOVE 'EQ' TO REL-DATE.
       CALL 'FOCUS' USING FSTCMD FCB WRKAREA TWOSEG ONESEG
          NUMB1 TESTREL TESTLIT.
```

The following code (taken from Lines 169 to 172) places the new salary into LIT-SALARY, places the value EQ in REL-DATE (which indicates that the one field to be replaced is the SALARY field in the current instance of segment TWO), and issues the CHA command:

```
        MOVE TR-SALARY TO LIT-SALARY. MOVE 'EQ' TO REL-SAL.
        MOVE SPACES TO REL-DATE.
        CALL 'FOCUS' USING CHACMD FCB WRKAREA TWOSEG NULL NUMB1
           TESTREL TESTLIT.
```

Changing a Segment Instance in a PL/I Program

The following code (taken from Lines 37 to 47 in the program shown in Sample PL/1 Program) establishes the CHAREL and CHALIT structures, which will contain the field values to change (based on byte offsets in the work area). CSALTEST, where a new salary value will be placed, corresponds to SALARY in WKAREA:

```
    DCL 1 CHALIT STATIC,
           2 CEMPTEST  FIXED BIN(31),
           2 CNAMETEST CHAR(20),
           2 CDATETEST FIXED BIN(31),
           2 CDUMMYT   FIXED BIN(31),
           2 CSALTEST  FLOAT BIN(53);
```

```
DCL 1 CHAREL STATIC,
       2 CEMPREL   CHAR(4) INIT(' '),
       2 CNAMEREL  CHAR(4) INIT(' '),
       2 CDATEREL  CHAR(4) INIT(' '),
       2 CSALREL   CHAR(4) INIT(' ');
```

The following code (taken from Line 133) establishes the current position of segment TWO using the logical read command NEX:

```
CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,NUMBER,TESTRL,TESTLT);
```

The following code (taken from Lines 138 to 140) reads the new salary into CSALTEST, places the value EQ in CSALREL (which indicates that the one field to be replaced is the SALARY field in the current instance of segment TWO), and issues the CHA command:

```
GET EDIT (CSALTEST) (F(9,2));
CSALREL=EQ;
CALL FOCUS (CHA,FCB,WKAREA,TWO,ONE,NUMBER,CHAREL,CHALIT);
```

Deleting Segments From a File

The DEL command deletes an instance of a target segment and all its descendants, regardless of what is activated in the show list.

Delete Segments From a File (DEL)

In C:

```
edahliCall(hliHandle, 3, del, &fcb, target);
```

In FORTRAN:

```
CALL FOCUS (del, fcb, target)
```

In COBOL:

```
CALL 'FOCUS' USING delfcbtarget.
```

In PL/1:

```
CALL FOCUS (del, fcb, target);
```

where:

hliHandle

Is a pointer to a memory control area for the HLI Interface. For instructions on creating the HLI handle, see Writing HLI Programs.

3

Is the number of parameters remaining in the call sequence, 3 for the DEL command.

del

(Delete) may be a variable containing the DEL command, or, in languages that allow it, may be the literal string 'DEL' (padded with one trailing blank).

fcb

Is the File Communication Block initialized in the beginning of the program.

target

Is the name of the segment you want to delete. The target instance deleted is the one at your current position in the file, established through logical reads. All of the target's descendants are also deleted.

Deleting a Segment Instance in a C Program

The following code (taken from Lines 167 to 172 in the program shown in Writing HLI Programs) reads a date into the test literal structure at the position for the DATE field, sets the test relation to EQ, and uses the NEX command to establish a position at the instance of segment TWO that matches the test date:

```
        scanf("%d", &teslit.date);
        if(teslit.date == 0) return;
        memset(testrl, ' ', sizeof(testrl));
```

```
    memcpy(&testrl[8], "EQ   ", 4);
    rc = edahliCall(hli, 8, "NEX ", pFcb, &wkarea,
                    SEG_TWO, SEG_ONE, &testL, testrl, &teslit);
```

The following code (taken from Line 178) deletes that segment instance:

```
rc = edahliCall(hli, 3, "DEL", pFcb, SEG_TWO);
```

Deleting a Segment Instance in a FORTRAN Program

The following code (taken from Lines 97 to 101 in the program shown in Writing HLI Programs) reads a date into the test literal array at the position for the DATE field, sets the test relation to EQ, and uses the NEX command to establish a position at the instance of segment TWO that matches the test date:

```
      READ (5,1200) TESTLT(7)
      IF (TESTLT(7).EQ.0) GO TO 40
      TESTRL(1)=BLANK
      TESTRL(3)=EQ
      CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,1,TESTRL,TESTLT)
```

The following code (taken from Line 106) deletes that segment instance:

```
3100 CALL FOCUS (DEL,FCB,TWO)
```

Deleting a Segment Instance in a COBOL Program

The following code (taken from Lines 199 to 204 in the program shown in Writing HLI Programs) moves the transaction date into the LIT-DATE field of the test literal structure, sets the test relation to EQ, and uses the FST command to establish a position at the instance of segment TWO that matches the test date:

```
     MOVE SPACES TO TESTLIT MOVE TR-EMP-DATE TO LIT-DATE.
     MOVE TR-EMP-DATE TO OUT-DATE.
     MOVE TR-SALARY TO OUT-SALARY.
     MOVE SPACES TO TESTREL MOVE 'EQ' TO REL-DATE.
```

```
        CALL 'FOCUS' USING FSTCMD FCB WRKAREA TWOSEG ONESEG
           NUMB1 TESTREL TESTLIT.
```

The following code (taken from Line 181) deletes that segment instance:

```
        CALL 'FOCUS' USING DELCMD FCB TWOSEG.
```

Deleting a Segment Instance in a PL/I Program

See the appropriate sample (FORTRAN: Line 106; COBOL: Line 181; PL/1: Line 161).

The following code (taken from Lines 154 to 158 in the program shown in Sample PL/1 Program) reads a date into the DATETEST field of the test literal structure, sets the test relation to EQ, and uses the NEX command to establish a position at the instance of segment TWO that matches the test date:

```
      GET EDIT (DATETEST) (F(6));
      IF DATETEST = 0 THEN RETURN;
      EMPREL=BLANK;
      DATEREL=EQ;
      CALL FOCUS (NEXT,FCB,WKAREA,TWO,ONE,NUMBER,TESTRL,TESTLT);
```

The following code (taken from Line 161) deletes that segment instance:

```
         CALL FOCUS (DEL,FCB,TWO);
```

Testing Status, Using Log Facilities, and Handling Errors

This chapter describes facilities for testing the success of HLI calls, logging program activity, and handling errors.

Testing Status

Although HLI does check the status of your file after every command, it does not issue error messages. Thus, you should build a status check after every HLI call in your program.

HLI Status Code	Meaning
`0`	Execution ended normally.
`1`	Reached end of file or end of chain

Any other code indicates an error condition (see HLI Status Return Codes).

Using the Diagnostic Log Facility: ECHO and STAT

You can use the ECHO and STAT facilities to produce diagnostic logs of all calls to HLI. This is useful in the development of new programs and tracking user access.

When the eighteenth word of the FCB (bytes 69-72) contains the string ECHO or STAT, each HLI call is logged. The HLI function calls create the line displayed in the log file immediately before returning from the call. Its presence does not affect the operation of user programs or their logic.

Log File Locations

For Mainframe FOCUS, the trace line is written to ddname HLIPRINT. You must issue a FILEDEF or ALLOCATE command for this ddname before running your program. When ECHO is specified, the LRECL of the output file is 88. STAT requires an LRECL of 133.

Under TSO, the recommended that ALLOCATE commands are:

- When displaying on the terminal:

```
ALLOC F(HLIPRINT) DA(*)
```

- When storing in a disk file:

```
ATTR HLIDCB LRECL(88) RECFM(F B) BLKSIZE(880)
```

```
ALLOC F(HLIPRINT) DA(dataset) USING (HLIDCB) SPACE (5,5)
TRACKS CATALOG
```

In the z/OS batch, the recommended JCL statements are:

- When printing:

```
//HLIPRINT DD SYSOUT=*
```

- When storing in a disk file:

```
//HLIPRINT DD DSN=dataset,DCB=(LRECL=88,RECFM=FB,BLKSIZE=880),
//     UNIT=unit,VOL=SER=volume,SPACE=space,
//     DISP=(NEW,CATLG)
```

For the WebFOCUS Reporting Server and WebFOCUS, if FDS is active, the server configuration controls the file location, and the file is written by the server. Without FDS, the HLIPRDIR variable specifies the HLIPRINT output directory, and the file name is always hliprint.log.

For the WebFOCUS Reporting Server and WebFOCUS, if FDS is active, the server configuration controls the file location, and the file is written by the server. Without FDS, the HLIPRDIR variable specifies the HLIPRINT output directory, and the file name is always hliprint.log.

Using the ECHO Log Facility

Your application program can turn the log facility on or off at any time during its execution. The following is a sample ECHO log, produced when a FCB word 18 contains the string ECHO:

```
CMD  FILENAME        STATUS NEWSEG  TARGET   ANCHOR   NTEST  USERID
REFNUMB
OPN  CAR       FOCUS   00
00000001
OPN  EMPLOYEE  FOCUS   00
00000002
FST  CAR       FOCUS   00   ORIGIN  CARREC   SYSTEM   00
00000003
FST  EMPLOYEE  FOCUS   00   EMPINFO ADDRESS  SYSTEM   00
00000004
NEX  CAR       FOCUS   00   CARREC  CARREC   SYSTEM   01
00000005
NEX  EMPLOYEE  FOCUS   00   EMPINFO ADDRESS  SYSTEM   01
00000006
CLO  EMPLOYEE  FOCUS   00
00000007
CLO  CAR       FOCUS   00
00000008
```

The columns in the log file, have the following meanings:

Column Title	Meaning
`CMD`	Command that was issued.
`FILENAME`	File name in the FCB.
`STATUS`	Status return code, FCB word 24.
`NEWSEG`	Name of the first segment of new information, FCB words 21 and 22.

Column Title	Meaning
`TARGET`	Name of the target segment.
`ANCHOR`	Name of the anchor segment.
`NTEST`	Value of the *numbtests* argument.
`USERID`	Blank for local HLI programs.
`REFNUMB`	A sequentially assigned number equal to the transaction number in the ? FILE file name command.

Using the STAT Log Facility

If a word 18 contains the string STAT, the HLIPRINT file is 133 bytes wide and contains additional fields showing access times and I/O information, which may be more useful in some situations.

The following is an example of the HLIPRINT file using the STAT option:

```
1CMD  FILENAME        STATUS NEWSEG  TARGET   ANCHOR  NTEST USERID   REF NUMB  DATE   TIME   UTIME TTIME IOS  PROC NAME CASE NAME
0
 RD   employee(focus)    0                               1  HOFFMAN 00000001 950305 162254 .0000 .0000  1
 RD   jobfile(focus)     0                               1  HOFFMAN 00000002 950305 162254 .0000 .0000  1
 RD   educfile(focus)    0                               1  HOFFMAN 00000003 950305 162254 .0000 .0000  1
 OPN  employee(focus)    0                                  HOFFMAN 00000004 950305 162255 .0000 .0000  1
 MINT employee(focus)    0                                  HOFFMAN 00000005 950305 162255 .0000 .0000  0
 MATC employee(focus)    0           EMPINFO                HOFFMAN 00000006 950305 162319 .0000 .0000  0
 MATD employee(focus)    0           SALINFO                HOFFMAN 00000007 950305 162319 .0000 .0000  1
 MUPD employee(focus)    0           SALINFO                HOFFMAN 00000008 950305 162326 .0000 .0000  0
 CLO  employee(focus)    0                                  HOFFMAN 00000009 950305 162330 .0000 .0000  1
 RD   employee(focus)    0                               1  HOFFMAN 00000010 950305 162347 .0000 .0000  1
 RD   jobfile(focus)     0                               1  HOFFMAN 00000011 950305 162347 .0000 .0000  1
 RD   educfile(focus)    0                               1  HOFFMAN 00000012 950305 162347 .0000 .0000  1
 RD   employee(focus)    0                               1  HOFFMAN 00000013 950305 162348 .0000 .0000  1
 RD   employee(focus)    0                               2  HOFFMAN 00000014 950305 162348 .0000 .0000  1
 RD   employee(focus)    0                               3  HOFFMAN 00000015 950305 162348 .0000 .0000  1
 RD   jobfile(focus)     0                               4  HOFFMAN 00000016 950305 162348 .0000 .0000  1
 RD   jobfile(focus)     0                               1  HOFFMAN 00000017 950305 162348 .0000 .0000  1
 RD   jobfile(focus)     0                               2  HOFFMAN 00000018 950305 162348 .0000 .0000  1
 RD   jobfile(focus)     0                               3  HOFFMAN 00000019 950305 162348 .0000 .0000  1
 RD   employee(focus)    0                               4  HOFFMAN 00000020 950305 162348 .0000 .0000  1
 RD   employee(focus)    0                               5  HOFFMAN 00000021 950305 162348 .0000 .0000  1
 RD   employee(focus)    0                               6  HOFFMAN 00000022 950305 162348 .0000 .0000  1
 RD   educfile(focus)    0                               3  HOFFMAN 00000023 950305 162348 .0000 .0000  1
 RD   educfile(focus)    0                               2  HOFFMAN 00000024 950305 162348 .0000 .0000  1
 RD   educfile(focus)    0                               1  HOFFMAN 00000025 950305 162348 .0000 .0000  1
 RD   employee(focus)    0                               7  HOFFMAN 00000026 950305 162349 .0000 .0000  1
 RD   employee(focus)    0                               7  HOFFMAN 00000027 950305 162350 .0000 .0000  1
 RD   employee(focus)    0                               8  HOFFMAN 00000028 950305 162350 .0000 .0000  1
 RD   employee(focus)    0                               5  HOFFMAN 00000029 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               3  HOFFMAN 00000030 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               2  HOFFMAN 00000031 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               1  HOFFMAN 00000032 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               2  HOFFMAN 00000033 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               3  HOFFMAN 00000034 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               2  HOFFMAN 00000035 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               1  HOFFMAN 00000036 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               2  HOFFMAN 00000037 950305 162350 .0000 .0000  1
 RD   educfile(focus)    0                               3  HOFFMAN 00000038 950305 162350 .0000 .0000  1
```

The columns have the following meanings:

Column Title	Meaning
`CMD`	The command that was issued.
`FILENAME`	The file names present in the FCB.
`STATUS`	The status return code, FCB word 24.
`NEWSEG`	The name of the first segment of new information, FCB words 21 and 22.
`TARGET`	The name of the target segment.
`ANCHOR`	The name of the anchor segment.

Column Title	Meaning
`NTEST`	The value of the numbtests argument.
`USERID`	Blank for local HLI programs.
`REFNUMB`	A sequentially assigned number equal to the transaction number in the ? FILE file name command.
`DATE`	The date on which the command was executed, in YYMMDD format.
`TIME`	The time at which the command finished executing, in HHMMSS format.
`VTIME`	**Under z/OS:** The amount of elapsed job-step time required for the command as indicated in the ASCBEJST field of the ASCB for the sink machine.
`TTIME`	**Under z/OS:** The value of this field is the same as that of VTIME.
`IOS`	The number of FOCUS database I/Os required to execute the command.
`PROC NAME`	The contents of FCB words 7 and 8.
`CASE NAME`	Blank for HLI transactions. For MODIFY/SU, case name.

A Master File named HLIPRINT is provided on the distribution tape. You can use this Master File to report on the STAT output file.

Error Handling

HLI returns an error status code in the FCB at word 24. Any code other than 0 or 1 indicates an error. HLI does not, however, issue an error message; error handling is left to the programmer (see HLI Status Return Codes).

An error message file is supplied with the Host Language Interface. The messages in the file correspond to the status return code error numbers. You might use the error message file to display an error message at the terminal when an error occurs.

The message file is called:

- Member HLI000, HLI00FRE (for French), or HLI00SPA (for Spanish) in ERRORS.DATA under z/OS.
- hli0000.err under the WebFOCUS Reporting Server and WebFOCUS.

In case of fatal errors, HLI writes error information to ddname HLIERROR or HLIPRINT for Mainframe FOCUS or to the hliprint.log file under the WebFOCUS Reporting Server and WebFOCUS.

Creating an Executable HLI Program

Before you can run HLI, you must link your program with the HLI routines needed to execute the HLI commands. You do this by constructing a DLL under WebFOCUS or a load module under z/OS. The result is a module that contains all of the programs required to run your HLI application.

> **Note:** If an existing HLI program is not LE-compliant, it must be made so. Once an HLI application is LE-compliant, dynamic HLI programs run without change. Static HLI programs should be re-linked.

Constructing a DLL Under ibi WebFOCUS

A script named gencpgm can assist in simple compilation of C programs. The script is located in the bin directory of EDAHOME. Its basic function is to build a dynamically loadable library program.

There is no requirement that gencpgm is used in actual program creation, only that a given program be a properly compiled and linked as a loadable library program. In addition, the physical name of the routine and the initial entry point must match. If the routine is to call other routines, the called routines must either be included in the module, or the calling routines must include *loader* logic. The results of the routine must also be passed back as the last argument of the function call.

The gencpgm script is written for simple compilation cases. Complex cases such as multiple sources, including library locations, ordering of libraries, special compilers, and linker options are not handled and are up to the developer to create their own build scripts.

In complex cases, the gencpgm script may be used as a model for forming an application-specific script.

For more information about GENCPGM, see Using the GENCPGM Build Tool

Compile and Link a C Program

Procedure

1. Copy your .c sample and gencpgm.sh (if not using the full path name) to a working directory.
2. Issue environment variables for the EDAHOME and EDACONF directories. For example:

   ```
   export EDAHOME=/home/iadmin/ibi/srv77/home
   export EDACONF=/home/iadmin/ibi/srv77/wfs
   ```

3. Compile the program using gencpgm with the -m hli switch.

   ```
   gencpgm.sh -m hli programname.c
   ```

   Where:

   programname

   Is the name of your c program. An executable for *programname* will be created in your current directory plus a DLL helper script (called *programname*.sh) that sets environment variables for runtime and invokes the actual program.

4. Run the program with the following commands.
   a. To run locally:

      ```
      programname.sh
      ```

   b. To run on a FOCUS Database Server (FDS):

      ```
      programname.sh fdsname
      ```

Result

If you want to see the linking options used, run gencpgm with the -x switch.

Constructing a Load Module Under z/OS

Under z/OS, to construct a load module for a new HLI application, you must link-edit your program with HLI routines.

As of FOCUS Release 7.6, all HLI programs running under z/OS must be LE-compliant and linked with AMODE 31, RMODE ANY. Prior to exiting the program, HLIEND must be called.

The link-edit JCL should contain the following:

```
INCLUDE FOCLIB(HLIFOCUS)
ENTRY mynameNAME myprog(R)
```

where:

FOCLIB

Must be allocated to the FOCUS load library.

myname

Is a user-defined name for the entry point in the program.

myprog

Is a user-defined name for the resulting load module. This module will replace any member with the same name in the SYSLMOD library. This name can be the same as *myname*.

Note: If you receive an error message about duplicate entry points, you may have chosen a reserved name (such as HLIMAIN) that is already used by FOCUS, and you must change it.

Creating a Load Module Under z/OS

The following is sample link-edit JCL that creates a load module from a FORTRAN HLI program called MYPROG:

```
//LINKIT    EXEC  PGM=IEWL,PARM='MAP,LET,LIST,SIZE=1024K'
//SYSLIB    DD    DISP=SHR,DSN=SYS1.FORTLIB    <=FORTRAN run-time
library
```

```
//              DD    DISP=SHR,DSN=CEE.SCEELKED
//OBJECT     DD    DISP=SHR,DSN=prefix.MY.OBJ   <=HLI program object code
//SYSLMOD    DD    DISP=SHR,DSN=prefix.MY.LOAD  <=Output library
//FOCLIB     DD    DISP=SHR,DSN=FOCLIB.LOAD     <=FOCUS library
//SYSUT1     DD    UNIT=SYSDA,SPACE=(CYL,3)
//SYSPRINT   DD    SYSOUT=*
//SYSLIN     DD    *
  INCLUDE OBJECT(MYPROG)         <=Include HLI object code
  INCLUDE FOCLIB(HLIFOCUS)       <=Include the HLI stub from FOCUS
  ENTRY MYPROG                   <=Declare entry point
  NAME MYPROG(R)                  <=Create member called MYPROG in
SYSLMOD
```

HLI Allocations

At run time, you must have allocated the following ddnames, in addition to the file itself.

STEPLIB

Must be allocated to the library containing the linked load module.

FOCLIB

Must be allocated to the standard FOCUS load library.

SYSPRINT

Must be allocated to the output job stream.

MASTER

Must be allocated to a library containing the Master Files of all FOCUS databases accessed by your HLI program.

ERRORS

Must be allocated to the standard FOCUS ERRORS.DATA library.

You should also allocate the following at run time:

HLIERROR

Must be allocated to a sequential data set with RECFM F and LRECL 80, or to the job output stream. HLIERROR should be allocated if you want a log of any fatal internal FOCUS errors that occur (for information, see HLI Status Return Codes.)

HLIPRINT

Must be allocated to a sequential data set or to the job output stream. HLIPRINT should be allocated if you are using the ECHO or STAT options in your HLI program (for information, see Using HLI), or if you would like a log of any fatal FOCUS errors and you have not allocated HLIERROR.

HLIPRINT should be allocated with LRECL 88 if you are using the ECHO option, or LRECL 133 if you are using the STAT option. The RECFM should be FB.

If you are using HLIPRINT for diagnosing problems with your HLI program, the blocksize should be equal to the LRECL, in which case, the RECFM is F.

If you are using the STAT option for performance analysis, you will want to specify a very large HLIPRINT blocksize so that the HLIPRINT trace has a minimal impact on the performance of your HLI program.

Note: Any databases you intend to access locally (that is, without the Simultaneous Usage facility) should be allocated with the parameter DISP=OLD. You should not allocate databases you intend to access through a central SU sink machine. If the database is to be used locally but is under the control of a sink machine, it should be allocated with DISP=SHR.

Sample HLI Batch Job

The following is a sample JCL for a typical HLI batch run:

```
//JOBNAME     EXECPGM=MYPROG
//STEPLIB     DD DSN=prefix.MY.LOAD,DISP=SHR  <=Previously linked lib
//FOCLIB      DD FOCLIB.LOAD,DISP=SHR         <=FOCUS load library
//SYSPRINT    DD SYSOUT=A
//MASTER      DD DSN=MASTER.DATA,DISP=SHR     <=Master files to be used
//CARD        DD DSN=CAR.FOCUS,DISP=OLD       <=Database to be used
//HLIPRINT    DD *,DCB=(LRECL=88,RECFM=FB,BLKSIZE=88)
//ERRORS      DD DSN=ERRORS.DATA,DISP=SHR
```

HLI and Simultaneous Usage of FOCUS Databases

When a FOCUS database is on a FOCUS Database Server (FDS or sink machine), the Host Language Interface allows two or more programs to operate on that FOCUS database at the same time (this is called Simultaneous Usage or SU). Each user is unaware of other users and may retrieve, add, delete, or change data independently. All users share one copy of the database and communicate to it through the usual HLI routines. The only changes necessary are to place the characters 'SU ' (requires two trailing blanks) in word 6 of the FCB and place the following in FCB words 9 and 10:

- The ddname of the communication file under z/OS FOCUS.
- The FDS node name, FOCSU001, under the WebFOCUS Reporting Server and WebFOCUS.

For more information on using SU with HLI, see the version of the *Simultaneous Usage Manual* for your operating system.

Using the SU Profile

The Simultaneous Usage Profile (SU Profile) enables you to set several parameters for the FOCUS Database Server (sink machine) in a profile. These parameters are also used to control a local HLI program.

Under z/OS FOCUS, the profile is member HLIPROF in a PDS allocated to the ddname FOCEXEC in the sink job. The DCBs for this PDS are the same as for any FOCEXEC PDS.

Under the WebFOCUS Reporting Server and WebFOCUS, the HLI program processes the server profile, edasprof.prf first. Then it processes the SU profile, suprof.prf. Most applications use edasprof to customize parameters. Use suprof.prf if you need to override settings for HLI only, for example, for application pathing to find files.

Note: Do not put USE commands in any profile.

You can include the following commands in the sink machine profile, although these settings offer no advantage to normal processing in the WebFOCUS Reporting Server and WebFOCUS:

```
SET BINS = nn
SET CACHE = nn
```

where:

SET BINS = nn

Controls the number of I/O buffers for the sink machine. A maximum of 63 bins (which are shared by all users) are allowed. Set BINS to the maximum for optimal performance. If BINS is not set in the profile, the sink calculates how many BINS to allocate based on the available storage.

SET CACHE = nn

Cache memory buffers FOCUS database pages between disk and BINS and reduces I/O to disk. Using the SET CACHE command in the profile keeps the entire database in memory and, therefore, improves performance. The default is no cache memory. See for information on how to use cache memory with FOCUS files.

Some of the commands useful for describing file search paths under the WebFOCUS Reporting Server and WebFOCUS include APP commands:

```
APP PATH
APP HOLDMETA
```

For information about these application commands, see the WebFOCUS *Developing Reporting Applications*, the FOCUS *Developing Applications* manual, or the *Server Administration* manual. HLI must be able to find the Master File and FOCUS database file for any file it attempts to open, otherwise the OPN command fails.

Multi-Threaded HLI/SU Reporting Facility (Mainframe FOCUS Only)

FOCUS for Mainframe includes the Multi-Threaded HLI/SU Reporting Facility, which enables users to directly access a central database from an HLI program, thereby bypassing the sink machine.

This feature is used in read-only mode, and all modifications to the database are routed through the sink machine. In addition, a single Multi-Threaded HLI/SU program cannot open a file simultaneously in both read-only and read/write mode. Separate FCBs must be declared for local reporting from and updating the centrally controlled database.

When you use this feature, the database updates and read operations are handled separately, without synchronization. So, for example, you may update a segment and then read it back and not see the update reflected immediately.

Multi-Threaded HLI/SU Reporting Under z/OS FOCUS

When using Multi-Threaded HLI/SU Reporting under FOCUS on z/OS, the central database is accessed locally, bypassing the sink machine; for database modifications, the central database is accessed through the sink machine. In the HLI environment, the FCB is modified to denote the parallel configuration (in the FOCUS environment, a USE command accomplishes this).

The specific steps follow (assume the sink job is running on user ID SINKMA):

1. Allocate the database and the communication data set in the sink job using DISP=SHR. For example:

   ```
   //CAR     DD   DSN=SINKMA.CAR.FOCUS,DISP=SHR
   //FOCUS   DD   DSN=SINKMA.FOCSU.DATA,DISP=SHR
   ```

2. Allocate the database and the communication data set for the HLI program using DISP=SHR. For example:

   ```
   ALLOC F(CAR)    DA(SINKMA.CAR.FOCUS) SHR
   ALLOC F(FOCSU)  DA(SINKMA.FOCSU.DATA) SHR
   ```

   or

   ```
   //CAR    DD   DSN=SINKMA.CAR.FOCUS,DISP=SHR
   //FOCUS  DD   DSN=SINKMA.FOCSU.DATA,DISP=SHR
   ```

3. Set up the FCB according to the following steps:

a. Set FCB word 6 (SU) to SULO.

b. Set FCB word 9 (SINKID) to the ddname of the FOCUS communication dataset.

Preparing an FCB for Multi-Threaded HLI/SU Reporting Under z/OS FOCUS

Assume FOCSU is the communication data set and the following is the COBOL HLI FCB:

```
01  FCB.
    05  FCB-FN          PIC X(08) VALUE SPACES.
    05  FCB-FT          PIC X(08) VALUE SPACES.
    05  FCB-FM          PIC X(04) VALUE SPACES.
    05  FCB-SU          PIC X(04) VALUE SPACES.
    05  FCB-DN          PIC X(08) VALUE SPACES.
    05  FCB-SINKID      PIC X(08) VALUE SPACES.
    05  FILLER          PIC X(28) VALUE SPACES.
    05  FCB-ECHO        PIC X(04) VALUE "ECHO".
    05  FILLER          PIC X(20) VALUE SPACES.
    05  FCB-STATUS      PIC S9(5) COMP-3 VALUE +0.
    05  FILLER          PIC X(104) VALUE SPACES.
```

The FCB would be set up as follows:

```
105-SET-UP-FCB-HLI-SU-MVS.
    MOVE "CAR"     TO FCB-FN.
    MOVE "SULO"    TO FCB-SU.
    MOVE "FOCSU"   TO FCB-SINKID.
```

HLI Command Summary

This chapter provides summary charts of HLI commands and parameters followed by a description of each HLI command in alphabetical order.

HLI Command Summary Chart

HLI commands must be entered as 4-character operands. Since all HLI commands (except for the INFO command) are three characters long, they must be padded with a trailing blank.

Command Name	Meaning
`CHA`	(Change) changes data values within one segment.
`CLO`	(Close) closes the FOCUS database file. If there is more than one FCB per file, closes the one that is open.
`DEL`	(Delete) deletes a segment and all of its descendants.
`FSP`	(First physical) retrieves the first physical qualified or unqualified segment of the target segment in the file.
`FST`	(First) retrieves the first logical qualified or unqualified occurrence of a target segment within a parent segment.
`INFO`	(Information) returns file description information such as segment names, field names, and formats.

`INP`	(Input) includes a new instance of the target segment.
`NEX`	(Next) retrieves the next logical qualified or unqualified occurrence of a target segment within an anchor segment.
`NXD`	(Next indexed) locates a segment instance directly based on an indexed key value.
`NXK`	(Next via key) goes back to a prior segment located by a saved address key (the backkey).
`NXP`	(Next physical) retrieves the next physical qualified or unqualified segment instance of the target segment.
`OPN`	(Open) opens the FOCUS database for use.
`SAV`	(Save) saves changed values. Forces a write to the file of changes since the last save.
`SHO`	(Show) resets the order and number of fields currently activated.

HLI Parameter Description Chart

The following chart lists the parameters used in HLI calls:

Parameter	Meaning
`workarea`	A buffer in the application program that receives data from HLI.

Parameter	Meaning
`target`	The 8-character name of the segment sought, included, changed, or deleted. This is a SEGNAME value in the Master File.
`anchor`	The 8-character name of a segment to use as a base point from which to search for another segment. This is a SEGNAME value in the Master File or the word SYSTEM.
`numb`	The number of non-blank test conditions needed to qualify a record for retrieval, or the number of changes to be made.
`backkey`	An address in the file that allows you to move back to a previously retrieved segment with the NXK command.
`testrelations`	A structure or array of four bytes for each active field containing test relations necessary to qualify a record. Test relations must always be capital letters padded with blanks to four characters. The array should contain one of the test relations below. • **EQ.** Equal. • **NE.** Not equal. • **GE.** Greater than or equal. • **LE.** Less than or equal. • **GT.** Greater than. • **LT.** Less than. • **CO.** Contains. • **OM.** Omits. • **' '.** Blank (everything qualifies).
`testliterals`	Structure or array that contains the literal values necessary for a qualifying record. The testliterals array is compared to the data in the file with the relations specified in the

Parameter	Meaning
	testrelations array. When the test is successful, the action of the command is executed.
`changelist`	A structure or array of four characters per element in which the value of an element is EQ for fields that should be changed or blank for unchanged fields.
`changeliterals`	The new values that are to replace the values currently in the file.
`savearea`	A 32-byte area used to save information when an indexed retrieval is requested (NXD). This should be set to binary zero for the first occurrence.
`nrepeat`	The number of times to repeat the retrieval of a record (1 to 255). Used only by the NEX or FST command.
`showlist`	A structure or array containing the names of the data fields to be activated. There are 12 bytes for each field name.
`showcount`	A 4-byte integer representing the number of fields in the show list.
`option`	A 4-byte integer used by some commands.
`inpliterals`	New data to be included in the database.
`null`	Eight blank characters.
`information area`	A record area in which information about the file is returned.

Note: The changeliterals, inputliterals, testliterals, and workarea arrays all have the same format. This format is defined by the length and format of the fields in the show list.

Alphabetical List of HLI Commands

This section describes the use and function of each HLI command. Commands are listed in alphabetical order.

CHA (Change) Command

C Syntax:	`edahliCall(hliHandle, 7, cha, &fcb, &workarea, target, n, changelist, changeliterals);`
FORTRAN Syntax:	`CALL FOCUS (cha, fcb, workarea, target, null, n, changelist, changeliterals)`
COBOL Syntax:	`CALL 'FOCUS' USING cha fcb workareatargetnullnchangelistchangeliterals.`
PL/1 Syntax:	`CALL FOCUS (cha, fcb, workarea, target, null, n, changelist, changeliterals);`
Function:	Changes fields that have EQ in the changelist array to the values in the changeliterals array. Returns a copy of the changed segment image in the work area. The changeliterals array must have the new data located in the position of the field as described in the SHO field layout. *n* is the number of EQs in the changelist array for this segment.

There are four bytes in the changelist array for each data field in the show list. Only fields in the named target segment are changed. The work area must be at least as large as one retrieved record.

When the CHA command is issued, the non-blank changelist array entries indicate which fields are to be changed. Hence, different changes can be accomplished by either supplying other changelist arrays (such as CHALST1, CHALST2) or replacing the blank or non-blank values in one area.

When the CHA command is used on a FOCUS Database Server (sink machine), the server tests to see that the record is current before applying changes. This ensures that no changes are made if another user has already changed the record.

Note: *null* is an 8-byte field of spaces used for positioning.

The following diagram illustrates using the CHA command

	Show list	Change list	Change literals
SEGNAME=ORIGIN	COUNTRY		GERMANY
SEGNAME=COMP	CAR	EQ	VW
SEGNAME=BODY	MODEL		BUG
	SEATS	EQ	4
	DEALER_COST	EQ	3187

Consider the following command:

```
CALL FOCUS (CHA , FCB, WKAREA, BODY   , null, 2, CHALST, CHALIT)
```

The two fields named SEATS and DEALERCOST in the target segment are changed to the new values provided in the changeliterals array.

The car is not changed to VW because it is not in the target segment.

Note that the 2 in the argument list represents the number of non-blank entries in the *changelist*. The fields corresponding to those entries are changed to the values specified in *changeliterals*.

CLO (Close) Command

C Syntax:	`edahliCall(hliHandle, 2, clo, &fcb);`
FORTRAN Syntax:	`CALL FOCUS (clo, fcb)`
COBOL Syntax:	`CALL 'FOCUS' USING clofcb.`
PL/1 Syntax:	`CALL FOCUS (clo, fcb);`
Function:	The FCB is removed from active use. Any outstanding changes to the data are written to the disk, and all internal buffered storage space is returned to the system pool. If there is more than one FCB open for a file, a CLO command must be issued for each open FCB.

DEL (Delete) Command

C Syntax:	`edahliCall(hliHandle, 3, del, &fcb, target);`

FORTRAN Syntax:	`CALL FOCUS (del, fcb, target)`
COBOL Syntax:	`CALL 'FOCUS' USING delfcbtarget.`
PL/1 Syntax:	`CALL FOCUS (del, fcb, target);`
Function:	Deletes the target segment and all of its descendant segment instances from the file.

FSP (First Physical) Command

C Syntax:	`edahliCall(hliHandle, 7, fsp, &fcb, &workarea, target, ntest, testrelations, testliterals);`
FORTRAN Syntax:	`CALL FOCUS (fsp, fcb, workarea, target, ntest, testrelations, testliterals)`

COBOL Syntax:	`CALL 'FOCUS' USING`
	`fsp`
	`fcb`
	`workareatargetntesttestrelationstestliterals.`

PL/1 Syntax:	`CALL FOCUS (fsp, fcb, workarea, target, ntest, testrelations, testliterals);`
Function:	Locates and retrieves the first physical occurrence of the target segment. After the command is executed, the parents can be retrieved with the FST (First) command. Descendants can be retrieved with either FST or NEX.

	Unqualified Retrieval: No test conditions apply if the value of *ntest* is 0. Therefore, the *testrelations* and *testliterals* arrays are ignored. **Qualified Retrievals**: When *ntest* is not zero, it specifies the number of non-blank test conditions in the *testrelations* array. These tests are used to compare the values in the database to the values in the *testliterals* array. The record is retrieved if it passes all the tests. For information about the comparison between logical commands and physical commands, see Introduction to HLI.

FST (First) Command

C Syntax:

```
edahliCall(hliHandle, 10, fst, &fcb, &workarea,
target, anchor, ntest, testrelations,
testliterals, null, nrepeat);
```

FORTRAN Syntax:

```
CALL FOCUS (fst, fcb, workarea, target, anchor,
ntest, testrelations, testliterals, null,
nrepeat)
```

COBOL Syntax:

```
CALL 'FOCUS' USING
fst
fcb
workarea
target
anchor
ntesttestrelationstestliteralsnullnrepeat.
```

PL/1 Syntax:

```
CALL FOCUS (fst, fcb, workarea, target, anchor,
ntest, testrelations, testliterals, null,
nrepeat);
```

Function:	Retrieves the first target segment within the anchor segment that meets the qualifying conditions. (There is no change in the anchor position.) When a record is retrieved, the status in the FCB is 0. If no record is found, the status is 1. If the anchor segment name is SYSTEM, the first logical target segment in the file will be retrieved. If the value of ntest is 0, the testrelations and testliterals parameters are ignored. The target segment may be below or above the anchor segment. When a segment is retrieved directly, through a NXP (Next Physical) command or through an NXD (Next through Index) command, the balance of the attached segments can be obtained via the FST command. The nrepeat option lets you retrieve multiple records in a single call. (See the documentation for the NEX command for a description of qualified and unqualified retrieval.)

INFO (Information) Command

C Syntax:	`edahliCall(hliHandle, 4, info, &fcb, &workarea, option);`
FORTRAN Syntax:	`CALL FOCUS (info, fcb, workarea, option)`
COBOL Syntax:	`CALL 'FOCUS' USING infofcbworkareaoption.`
PL/1 Syntax:	`CALL FOCUS (info, fcb, workarea, option);`
Function:	Returns information about the location, length, and format of each field in the current SHO command.

Note: When a file is first opened, all fields are considered shown; by issuing the INFO command immediately after the OPN command, complete file information is available.

The field information returns the current show list (a list of names of fields that are activated, in their order in the work area). The segment information always returns the full list of segment names from the Master File.

Option Data Returned

0 All field information in the show list.

1 All segment information.

Layout of Return Work Area in INFO Command When Option=0

The information returned in the work area contains 12 words (48 bytes) for each field. The order of the fields is the order of the current SHO command.

Comment	Word	Description
	1	Number of fields which follow.
	2-3	Name of segment.
Words 2 through 13 repeat for each field.	4-6	Name of data field.
	7-9	Alias name of data field.
	10-11	Usage format of data field.
	12	Length of field in bytes (binary integer).
	13	Starting byte offset in current work area (binary integer).

Layout of Return Work Area in INFO Command When Option=1

When the value of *option* is 1, returns information about the segments and file structure only. This information is independent of the current SHO command. The information returned in the work area contains 16 words per segment (643 bytes). Note that the size of this area differs from the size when option=0.

Comment	Word	Description
Words 2 through 17 repeat for each field.	1	Number of segments.
	2-3	Name of segment.
	4-5	Name of parent of segment.
	6	Starting byte in real segment of first field.
	7	Length of segment in bytes.
	8	Segment type, alpha name.
	9-11	Cross-reference key field name.
	12	Number of sequence keys.
	13-17	Reserved.

The last argument in the INFO command must be either 0 or 1.

INP (Input) Command

C Syntax:	`edahliCall(hliHandle, 5, inp, &fcb, inputliterals, target, n);`
FORTRAN Syntax:	`CALL FOCUS (inp, fcb, inputliterals, target, option)`
COBOL Syntax:	`CALL 'FOCUS' USING inpfcbinputliteralstargetoption.`
PL/1 Syntax:	`CALL FOCUS (inp, fcb, inputliterals, target, option);`
Function:	Using the values in the inputliterals array that correspond to the fields in the target segment, a new segment instance is created in the file. Values not provided for the segment are given default values: blank if alphanumeric and 0 if numeric.

If the segment type is S0 or blank, the possible values of *n* are:

`0`	Inserts the new segment after the current segment, if the key fields are the same.
`1`	Inserts the new segment before the current segment, if the key fields are the same.

If the segment type is S*n* or SH*n* the value of *n* can be:

`0`	No test is made for duplicate values.

2	If the segment has a key, rejects the new segment if the key already exists.

A segment cannot be inserted if it would break the sort sequence of existing segments. The proper place for a keyed segment is found automatically regardless of the value of the *option* parameter. Option 2 rejects duplicate keys. Using this option is much faster than locating a segment and, if it does not exist, adding it.

Only the target segment is included. You must add descendant segments using separate INP commands. The new segment becomes the current position. Unique child segments are not automatically retrieved with their parent segment. To retrieve a unique segment instance, issue a separate retrieval command with the unique segment as the target.

Prior to issuing an INP command for the unique segment instance, you must check for the existence of a unique segment instance for the parent. The return code from this call (FST or NEX with the unique instance as the target) determines if the INP is appropriate. If you issue an INP command and a unique segment instance already exists for the specified parent, a duplicate unique segment instance is created for that parent segment. (The REBUILD facility eliminates the original unique segment instance.)

NEX (Next) Command

C Syntax:

```
edahliCall(hliHandle, 10, nex, &fcb, &workarea,
target, anchor, ntest,testrelations,testliterals,
null, nrepeat);
```

FORTRAN Syntax:

```
CALL FOCUS (nex, fcb, workarea, target, anchor,
ntest, testrelations, testliterals, null,
nrepeat)
```

COBOL Syntax:

```
CALL 'FOCUS' USING
nex
fcb
workarea
```

	`target anchorntesttestrelationstestliteralsnullnrepeat.`
PL/1 Syntax:	`CALL FOCUS (nex, fcb, workarea, target, anchor, ntest, testrelations, testliterals, null, nrepeat);`
Function:	The next target segment instance within the anchor segment instance that meets the qualifying conditions, if any, is retrieved. If a record is retrieved, the status returned is 0. If there are no more target segments available within the anchor segment, the status returned is 1. If an error occurred, the status is greater than 1 and is returned with an error number. The retrieved segment instances are placed in the work area as requested in the SHO command.

Unqualified Retrieval: If the value of *ntest* is 0, there are no testing operations on the records retrieved, and the contents of the testrelations and testliterals arrays are ignored.

Qualified Retrieval: A value of *ntest* greater than 0 specifies the number of non-blank test conditions in the testrelations array. These tests are used to compare the values in the database to the values in the testliterals array. Only records that pass all tests are retrieved.

Multiple Record Retrievals: The default, when *nrepeat*=1, retrieves only the next target or path of segments meeting the test conditions, if any. One HLI call can retrieve up to 256 segment instances depending on the *nrepeat* value. This argument must be a 4-byte binary integer from 1 to 255. The retrieved records are placed in the work area one after the other. They are separated by eight bytes, where the backkey is placed. The effect of the *nrepeat* parameter is the same as if *nrepeat* identical calls were issued, but is faster.

The number of records actually retrieved is placed in the FC B at word 25. For example, if 15 records were requested but only 12 were available, *nreturned* (FCB[25]) would equal 12. At the conclusion of the call, the current position of the last record retrieved is placed in the backkey in the FCB. The backkey for each record is also placed in an 8-byte area following each record. It can be used to reposition the file pointer to the record it followed.

NXD (Next Through Index) Command

C Syntax:	`edahliCall(hliHandle, 9, nxd, &fcb, &workarea, target, field, ntest, testrelations, testliterals, savearea);`
FORTRAN Syntax:	`CALL FOCUS ( nxd , fcb , workarea , target , field,ntest,testrelations,testliterals,savearea)`
COBOL Syntax:	`CALL 'FOCUS' USING nxd fcb workarea target field ntest testrelations testliterals savearea.`
PL/1 Syntax:	`CALL FOCUS ( nxd , fcb , workarea , target , field ,ntest,testrelations,testliterals,savearea);`

Function:	The target segment is the segment that contains the indexed field; it is retrieved using the index of the field named in the *field* argument. The value of *ntest* must be at least 1; the literal value of the indexed field must be in the testliterals array, and the testrelations array must have a value of EQ for the indexed field. The first time the specific value of the key is provided, the save area must be all binary zeros (the save area is 32 bytes long). When the save area is binary zeros, the first occurrence of the key value is retrieved, regardless of the current file position. Only the target segment is retrieved, and it becomes the anchor point for obtaining related parent or descendant segments. If there are multiple segments matching the key value, the next match is retrieved by another HLI call with the NXD command that has the same save area as the last call. Do not clear the save area between calls, as it contains the information necessary for processing all occurrences. When a new key value is to be supplied, start with a cleared save area. There may be other qualifying test conditions on the target segment. These are provided in the testrelations and testliterals arrays in the manner specified in the NEX command. **Note:** The *field* argument must be 12 characters in length and contain the field name of a field whose index is to be used. If the field name is less than 12 characters long, it should be padded with trailing blanks.

NXK (Next Through Backkey) Command

C Syntax:

```
edahliCall(hliHandle, 5, nxk, &fcb, &workarea,
target, backkey);Commands (HLI):NXKNXK command
```

FORTRAN Syntax:	`CALL FOCUS (nxk,fcb,workarea,target,key)`
COBOL Syntax:	`CALL 'FOCUS' USINGnxk fcb workarea target key.`
PL/1 Syntax:	`CALL FOCUS (nxk,fcb,workarea,target,key);`
Function:	After every successful HLI call (status code 0), the backkey containing the database address of the target segment is stored in the FCB at bytes 61 to 68. If these eight bytes are saved and later provided as an argument to the NXK command, the former target is returned. If this segment is a descendant segment, its parent is also retrieved, if its position was established logically by following pointers (using NEX or FST). If the descendant segment was located with a NXD or NXP command, the pointers are not available for inserting or deleting instances at the target level or above. The target segment in the call must correspond to the restored backkey value, (that is, the target segment retrieved by the call that formed the backkey). This target segment is associated with the backkey, and it must be provided in the NXK call. It is the programmer's responsibility to save the target segment name associated with the backkey (using the SAV command).

NXK retrieval allows a program to work in several separate positions in a given file without opening an FCB for each position. The processing sequence is:

1. Process current position.
2. Save current position.
3. Move to another position.
4. Save new position.
5. Restore previous position.

Users may construct their own index to a file and store the index value and key in another FOCUS or external file. In this way, files may be interconnected, based on arbitrary rules, and not require common data values as does the FOCUS cross- reference facility.

NXP (Next Physical) Command

C Syntax:	`edahliCall(hliHandle, 7, nxp, &fcb, &workarea, target, ntest, testrelations, testliterals);`**`Commands (HLI):NXPNXP command`**
FORTRAN Syntax:	`CALL FOCUS ( nxp , fcb , workarea ,target,ntest,testrelations,testliterals)`
COBOL Syntax:	`CALL 'FOCUS' USING nxp fcb workarea target ntest testrelations testliterals.`
PL/1 Syntax:	`CALL FOCUS ( nxp , fcb , workarea ,target,ntest,testrelations,testliterals);`
Function:	The next physically adjacent occurrence of the target segment

	Unqualified Retrieval: When the value of *ntest* is 0, no test conditions apply. Therefore, the testrelations and testliterals parameters are ignored. **Qualified Retrieval:** A value of ntest greater than 0 specifies the number of non-blank test conditions in the testrelations array. These tests are used to compare the values in the database to the values in the testliterals array. If all the tests pass, the record will be retrieved.

For example, consider the following file structure:

FIGURE12

You issue the NXP command to retrieve segment B. To retrieve the two related segments, A (B's parent) and C (B's descendant) you use the FST command, as follows:

Command	Target	Anchor	Current Position
`NXP`	`B`		At B1: New segment position.
`FST`	`A from`	`B`	At A1: Parent of B. B1: Current position in the segment does not change because it is anchored there.
`FST or NEX`	`C from`	`B`	At A1: Parent does not change. B1: Current position in the segment does not change because it is anchored there. C1: Get first child.

OPN (Open) Command

C Syntax:

```
edahliCall(hliHandle, 2, opn, &fcb)
```

FORTRAN Syntax:	`CALL FOCUS (opn, fcb, option)`
COBOL Syntax:	`CALL 'FOCUS' USING opnfcboption.`
PL/1 Syntax:	`CALL FOCUS (opn, fcb, option);`
Function:	The Master File and data files are located and opened. This must be the first call to HLI in an application program for each database to be processed. The *option* argument must be a 4-byte integer value of either 0 or 1. If the value is 0, the FOCUS file must already exist. A status return code of 0 means the file has been successfully opened. If the status return code is not 0, the file has not been successfully opened and any subsequent calls for its use will be rejected. A total of 255 FOCUS files and their Master Files may be open simultaneously. After this command is successfully issued and before any SHO commands are issued, all the fields in the file are considered on the show list. **Note:** • Some compilers allow a variable number of arguments to be passed in a subroutine call. If your compiler gives you errors, then all calls to FOCUS must contain ten parameters, providing the unused parameters are set to 0. • When you have declared multiple FCBs for a file, each FCB must be opened. • Cross-referenced files are opened automatically with read-only access. If you need to update values in a file, open the file before you cross-reference it.

SAV (Save) Command

C Syntax:	`edahliCall(hliHandle, 2, sav, &fcb)`
FORTRAN Syntax:	`CALL FOCUS (sav,fcb)`
COBOL Syntax:	`CALL 'FOCUS' USINGsav fcb.`
PL/1 Syntax:	`CALL FOCUS (sav,fcb,option);`
Function:	All data values which have been changed via the CHA, DEL, or INP commands are written to disk storage. Current position is unchanged. When a SAV command is issued for one file, all the changes for all the files are saved. An implicit SAV is done when a CLO command is issued. **Note:** When FOCUS accepts transactions, it does not write the transactions to the database immediately; rather, it collects them in a buffer. FOCUS writes all the transactions from the buffer to the database at the same time when any of the following occurs: • A program issues a SAV command. • A program closes down a FOCUS file by issuing a CLO command.

SHO (Show) Command

C Syntax:	`edahliCall(hlihandle, 4, sho, &fcb, showlist, &numb);`
FORTRAN Syntax:	`CALL FOCUS (sho, fcb, names, numb)`
COBOL Syntax:	`CALL 'FOCUS' USINGsho fcb names numb.`
PL/1 Syntax:	`CALL FOCUS (sho,fcb,names,numb);`
Function:	The list of field names supplied in the area called *names* (the show list) controls the layout of the work area, and subsequent records retrieved will return the data to the work area in the new order. When new field values are supplied for input, they are taken from the work area record in the order set by this command. In addition, this order is used when test conditions are supplied to qualify a record. Each field name occupies 12 characters in the show list. The value of *numb* is a 4-byte integer that specifies the number of field names supplied in the show list. It is highly recommended that a SHO command be issued after each open to ensure the format of the work area. The current fields defined by the show list are made available to an application program by issuing the INFO command with the option for field information. Range checking can be accomplished by specifying a field in the show list twice and applying an inequality test to these two fields. If duplicate occurrences of a field exist in the show list, the

	data for the last occurrence of the field is saved in the database on an input or change if duplicate EQ conditions exist in the changelist array.

HLI Status Return Codes

This appendix provides a list of status return codes and their explanations.

You can find the status code in word 24 of the FCB.

HLI Return Code Chart

Return Code	Meaning
`0`	Command executed normally.
`1`	Command executed normally, but no segments were retrieved. The current position is unchanged. Either the qualifying conditions failed to locate the desired record, or an end-of-chain condition occurred (that is, no more target segment instances exist within the anchor segment).
`760`	Command not recognized. An invalid HLI command was issued (for example, NXT instead of the NEX command).
`761`	Too few arguments have been provided in the HLI command.
`762`	The command cannot be executed because the file has not yet been opened (that is, no OPN command has been issued).
`763`	HLI requires more virtual storage to operate. Restart the HLI program in a larger region.
`764`	The requested Master File cannot be found.

Return Code	Meaning
765	The requested Master File cannot be found by the sink machine.
766	No data is found in the specified file. The file must first be initialized by the CREATE command even if no data is entered.
767	An error was encountered in the Master File.
768	An invalid parameter was encountered in an HLI call (for example, NTEST is less than zero).
769	The field name referenced on a SHO command does not exist in the Master File. Check the spelling or the structure of the NAMES array passed into the SHO command.
770	The file specified in the CLO CMD command is not open. The CLO CMD command has been ignored.
771	Segment name not recognized. The segment specified as a TARGET or an ANCHOR segment is not found in the Master File.
772	The HLI command is recognized but is not yet supported.
773	No current position has been established from which to execute the command.
774	Test relation not recognized. An invalid test relation was used in an HLI call. Possibly the number of tests was specified incorrectly. Valid test conditions are LT, LE, GT, GE, EQ, NE, CO, and OM.
775	Improper use of virtual segment. An attempt was made to change or improperly use a cross reference segment.

Return Code	Meaning
776	An attempt was made to include a second instance of a unique segment for a particular parent instance. The transaction is ignored.
778	The ANCHOR and TARGET segments specified do not lie on the same path in the file.
779	An error has occurred in the use of an indexed field, or a field named in an NXD command is not indexed.
780	On an NXD call, no tests on the target segment were provided.
781	No parent position has been established for the retrieval of a cross-reference segment. The key for the linked segment is not active, and no retrieval can be performed.
782	The password does not provide file access rights. Check the password provided in the FCB.
783	The command issued is not allowed with the current password. Check the password provided in the FCB.
784	The segment instance identified by the key values is already in the database. INP command does not allow duplicates. Analogous to FOCURRENT = 1 in MODIFY/SU. The database has not been changed.
785	Segment instances obtained with NXK or NXD may not be deleted.
786	Attempt to change the instance has not been performed, because the instance has been deleted by another user. (Analogous to FOCURRENT=2 in MODIFY/SU.)
787	An OPN command was issued for an FCB which was already open. Before reusing the FCB, it must be closed.

Return Code	Meaning
788	Too many files are open on the central database machine (limit 255) or too many FCBs are open (limit 4096).
789	Attempt to change the instance has not been performed because it has already been changed by another user. (Analogous to FOCURRENT=3 in MODIFY/SU.)
790	SU cannot be used with more than 1500 fields in a FOCUS file, including cross references.
792	Deprecated.
793	You cannot update, delete or include records for a file that has been specified as read-only for use with Multi-Threaded HLI/SU Reporting.
794	You cannot declare a file to be read via Multi-Threaded HLI/SU Reporting if it has been opened by another FCB not operating Multi-Threaded HLI/SU Reporting. You cannot open a file that will not be read via Multi-Threaded HLI/SU Reporting if it has already been opened by another FCB operating under Multi-Threaded HLI/SU Reporting.
798	The Master File read by the central data base machine is not the same as the one used by the source or local machine.
800	The central SU database machine has terminated because of a fatal error.
801	A communication error has occurred on the central SU database machine.
802	The central database machine has not been started or the wrong name has been used for the central SU database machine.

Return Code	Meaning
803	A communication error has occurred. Check to see that the sink machine is still active and, if not, restart it.
804	The central data base is not active. Check to see if it has been terminated, and restart it.
805	A fatal error has occurred on the sink machine. Check to see if the sink machine is active and, if not, restart it.
806	A fatal error has occurred on the sink machine. Check to see if the sink machine is active and, if not, restart it.
807	A limit of 64 sink machines can be accessed simultaneously by one user.

Using the GENCPGM Build Tool

The building and compilation of 3GL applications is platform-specific and sometimes driven by standards with which a site must conform in terms of programming style or managing programming source. Due to this wide variation, we only make recommendations, test certain languages, and provide limited examples with a script that minimally compiles the test examples.

The specific uses for 3GL programs and examples are documented elsewhere, but the general purposes are:

- To create and add a user-written routine to the functions of the product (also known as a FUSELIB).
- To create and customize user exits that provide special functions.
- To create CALLPGM programs that the server executes.

Using GENCPGM

GENCPGM is the general term for a series of platform specific scripts for compiling and linking 3GL programs (for example, C, COBOL, Fortran, Java, etc.) that interact with ibi® products.

The scripts and their associated platforms are:

- gencpgm.sh (UNIX, Linux, z/OS and OS400)
- gencpgm.bat (Windows)

The script for a given platform is located in the bin directory of the software installation directory (EDAHOME), except on a z/OS PDS deployment, where it is it in the member hlq.HOME.ETC(GENCPGM), and must be copied to and given execute privileges to be used under HFS.

Examples of the types of programs that can be built are:

- HLI applications to talk directly to FOCUS databases or servers using FDS access to FOCUS databases

- Call Java Adapter (CALLJAVA) applications which use Java classes to retrieve row(s) data
- Call Procedural Program Adapter (CALLPGM) applications which use a DDL to retrieve row(s) data.
- Subroutine applications which use a DLL to do specialized inline calculations for Dialogue Manager, DEFINEs or COMPUTEs.

From a technical perspective, the above list breaks down into 3 classes of 3GL programs that GENCPGM builds:

- Dynamic link libraries.
- Executables.
- Java applications as a class in a jar.

A dynamic link library is also known as a DLL and is generally thought of as a Windows specific term, however, there are equivalences on all other platforms. DLL libraries have an extension of .dll on Windows. On UNIX and USS, the term for DLL is shared library with an extension of .so. On OS400, a DLL is a service program (programs marked SRVPGM).

The GENCPGM scripts are solely supplied as an assist tool for building basic applications. The GENCPGM scripts are not intended to support all languages and complex cases, like building several objects all linked into a final program. The GENCPGM scripts actually depend on an appropriate native compiler and linker being installed and accessible. *Native* refers to the compiler of the operating system vendor (for example, Microsoft on Windows, IBM on AIX, and so forth). *Accessible* means that it is known to the registry on Windows or is in the PATH for other operating systems, and that it employs the normal program names used by the originating vendors (for example, cl.exe on Windows, cc on many UNIX systems, gcc on Linux, javac (and jar) for Java, and so forth). The compiler also needs to generate binaries that match the bit requirements of the application software (32-bit servers require 32-bit compilers), although some compilers control this using a switch (for example, -m32 and -m64).

Because of the widespread use of GNU GCC (which is free), the Windows and UNIX versions of GENCPGM also recognize and allow *gcc* as a compiler specification, although they still depend on gcc being in the path or, in the case of Windows, having the variable MINGWROOT set (see the Windows section for more details). In short, GENPGM is a build assistant to access existing compiler tools, but is not itself a compiler/linker and, as such, the user is responsible for having the appropriate compilers and linkers installed and accessible if GENCPGM compilation is needed. Note that many instances are strictly for build-time use, and the resulting binaries may simply be deployed thereafter if the

operating system and application bit requirements match (32-bit or 64-bit), and the deployment machines do not have compiler/linker requirements.

The use of GENCPGM as a build tool is not actually required for applications when proper build rules are followed, as implemented in GENCPGM and outlined in the build rules section. Since complex cases that use other languages or multiple sources are a legitimate requirement, it is left to the user to code and maintain their own build scripts for these cases and alternate languages (possibly using GENCPGM as a template and following the rules outlined in the build rules section).

It should also be noted that a subroutine is sometimes referred to by its former terminology of Fuselib or Fuselib Routine. From an application perspective (that is, a focexec) they are one in the same, however, FOCUS products used a single library to implement and store multiple routines where WebFOCUS uses individual libraries for each routine. This means older existing FOCUS libraries are not directly usable with WebFOCUS, but the underlying 3GL sources are usable and simply need to be built using the current methodologies documented here.

While there are a few platforms that need specific switch options, most switches and many languages work on most platforms. Concerning specific 3GL languages, C is the officially supported language on all platforms, other languages vary by platform as noted in the samples. Theoretically, any 3GL language that is capable of being compiled and linked into a DLL or executable and is capable of being used with ibi® products, however, GENCPGM is only coded for certain commonly used languages that we have easy access to and expertise in creating scripts and working samples. Requests for additional languages will be considered on a case by case basis.

GENCPGM is also used dynamically in the server product for the COMPILED DEFINE feature and as such the version in the EDAHOME directory should never be customized to prevent changes from affecting the COMPILED DEFINE feature. If you have customizations that you feel would be useful to others, they may be submitted via Customer Support for consideration as a permanent change.

USAGE Chart (Typical Syntax Plus Extended Options)

UNIX®, Linux®, IBM® i, USS:

```
{path}gencpgm.sh [-h] [-x] [-q] [-v] [-e] [-n] [-s script]
 [-H EDAHOMELIB] [-p LOADLIB] [-d directory|-w directory] [-g]
 [-c language] [-m application type]  [-b lib/srvprg] [-j jarname]
 {path}{program name}[.{extension}]
```

Windows:

```
{path}gencpgm.bat [-h] [-x] [-q] [-v] [-e] [-n] [-s script]
 [-d directory|-w directory] [-g]  [-c language]
 [-m application type] [-j jarname]
 {path}{program name}[.{extension}]
```

where:

Switch/Option	Description
-h	Outputs this Help text.
-x	Turns on set -x shell tracing to assist in debugging.
-q	Turns on quiet mode to redirect Microsoft® compiler/linker output to nul: on Windows. The switch does nothing on other platforms because the compliers and linkers on most other platforms are already "quiet".
-v	Turn on compiler/linker verbose options plus selective informational messages.
-e	Extended trace/compiler/linker info from just before compiler/linker step.
-H EDAHOMELIB	Server for z/OS® in a PDS deployment only. Indicates installation home {HLQ}.HOME of ETC.H for picking up standard C include files (needed for some samples).
-C EDACONFHLQ	Server for z/OS in a PDS deployment only. Indicates installation configuration {HLQ} of ETC for picking up standard ibi files (that is, server and communications configuration files).
-A APPROOTHLQ	Server for z/OS in a PDS deployment. Indicates installation configuration {HLQ} of APPS (APPROOT) for picking up application files for HLI applications.

Switch/Option	Description
-S SCRIPTSPDS	Server for z/OS in a PDS deployment only. Indicates PDS to copy application build JCL and run time execution scripts for batch JCL and interactive CLIST and REXX of the application.
-s *SCRIPT*	For z/OS PDS Unified Environment deployment. Indicates generate only (no compile/link) to a fully qualified PDS or dataset name.
-p LOADLIB	Server for z/OS in a PDS deployment only. Indicates JCL type compilation and points to the load lib to use. The load lib must be in run time STEPLIB.
-d *directory*	Work in the given directory. The C file should be in this directory. All resulting files will be generated in this directory. This is for COMPILED DEFINE purposes and not intended for customer use.
-w *directory*	Write final executable (and any helper scripts) in the given target directory.
-i *directory*	Include directory. Multiple uses allowed.
-n	No runner shell creation for api*, hli and odbc programs. Use to prevent overwrite of an existing shell that may have been customized.
-g	Generate a debuggable program by including debug switch in the compilation and link.

The -c option is described in the following chart:

Language	Compiler to use for a given language source.
cc	Use standard C compiler to compile "progname.c". C is the default compiler language.
assembler	Use assembler compiler. Only implemented for z/OS currently. HFS usage requires source to have a .s file extension.

Language	Compiler to use for a given language source.
fortran for f	Use default Fortran to compile Fortran with a .fortran extension. Use default Fortran to compile Fortran with a .for extension. Use default Fortran to compile Fortran with a .f extension. Supply explicit extension to override extension. If default compiler is not available, GNU g77 will be checked for availability and used.
f77 f90 f95 old_f77	Use a specific Fortran compiler to compile. Fortran implementation on UNIX is limited to Sun SUNWspro f95 and GNU (g77/f77) as most UNIX OS vendors do not supply Fortran compilers.
g77	Use the GNU Fortran (g77/f77) compiler to compile "progname.f". GNU is only selectively supported as we do not have it installed on all platforms, but should work because GNU is GNU.
gcc	Use the GNU C (gcc) compiler to compile "progname.c". GNU is only selectively supported as we do not have it installed on all platforms, but should work because GNU is GNU.
CC CXX cpp	Use the "C++" compiler to compile "progname.cpp" programs. C++ programs are expected to have a .cpp suffix on all platforms except on MVS OE, which requires .C as an explicit extension.
rpg	IBM i Only: Use RPG compiler to compile IFS "progname". Default extension is .rpg. Source may alternately exist as member in *CURLIB/QRPGLESRC.
pl1	z/OS Only: PL/1
cobol cob cbl	Use COBOL compiler to compile *progname*.cobol, *progname*.cob, and *progname*.cbl. Supply explicit extension to override.

Language	Compiler to use for a given language source.
java	Dummy placeholder, -m cjava is the driving factor for Java source compilation.

The -m option is described in the following chart:

Application Type	Type of Application to Build
hli	Generate an HLI program linked to the EDA HLI library that opens and modifies FOCUS data files.
odbc	Generate an ODBC API client program linked to the ODBC API driver w/o Visigenics Driver Manager (deprecated).
cpgm dll	Generate a "callpgm" server program library or sub routine (also known as a Fuselib routine). Default is a C source with .c extension unless specified by explicit (known) extension or specific -c compiler flag.
cjava	Generate a class in a jar for "calljava" server program usage. Multiple .java sources are allowed under this feature. Default jar file name is the same as the first named java source (use -j to create specific jar file names).
cl	IBM i only: Compile CL command file. Default extension of .cl; LIB/FILE(MBR) type of file specification allowed if quote enclosed to prevent sub shell interpretation by the command line parser.
cmd	IBM i Only: Compile CMD command file. Default extension of .cmd; LIB/FILE(MBR) type of file specification allowed if quote enclosed to prevent sub shell interpretation by the command line parser.
dds	IBM i Only: Compile DDS screen file. Default extension is

Application Type	Type of Application to Build
	.dds. Source may alternately exist as member in *CURLIB/QDDSSRC. DDS is an IBM i only extension for compiling screen handling files for RPG and other IBM i languages that use DDS.
-b lib/srvprg	IBM i Only: Bind in additional IBM i service programs during the link phase.
-j *jarname*	Used solely in conjunction with -m cjava to specify a specific jar to create. A .jar extension may be supplied, but extensions other than .jar are ignored and automatically switched to .jar. If the switch is not included, the first java source will be used to form the jar name (that is, myapp.java yields myapp.jar).
[{*path*}]{*program name*} [[.*extension*}]	Name of source program to build. Must be last argument. All arguments after *program name* are ignored. Extension is optional, but can serve to override a language default for a -c language specification. A path to a source is allowed (that is, source/foo.c), but non-system includes must be in current directory or the -i option must be used. On Server for z/OS in a PDS deployment, a dataset name or PDS name may be specified if -p option is in use, however, the use of parenthesis characters in the specification also requires the name to be quoted to prevent sub shell interpretation by the command line parser.

Compile and Link a Procedure

This section outlines the steps required to compile and link a sample procedure provided with the product:

Procedure

1. Copy GENCPGM from the EDAHOME bin directory to your working directory, or use the full path name to the location, and:
    - For a CALLPGM program or to build the CALLPGM sample program (CPT, SPG*.CBL, or SPG*.RPG), copy the sample program and any required include files from the etc/src3gl directory of EDAHOME to your working directory.
    - For user exits, copy the desired sample exit from the etc/src3gl directory of EDAHOME to your working directory.
    - For user routines, write the routine or copy and modify an existing routine to your working directory. (This document provides MTHNAME samples for C, COBOL, RPG, and Fortran, which you can use for reference.)
2. Issue an EDAHOME environment variable pointing to the EDAHOME directory. For example:

Windows	SET EDAHOME=C:\ibi\srv82\home
IBM i (formerly known as i5/OS)	export EDAHOME=/home/iadmin/ibi/srv82/home
UNIX	export EDAHOME=/home/iadmin/ibi/srv82/home
USS	export EDAHOME=/home/iadmin/ibi/srv82/home

3. If building an API program, also issue an EDACONF environment variable pointing to the EDACONF directory. For example:

Windows	SET EDACONF=C:\ibi\srv82\wfs
IBM i	export EDACONF=/home/iadmin/ibi/srv82/wfs
UNIX	export EDACONF=/home/iadmin/ibi/srv82/wfs
USS	export EDACONF=/home/iadmin/ibi/srv82/wfs

4. Run GENCPGM. For example, on UNIX:

```
gencpgm.sh -m cpgm mysub.c
```

GENCPGM Usage Notes

While there may not be a sample in every language for every application type, the first step is to confirm that there is a working environment by building one of the standard samples for the desired application type and confirming that it runs. If the samples do not work, there is little hope that a custom program will work.

Switches function similarly on all implementations, although, some are platform/need specific.

Programs generated for HLI will also have command file shell wrappers created with the system variables used for run time execution (that is, EDAHOME, EDACONF, EDACS3 and the library path needed for HLI to load) producing a self contained environment for runtime execution. At runtime, all application setup needs are self-contained within the wrapper so that the application simply runs. The explicit use of a GENCPGM-generated application wrapper is not required if the settings within a given wrapper are issued elsewhere (such as in a system or login profile) and the executable is directly used.

On IBM i CL and CMD, wrappers are also created in *CURLIB so applications can also be called directly on the IBM i command line. On z/OS PDS deployment, JCL, CLIST, and REXX wrappers are created as appnameJ, appnameC and appnameR (respectively), if the application is 7 characters (or less) and the -S switch has been used to indicate a save PDS name. Since the running interactive or batch and selecting a preferred language are strictly run time choices, PDS mode creates all three scripts to be prepared for all situations.

Due to the numerous third party vendors of COBOL, Fortran and other languages, inconsistency of switches between third party vendors and across platforms, GENCPGM has only limited testing of third party compilers. The actual supplied COBOL and Fortran sample programs themselves are known to work on several platforms where we do have compilers so if GENCPGM for you platform doesn't support a particular language the sole question is of figuring out how to compile and link them in order to work. Please also note that some samples (particularly COBOL) have comments of specific platform related changes that must be made for to accomplish proper compilation such as changing the PROGRAM-ID to a quoted lowercase string to achieve a properly created program entry point.

For CALLJAVA applications (-m cjava) more than one source to compile is allowed and the resulting classes are created into a single jar is supported. Java sources must have file extension of .java and specifying the actual extension on the GENCPGM command line is optional. If there are multiple source and no -j jar switch is supplied, the first source will be used to form the jar name.

The language parameter value for -c drives the default extension for a given language (except for Java), but supplying a full program name (ie mthname.cbl) will override a default.

If the compilation was for CALLPGM, a user exit, or a routine, the final step is to either copy the resulting routine to the user directory of EDACONF or set the environment variable IBICPG to the name of the actual working directory (and restart the server). This final step puts the resulting routine in a path that the server searches for routines at run time. User exits are not explicitly covered in this manual, but follow the same rules as a routine.

Language and Platform Notes

Theoretically any compiled 3GL language can be used to create an HLI, Call Procedural Program, or Subroutine programs. C is generally considered the standard language and is universally tested and implemented on all supported platforms with samples for all application types. Other languages are more selective in terms of applications for which samples exist and platforms in which they can be tested (usually due to complier availability on a given platform).

Java and JavaScript do not have options for generating Dynamic Load Libraries (DLLs), and, as such, cannot be used for creating HLI, Call Procedural Program, or Subroutine programs. However, in a language like C, it is possible to create a wrapper that loads and passes parameters to Java and receives parameters back. Thus, while Java is feasible, it is not direct and would present performance issues if done in this context and thus can not be recommended or officially supported.

Fortran: It is possible to build DLLs and programs using Fortran on any platform, however, at this time GENCPGM is only coded for Fortran on z/OS, UNIX GNU g77 and SunOS SUNWspro. Additionally, there is only a sample for Subroutine usage. The -c fortran switch on SunOS defaults to f77 usage on AMD64 and f90 on Sparc9, use the -c f77/f90/f95 switch to force other specific levels. The -c fortran switch on UNIX's will attempt to use g77, if found on the path.

COBOL: It is possible to build routines using Cobol on any platform. At this time gencpgm is coded to do Cobol only on select platforms and using select Cobol vendors, specifically

... on UNIX with MicroFOCUS Cobol (using the mf* switches), on IBM i using IBM ILE Cobol and IBM z/OS using Enterprise Cobol in -p mode. MicroFOCUS Cobol use has some additional use restrictions as described in the -c mfcobol section of the gencpgm chart.

On IBM i: Only ILE compilers are supported. Only the IBM i C compiler can directly compile files on the IFS file system. GENCPGM on IBM i does this feat for other languages by checking the default library location (for a given source type) for the existence of the desired file and if it does not exist it does a CPYFRMSTMF to duplicate the file into the library for the compilation process. If GENCPGM does a source file copy to a library, it will also remove the file afterwards so extra copies aren't floating around. In this way, sources on IBM i can exist as either IFS or library files.

On z/OS PDS Deployment: The script in hlq.HOME.ETC(GENCPGM) is an OMVS shell script and is not JCL, so it cannot be directly run from the PDS. To use under PDS deployment, copy the GENCPGM member to HFS, do a chmod +x to the script, and use as described below with z/OS switches.

Once the script is copied to an HFS directory, it is executed with the -L, -C, -A, -,S and -p options, which creates and then submits a JCL compilation stack that is language- and application-specific. If the JCL is successful, the resulting program will end up in the specified -p PDS. Regardless of build success, the JCL stack is always left behind and is saved in the current directory as the program name with a .jcl extension, as well as in the -S location if a -S switch was used. Additionally the -s switch will allow you to directly generate JCL into an HFS file or DSN, but not execute. The -s switch (lowercase s is required) is useful for sites where standard IBM libraries locations are not used for compiler and link library updates (as is commonly done at sites for add on features and updates), thus allowing a site to generate and then "adjust" the JCL for site specific needs before submission. The -S switch (uppercase S is required) does actual compilation plus saves build and run time scripts into the specified PDS for later use.

Build Rules

Should you chose to write a build script instead of using GENCPGM, the rules are fairly simple.

DLLs for Subroutine and CALLPGM Usage: Library name (less extension and any prefix such as lib) and entry point name must match. Some compilers are case sensitive on entry point name usage and some are not or uppercase entry points automatically; thus some require special coding to force lower case names as in mentioned COBOL cases. Specifically entry points must be lowercase.

Executables for HLI Usage: Must link in edahli DLL and create an executable with a main. Various environment variables must be set in order for application to run, the wrapper created by building the appropriate test sample should be used as a template as it contains any general and platform specific coding.

In both cases it is suggested that you use the standard test samples for your language of choice with the -x switch to examine the precise build switches used in any particular environment to assist in any custom built scripts.

Generating a Subroutine Program From a C Source File

The following example will generate a debuggable callpgm program library from a C source code file named myprog.c using the standard C compiler.

Optionally, the explicit API switch could have been used:

