Airline Control System
Version 2.4.1

Application Programming Guide

SH19-6948-21

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xv.

This edition applies to Release 4, Modification Level 1, of Airline Control System Version 2, Program Number 5695-068, and to all subsequent releases and modifications until otherwise indicated in new editions. Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below. A form for readers’ comments appears at the back of this publication. If the form has been removed, address your comments to:

ALCS Development
2455 South Road
P923
Poughkeepsie NY 12601-5400
USA

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

© Copyright International Business Machines Corporation 2003, 2019.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures ..... **ix**

Tables ..... **xiv**

Notices ..... **xv**
Programming interface information xv
Trademarks xvi

About this book ..... **xvii**
Who should read this book xvii
How this book is organized xvii
ALCS services xviii
TPF compatibility xviii
Note on the example code xix

Chapter 1. Overview of ALCS **1**
What ALCS provides 1
Agent sets 1
Functions performed by ALCS 2
Processing a message 2
Using information from previous messages 6
Application programs in ALCS 6
Direct access files 7
Fixed-file records 8
Pool-file records 9
Chains of pool-file records 9
Recovering long-term pool-file records 10
Application global area 11
General files and general data sets 11
Virtual file access 12
Sequential files 12
Communication with users, other systems, and other applications 12
ALCS applications 12

Chapter 2. Communication **14**
Communication facilities 14
Receiving input messages 15
Validating input data 16
Sending response messages to the originating terminal 16
Sending unsolicited messages 17
Message formats 17
APPC communication calls and conversation services 17
ZSTAT message counts and data collection statistics 17
Program preparation 18
Including CPI-C calls in application programs 18
Including APPC/MVS calls in application programs 19
APPC/MVS calls with asynchronous notification of completion 21
Ending the conversation 22
Message Queue Interface (MQI) 23

ZSTAT message counts and data collection statistics....24
Program preparation....24
Including MQI calls in application programs....24
MQI calls with asynchronous notification of completion....25
Closing MQI objects....28
Optimized Local Adapters (OLA) for WebSphere Application Server....28
WAS Bridge....28
ZSTAT message counts and data collection statistics....29
Program preparation....29
Including OLA calls in application programs....29
Testing the return code....30
Unregistering and releasing connection handles....30
3270 Screen mapping support....30
TCP/IP support....31
HTTP Client support....33
JSON Parser support....34

Chapter 3. Data in ALCS....36
Using direct access files....36
File addresses....36
Standard record header....36
Reading and writing DASD records....38
Reading DASD records....39
Writing DASD records....40
Sequential files....41
Serialization....41
Serialization - forcing exclusive access to resources....43
Serializing access to the application global area....43
SYNCC monitor-request macro and `global` C function....49
ALCS services for accessing DASD records....49
Summary of ALCS services for DASD processing....51
Offline access to general files and general data sets....52
Using sequential files....52
System sequential files....53
Real-time sequential files....53
General sequential files....53
Several entries using the same general sequential file....53
ALCS services for sequential file processing....55
Summary of ALCS services for sequential files....55
Global records and fields....56
Global directories....56
Keypointing global records....57
Logical global records....57
Using globals - attention....58
Global area serialization....59
C functions for processing globals....60
How a C language application program accesses the global area....60
ALCS services for global area handling....61
Global area protection....62
Global area protect key....62
Using global area protection....62
Using Structured Query Language (SQL)....64
Preparation....64
Making SQL calls....64
Distributing application logic across several programs....65
Designing databases for use with ALCS....65
Performance and design....65

TPF Database Facility (TPFDF)....66
Portability....66
Organizing data....67
Standard structures....67
TPF Database Facility (TPFDF)....67
Record classes....67
Fixed files....67
Pool files....67
Short-term pool file....68
Long-term pool file....68
Choosing which class of record to use....69
Protecting data....69
Commit and backout....70
Single-phase commit....70
Two-phase commit....70
ALCS commit and backout services....70
Keeping a transaction log....71
Sequential update technique....71
Why ALCS does not provide a synchpoint manager....73
Other considerations....74
Utility programs....74

Chapter 4. The entry control block (ECB) and other entry storage....75
Format of the ECB....75
ECB work areas 1 and 2....76
Work area fields....76
ECB local program work area....77
ECB data levels and storage levels....78
ECB data levels....79
ECB storage levels....79
Entry origin fields....80
User register save areas....80
General registers....80
Floating-point registers....81
ECB error indicators....82
ECB user area reserved for system-wide fields....82
Exit intercept information....82
TCP/IP socket descriptor, TCP/IP port number, and Listener index number....82
The routing control parameter list (RCPL) area....82
ECB areas reserved for ALCS use....83
Accessing the ECB....83
Other entry storage....83
ECB user data collection area....84
Heap storage used by assembler programs....84
Automatically-allocated storage....84
Initial storage allocation....84
Stack and heap storage....85
Storage blocks....85
Input and output messages....86
Entry storage on exit....86
Swapping storage levels....86
Detaching and attaching storage blocks....86
Storage block activity control....87
ALCS services for handling storage blocks....87
Local save stack....87
Data event control blocks (DECBs)....88
DECB fields....88

Managing DECBs..... 90
Using symbolic names.....90
ALCS services for managing DECBs..... 90
8-byte file address support.....91
Accessing DASD records and using storage blocks with a DECB.....91
Error handling..... 92
Accessing a DECB..... 93
Functional flow..... 93

Chapter 5. Application programming considerations.....97
Reentrant requirements of application programs..... 97
Links between programs..... 97
Creating new entries.....101
Events.....103
Condition code and program mask..... 106
Floating-point registers..... 106
Calling other functions and programs.....106
Calling a C program (from assembler).....106
Calling an assembler program (from C).....106

Chapter 6. Error conditions and error processing..... 110
Error conditions..... 110
The ALCS wait mechanism and error processing..... 111
Dealing with errors..... 115
Error indicators in the ECB and DECB.....115
Testing for errors..... 117
Multiple errors.....118
Sequential file errors..... 119
Testing for several conditions..... 119
Recovery from wait-completion errors.....121
Application program logic errors.....121
Error recovery..... 122

Chapter 7. Application program management.....124
Application programming languages..... 124
Assembler..... 126
SabreTalk..... 127
C language..... 128
Other high-level languages..... 129
Choosing languages for application programming..... 130
Providing callable services for high-level language programs..... 132
Developing callable services.....133
Serialization considerations for callable services..... 134
Return codes and reason codes..... 136
Parameters..... 137
Using callable services with existing application programs.....138
Example of a callable service for an existing application..... 138
Example callable service written in C language.....140
Calling other functions and programs..... 142

Chapter 8. Assembling, compiling, and link-editing application programs..... 145
Application source modules.....145
Program size.....145
Converting assembler DSECTs to C language structures.....146
Library control, naming conventions, and versions.....146
Application load modules.....148
Load module names.....149

How ALCS uses application load modules ... 149
Choosing what programs to include in a load module ... 150
Special considerations for high-level language programs ... 153
Special considerations for SQL programs ... 155
Static and dynamic program linkage ... 156
Program names, transfer vectors, and high-level language entry points ... 158
Overview of program preparation ... 160
How to assemble or compile application programs ... 161
SabreTalk compile ... 161
DB2 for z/OS precompile ... 161
Assemble ... 162
Compile ... 163
How to build program headers and prelink high-level language programs ... 164
Program header build - entry points definition file ... 166
Entry points definition file - syntax ... 166
Program header build - DXCBCLPP program ... 168
Building the HLL load module ... 168
How to link-edit application load modules ... 169
How to build application plans (DB2 BIND) ... 169

Chapter 9. Program testing and problem determination ... 170
Test database facility ... 170
How the test database facility works ... 170
Benefits ... 171
Conversational trace ... 171
Workstation trace and remote debugger ... 172