```
gencpgm -g -m cpgm myprog
```

Generating an HLI Program From a C Source File

Because the Standard C compiler and HLI mode are default options, the following example will generate a debuggable HLI program from a sample C source file named myprog.c using the standard C compiler.

Optionally, the explicit HLI switch could have been used:

```
gencpgm -g -m hli myprog
```

Generating a CALLPGM Program From a C Source File

The following example will generate a debuggable callpgm program library from a C source code file named myprog.c using the standard C compiler.

```
gencpgm -g -m cpgm myprog
```

For actual CALLPGM code samples, see Writing a 3GL Compiled Stored Procedure Program.

Migrating CMS HLI Programs to UNIX or Linux

FOCUS 7.7 does not support the CMS operating system. Customers running CMS FOCUS and upgrading their FOCUS release to 7.7 must migrate their FOCUS applications to an operating environment supported in Release 7.7. HLI programs that were compiled under CMS must be migrated to a supported operating system, as well, and recompiled in that operating environment. Typically, customers moving from CMS to another operating system move to UNIX or Linux.

Changes Needed to the CMS HLI Program

The following changes are needed when migrating a CMS HLI Program to UNIX or Linux.

- The file naming conventions are different, so the file references must be change. For example, any FILEDEF commands used in the FOCUS code will have to be changed.
    - The CMS file naming convention consists of three parts, a file name, a file type, and a file mode.

      ```
      fnft [fm]
      ```

      where:

      fn

      Is a name for the file, up to eight characters.

      ft

      Is the type of file. FOCUS uses a standard list of common file types.

      fm

      Identifies the disk that the file resides on and the type of access allowed.

The default filemode is A1.

- The UNIX/Linux file naming convention consists of an optional path to the location of the file, a file name, and a file extension.

```
[path/]fn.ext
```

where:

path

Identifies how to traverse the directory structure to the directory in which the file resides.

fn

Is a name for the file.

ext

Identifies the type of file. FOCUS uses a standard list of extensions.

Note: The entire file specification cannot be longer than 64 characters.

- File names on UNIX and Linux are case sensitive, with lowercase file names recommended, while file names in CMS are recommended to be all uppercase.
- If multiple FOCUS Database Servers (sink machines) are required, a user ID should be created on the UNIX box that controls these machines. Each sink machine should be set up in its own directory, since SU creates temporary work files using the same names, and these could override each other if all sinks were in the same directory.
- The source code for the HLI routines and any user-written subroutines must be ported to the new environment and re-compiled.
- The FOCUS databases controlled by the SU need to be ported to the new environment.
- If the FOCUS code uses any platform-specific options, such as operating system commands, these must be changed to be compatible with the new environment.

Changes Needed to the FCB When Migrating a CMS HLI Program

Because the file naming conventions are different in CMS and UNIX/Linux, the FCB layout is different.

Following are the relevant FCB words for CMS.

FCB Words	Contents	Description	Number of Bytes
1-2	FN	File Name	8
3-4	FT	File Type	8
5	FM	File Mode	4
7-8	PROCNAME	Procedure Name displayed on HLIPRINT	8
25	ERRORNUM	Detail error code (integer)	4

Following are the relevant FCB words for UNIX/Linux.

FCB Words	Contents	Description	Number of Bytes
1-2	blank	Reserved	8
3-4	blank	Reserved	8
5	blank	Reserved	4
7-8	blank	Reserved	8
25	RETURNUM	Number of instances returned (integer)	4
50-66	FILENAME	UNIX/Linux file name (up to 64 bytes with or without a full or relative path)	64

Migrating Simultaneous Usage Applications From CMS to UNIX or Linux

Using the Simultaneous Usage (SU) facility, multiple users can read and change a FOCUS or XFOCUS database at the same time, using FOCUS and Host Language Interface (HLI) commands.

SU Under CMS

With SU, a centrally controlled database is allocated to a background job called the FOCUS Database Server or sink machine. TSO IDs, MSO sessions, and batch jobs running FOCUS, as well as programs using HLI, that send database retrieval and update requests to the FOCUS Database Server are all called source machines or clients. Source machines send requests and transactions to the FOCUS Database Server, which processes their transactions and transmits retrieved data and messages back to the source machines.

Without SU, only one user can update a database at a time, even databases that are allocated for sharing (DISP=SHR).

Source machines communicate with the FOCUS Database Server through cross-memory posting. Messages traveling between a source machine and the FOCUS Database Server are placed in the z/OS Common Storage Area (CSA), which is accessible to both machines. When the FOCUS Database Server receives a request from a source machine, it changes or retrieves data from the centrally controlled database and transmits the results back to the source machine.

CMS is case sensitive. Upper case is highly recommended for all filenames.

LISTFILE lists the names of files on any disk or SFS directory. COPY[FILE] copies a file to another file.

```
COPYFILE fn1ft1fm1fn2ft2fm (options
```

The following example copies HISTORY SCRIPT H to HISTORY SCRIPT A.

```
copy history script h history script a
```

SU Under UNIX or Linux

On UNIX or Linux, any file or directory can have multiple names because the operating system uses inodes instead of names to identify files and directories. Additional names can be provided by using the ln command to create one or more hard links to a file or directory.

UNIX is case sensitive. Lower case is highly recommended for all filenames. Some common UNIX commands follow.

`ls directory`. List contents.

`cd directory`. Change to another directory.

The following command copies the file myfile1 from the current directory and places a copy in

/priv/home/henry/myfile2.

```
cp myfile1 /priv/home/henry/myfile2
```

If multiple sink machines are required, a user ID to control them should be created on the UNIX box to control them. Each sink machine should be set up in its own directory since each one creates temporary work files using the same name, and these could overwrite each other if all sinks were in the same directory.

COBOL subroutines need to be ported to the UNIX/Linux environment with source code and re-compiled.

The actual FOCUS databases controlled by SU must be migrated to the UNIX/Linux environment. in Release 7.6 or above, you can use Structured HOLD Files to extract the FOCUS data with its structural information on CMS and recreate it on UNIX or LInux. In releases prior to 7.6, you will have to HOLD each path separately and reassemble the FOCUS structure on UNIX or Linux.

FOCUS code may or may not need to change, depending on whether platform-specific options are used. Almost all platforms support the same core FOCUS code.

ibi Documentation and Support Services

For information about this product, you can read the documentation, contact Support, and join Community.

How to Access ibi Documentation

Documentation for ibi products is available on the Product Documentation website, mainly in HTML and PDF formats.

The Product Documentation website is updated frequently and is more current than any other documentation included with the product.

Product-Specific Documentation

The documentation for this product is available on the ibi™ FOCUS® Documentation page.

How to Contact Support for ibi Products

You can contact the Support team in the following ways:

- To access the Support Knowledge Base and getting personalized content about products you are interested in, visit our product Support website.
- To create a Support case, you must have a valid maintenance or support contract with a Cloud Software Group entity. You also need a username and password to log in to the product Support website. If you do not have a username, you can request one by clicking **Register** on the website.

How to Join ibi Community

ibi Community is the official channel for ibi customers, partners, and employee subject matter experts to share and access their collective experience. ibi Community offers access to Q&A forums, product wikis, and best practices. It also offers access to extensions, adapters, solution accelerators, and tools that extend and enable customers to gain full value from ibi products. For a free registration, go to ibi Community.

Legal and Third-Party Notices

SOME CLOUD SOFTWARE GROUP, INC. (“CLOUD SG”) SOFTWARE AND CLOUD SERVICES EMBED, BUNDLE, OR OTHERWISE INCLUDE OTHER SOFTWARE, INCLUDING OTHER CLOUD SG SOFTWARE (COLLECTIVELY, “INCLUDED SOFTWARE”). USE OF INCLUDED SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED CLOUD SG SOFTWARE AND/OR CLOUD SERVICES. THE INCLUDED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER CLOUD SG SOFTWARE AND/OR CLOUD SERVICES OR FOR ANY OTHER PURPOSE.

USE OF CLOUD SG SOFTWARE AND CLOUD SERVICES IS SUBJECT TO THE TERMS AND CONDITIONS OF AN AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER AGREEMENT WHICH IS DISPLAYED WHEN ACCESSING, DOWNLOADING, OR INSTALLING THE SOFTWARE OR CLOUD SERVICES (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH LICENSE AGREEMENT OR CLICKWRAP END USER AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE SAME TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of Cloud Software Group, Inc.

ibi, the ibi logo, FOCUS, iWay, WebFOCUS, RStat, Information Builders, Studio, and TIBCO are either registered trademarks or trademarks of Cloud Software Group, Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only. You acknowledge that all rights to these third party marks are the exclusive property of their respective owners. Please refer to Cloud SG’s Third Party Trademark Notices (https://www.cloud.com/legal) for more information.

This document includes fonts that are licensed under the SIL Open Font License, Version 1.1, which is available at: https://scripts.sil.org/OFL

Copyright (c) Paul D. Hunt, with Reserved Font Name Source Sans Pro and Source Code Pro.

Cloud SG software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the “readme” file for the availability of a specific version of Cloud SG software on a specific operating system platform.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. CLOUD SG MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S), THE PROGRAM(S), AND/OR THE SERVICES DESCRIBED IN THIS DOCUMENT AT ANY TIME WITHOUT NOTICE.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "README" FILES.

This and other products of Cloud SG may be covered by registered patents. For details, please refer to the Virtual Patent Marking document located at https://www.cloud.com/legal.

Copyright © 2021-2025. Cloud Software Group, Inc. All Rights Reserved.