Appendix A. Sample application: assembler ... 174
Purpose of the application ... 174
Application entry conditions ... 174
Application return conditions ... 174
Application normal responses ... 174
Application error responses ... 175
Installing the application ... 176
Running the application ... 176
Source file (APG1 ASSEMBLE) ... 176

Appendix B. Sample application: C language ... 183
Purpose of the application ... 183
How the program source files are related ... 186
DXCBINQ ... 187
DXCBINQH ... 187
DXCBINQ0 ... 189
DXCBINQ1 ... 190
DXCBINQ2 ... 191
DXCBINQ3 ... 193
DXCBINQD ... 194
DXCBINQM ... 195
DXCBINQP ... 197
Running the application ... 198
Debugging statements ... 199

Appendix C. Sample application: COBOL ... 200
Purpose of the application ... 200
Running the application ... 200
Sample output ... 200
Source files ... 201

Appendix D. Messages and message formats..... 215
Input and output messages..... 215
Standard format for input message text..... 215
Standard format for output message text..... 217
Message format - detail..... 220
RCPL contents..... 223

Appendix E. 3270 Screen mapping support for application programs..... 226
Acronyms and abbreviations..... 228
Glossary..... 240
Bibliography..... 270

Index..... 276

Figures

1. ALCS operating environment....1
2. Agent presses the ENTER key to send a message....3
3. ALCS creates an ECB for the message....3
4. ALCS selects a program for the ECB....4
5. Program requests data from a DASD record....4
6. Program receives data from a DASD record....5
7. Program transfers control to another program....5
8. ALCS sends a message to the terminal....6
9. Working storage....7
10. Fixed file ordinals....8
11. Fixed-file record containing a file address of a single pool-file record....9
12. Fixed-file record addressing a chain of pool-file records....10
13. Example ALCS applications and their constituent programs....13
14. Example CPI-C call -- assembler....18
15. Example CPI-C call -- C language....18
16. Example CPI-C call -- COBOL....19
17. A GET_CONVERSATION call -- assembler....19
18. APPC/MVS function call -- C language....20
19. APPC/MVS function call -- COBOL....21
20. An ATAWAIT call -- assembler....23
21. Using MQI for integrated transaction processing....24
22. An MQI MQOPEN call -- assembler....25
23. An MQI MQOPEN call -- C language....25

24. An MQI MQOPEN call -- COBOL....25
25. Example of an MQAWAIT call -- assembler (Part 1 of 2)....27
26. Example of an MQAWAIT call -- assembler (Part 2 of 2)....28
27. A WAS BBOA1REG call -- assembler....29
28. A WAS BBOA1REG call -- C language....30
29. Using TCP/IP to communicate between applications....31
30. Including TCP/IP Sockets calls in an assembler program....32
31. An HTTP Client HWTHINIT call – assembler....34
32. An HTTP Client HWTHINIT call – C/C++....34
33. A JSON Parser HWTJINIT call – assembler....35
34. A JSON Parser HWTJINIT call – C/C++....35
35. Standard record header....37
36. Contents of an ECB data level before a DASD read request....39
37. ECB data level and storage level after a successful DASD read....40
38. ECB data level and storage level before a DASD write request....40
39. Reading DASD records without holding the file address - updates performed by either entry can be lost....41
40. Reading DASD records and holding the file address - updates performed by either entry are recorded....42
41. ALCS services for processing DASD records....52
42. Two entries accessing the same general sequential file....54
43. Macros and functions for processing sequential files....55
44. ALCS global area directory - logical view....56
45. Removing headers to create a logical global record....58
46. Updating fields in protected global areas....64
47. An SQL UPDATE statement -- assembler....65
48. An SQL UPDATE statement -- C language....65

49. Adding a new structure....72
50. Updating a new structure....73
51. Format of the ALCS ECB (schematic)....76
52. Using the general register save area....81
53. Using the floating-point register save area....81
54. Detaching and attaching a storage block - assembler....87
55. Detaching and attaching a storage block - C language....87
56. Format of the ALCS DECB application area (schematic)....88
57. Allocating a DECB - assembler....93
58. Allocating a DECB - C++ language....93
59. Computing an 8-byte file address - assembler....94
60. Computing an 8-byte file address - C++ language....94
61. Requesting a FIND-type service using a DECB - assembler....94
62. Requesting a FIND-type service using a DECB - C++ language....94
63. Creating a new DECB - assembler....94
64. Creating a new DECB - C++ language....94
65. Locating a DECB - assembler....95
66. Locating a DECB - C++ language....95
67. Requesting a FILE-type service using a DECB - assembler....95
68. Requesting a FILE-type service using a DECB - C++ language....95
69. Releasing a DECB - assembler....95
70. Releasing a DECB - C++ language....95
71. Requesting a FILE-type service using a DECB - assembler....96
72. Requesting a FILE-type service using a DECB - C++ language....96
73. Program entering another program....98

74. The ENTER/BACK mechanism in assembler programs..... 99

75. The ENTER/BACK mechanism in C programs..... 99

76. An application program creates a new entry..... 101

77. How parent and child entries use a counter-type event.....104

78. How events use ECB levels.....105

79. Read using an implied-wait service.....112

80. Wait after a required-wait read service.....112

81. Multiple reads followed by an implied-wait read.....113

82. Summary of ALCS I/O services.....114

83. Error indicators in the ECB.....116

84. Read two DASD records and test for errors - assembler.....117

85. Read two DASD records and test for errors - C language.....118

86. Multiple implied-wait services..... 118

87. Processing multiple I/O errors - C language.....119

88. Test for sequential file errors - C language..... 119

89. Test for several sequential file errors - assembler program.....120

90. Test for several sequential file errors - C language.....120

91. Summary of ALCS application program interfaces..... 126

92. Callable services..... 133

93. Using a callable service with an existing application program.....138

94. Example callable service GETPNT - assembler..... 139

95. GPNT interface program - assembler.....139

96. Using the callable service GETPNT from a C language program.....140

97. Using the callable service from a COBOL program.....140

98. Callable service (QFLTTYPE) written in C language.....141

99. Using the callable service QFLTTYPE from a C language program....141

100. Using the callable service QFLTTYPE from a COBOL program....142

101. Relationship between source, object, and load modules....148

102. Load modules - static and dynamic program linkage....151

103. Relationship between base and update application load modules....152

104. Program structure - assembler and SabreTalk....153

105. Program structure - single high-level language program....154

106. Program structure - multiple high-level language programs....155

107. ALCS dynamic program linkage - schematic....156

108. ALCS static program linkage - schematic....157

109. ALCS application program headers....159

110. Preparing the ALCS application program header for a high-level language application program - schematic....160

111. Preparing the program header and prelinking a high-level language program....165

112. Example entry points definition file....168

113. The test database facility....170

114. Relationship between files in the sample application....186

115. Modules in the high-level language sample application....201

Tables

1. Block sizes 7

2. TPF global areas..... 63

3. ALCS global areas with global area protection..... 63

4. Names of work area fields.....76

5. Names of local program work area fields..... 78

6. Labels for the general register save area.....80

7. Labels for the floating-point register save area..... 81

8. How the 3 types of event are used in ALCS..... 104

9. Event services..... 105

10. Reporting of I/O errors with ALCS service.....114

11. Symbols for error conditions..... 116

Notices

References in this publication to IBM® products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

```
The Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
USA
```

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

```
IBM Corporation
Department 830A
522 South Road
Mail Drop P131
Poughkeepsie, NY 12601-5400
USA
```

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Programming interface information

This Application Programming Guide is intended to help programmers write applications programs to run under Airline Control System Version 2, Program Number 5695-068. This Application Programming Guide primarily documents General-Use Programming Interface and Associated Guidance Information provided by Airline Control System Version 2 Program Number 5695-068.

General-Use programming interfaces allow the customer to write programs that obtain the services of Airline Control System Version 2.

However, this Application Programming Guide also documents Product-Sensitive Programming Interface and Associated Guidance Information.

Product-Sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of this IBM software product. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-Sensitive programming interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-Sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Product-sensitive Programming Interface

© Copyright IBM Corp. 2003, 2019

Product-Sensitive Programming Interface and Associated Guidance Information...

End of Product-sensitive Programming Interface

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

About this book

This book describes Release 4.1 of Airline Control System (ALCS) Version 2. It is intended for application programmers writing application programs in assembler or a high-level language (HLL).

ALCS is one of a family of IBM programs designed to satisfy the needs of airlines and other industries with similar requirements for high-volume and high-availability transaction processing.

The product, which is also known as TPF/MVS, provides the Transaction Processing Facility (TPF) application programming interface (API) for z/OS® environments. It supersedes ALCS/Multiple Virtual Storage/Extended Architecture (ALCS/MVS/XA), known as ALCS Version 1.

Throughout this book:

- Airline Control System Version 2 is abbreviated to ALCS unless the context makes it necessary to distinguish between ALCS Version 2 Release 4.1, and the predecessor products.
- Airlines Line Control Interconnection (ALCI) includes the function of network extension facility (NEF).
- Advanced Communications Function for the Virtual Telecommunication Method is abbreviated to VTAM®.
- TPF refers to all versions of Transaction Processing Facility and its predecessor, Airlines Control Program (ACP).
- MVS™ refers to z/OS. MVS, S/390® and "OS/390®" also refer to operating systems.

Who should read this book

This book is written for ALCS application programmers who are writing in IBM assembler language or in a high-level language. Additional information on ALCS is available in the publications listed in the “Bibliography” on page 270.

Although the book is primarily intended for application programmers, some sections will also be of interest to system programmers.

How this book is organized

This book is organized as follows:

Chapter 1, “Overview of ALCS,” on page 1
This chapter is intended for programmers who are new to ALCS. It gives a brief description of each of the ALCS features and explains how ALCS manages transaction processing.

Chapter 2, “Communication,” on page 14
This chapter describes the communication facilities available with ALCS.

Chapter 3, “Data in ALCS,” on page 36
This chapter describes how ALCS holds data on its real-time database and in sequential files, and how application programs access this data. It also describes how an application can access data on other systems, for example data owned by a relational database manager.

Chapter 4, “The entry control block (ECB) and other entry storage,” on page 75
This chapter describes the entry control block (ECB) and other entry storage.

Chapter 5, “Application programming considerations,” on page 97
This chapter describes things that are specific to application programs that run under ALCS.

Chapter 6, “Error conditions and error processing,” on page 110
This chapter describes errors that application programs and ALCS can detect, together with suggested actions by the program.

© Copyright IBM Corp. 2003, 2019

Chapter 7, “Application program management,” on page 124
This chapter describes how you manage application programs written in assembler and high-level programming languages.

Chapter 8, “Assembling, compiling, and link-editing application programs,” on page 145
This chapter describes how you assemble or compile application programs, and how you link-edit and load them.

Chapter 9, “Program testing and problem determination,” on page 170
This chapter describes how you test and debug application programs that run under ALCS.

Appendix A, “Sample application: assembler,” on page 174
This appendix lists and describes a complete sample application written in assembler using the facilities described in this book.

Appendix B, “Sample application: C language,” on page 183
This appendix lists and describes a complete sample application written in C using the facilities described in this book.

Appendix C, “Sample application: COBOL,” on page 200
This appendix lists and describes a complete sample application written in COBOL using callable services written in C and assembler.

Appendix D, “Messages and message formats,” on page 215
This appendix shows the different message formats used with ALCS, together with the routing control parameter list (RCPL).

Appendix E, “3270 Screen mapping support for application programs,” on page 226
This appendix describes how you can use the ALCS screen mapping support provided for assembler programs.

“Acronyms and abbreviations” on page 228
This appendix contains a list of acronyms and abbreviations.

The book also includes a bibliography, a glossary of terms, and an index.

ALCS services

ALCS provides **monitor services**, which application programs use to request ALCS to perform various actions, for example, to read a record from or write a record to the real-time database.

In an assembler program, you request an ALCS service by means of a macroinstruction, or by calling another program (a **callable service**). For example, to write a record to DASD, an assembler program could call the `FILEC` macro. In a C language program you call a corresponding C function, in this case `filec`. In this book, for brevity, ALCS monitor services are described in sentences like the following: "Application programs request the `FILEC` (`filec`) service to write a DASD record."

Some assembler macros do not have corresponding C functions, and some C functions do not have corresponding assembler macros.

This book does not describe all the services that ALCS provides. For a full description of assembler and C language services, see *ALCS Application Programming Reference - Assembler* and *ALCS Application Programming Reference - C Language* respectively.

TPF compatibility

In this book some material is only relevant if you are writing programs that might need to work in a TPF environment, or if you are porting programs that were originally written for TPF. Such information is identified as in the following example:

TPF compatibility:

In TPF, the maximum size of a load module is 4KB.

If your programs do not need to be compatible with TPF, you can ignore these boxes.

Note on the example code

Assembler, C language, and other high-level language code examples are intended to appear in as readable a form as possible. They do not necessarily appear exactly as they would be coded. For example, in assembler programs:

- Continuation characters ('X') might not always appear in column 72.
- Some SPACE 1 lines have been replaced by blank lines.

In both assembler and high-level language program examples, the symbol ⋮ represents any number of lines of code which are not significant in the example.

Chapter 1. Overview of ALCS

This chapter presents an overview of the main features of ALCS.

What ALCS provides

ALCS is a software interface between customer-written application programs and the MVS operating system. It provides a number of services that application programs can use. To request these services in assembler programs you use the ALCS assembler macros described in *ALCS Application Programming Reference - Assembler*. In C programs you use the C functions described in *ALCS Application Programming Reference - C Language*.

ALCS runs as a job or started task under MVS. It provides real-time transaction processing for airlines and other industries needing high-volume transaction rates and high system availability. It is typically used for passenger reservation and ticketing systems.

A typical ALCS operating environment is shown in outline in Figure 1 on page 1.

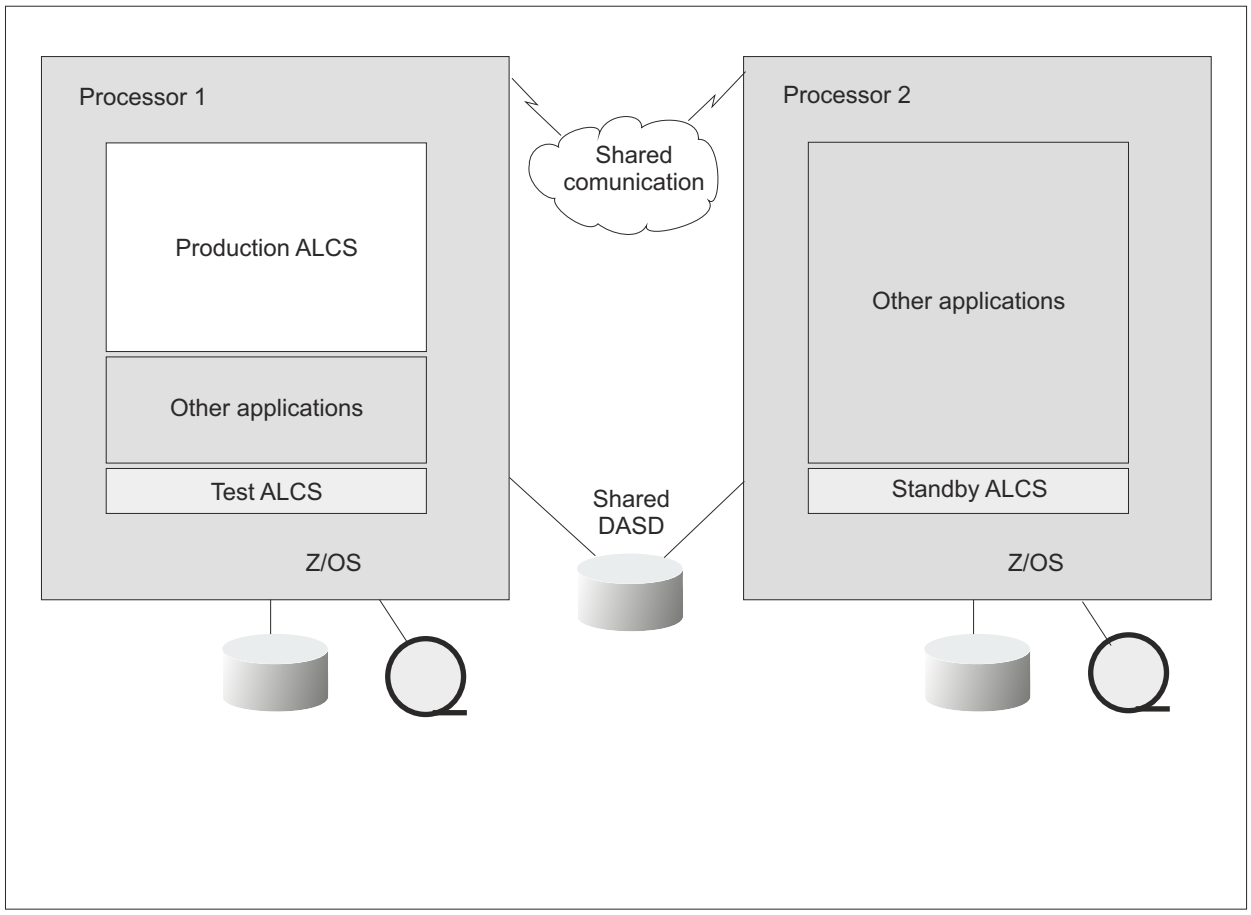


Figure 1. ALCS operating environment

Agent sets

There can be several thousand terminals linked to the same ALCS system. These terminals are called **agent sets**. If your installation uses Tivoli® NetView®, you can also use NetView operator identifiers (operator IDs) as agent sets.

© Copyright IBM Corp. 2003, 2019

An ALCS agent set intended for operators, programmers, and data administrators is called a **computer room agent set** (CRAS).

Some CRAS terminals have specific uses:

Prime CRAS
This is the primary terminal that the operator uses to control ALCS. The Prime CRAS accepts all operator commands.

Receive Only CRAS (RO CRAS)
This is a printer designated to receive messages sent by ALCS.

Alternate CRAS terminals
These terminals accept some, but not all of the operator commands. ALCS is usually set up so that if the current Prime CRAS fails, it automatically switches to an alternate CRAS terminal, which then becomes the Prime CRAS.

Alternate CRAS printers
There can be a number of alternate CRAS printers. ALCS is usually set up to switch to one of these if the RO CRAS fails.

Functions performed by ALCS

ALCS (working with other system software, such as MVS and VTAM) performs the following functions:

Controls incoming and outgoing messages
ALCS controls the transmission of all messages between end-user terminals and application programs.

Manages processor storage and file storage
ALCS controls the allocation and release of processor storage and DASD storage.

Queues work to be done on messages
ALCS maintains lists of work to be done on all messages currently being processed.

Controls input and output to DASD
ALCS controls all input and output operations to DASD, usually on request of the application programs.

Checks errors and recovers where possible
ALCS identifies, logs, and resolves (where possible) all permanent and transient errors.

Communicates with the operator
ALCS allows the operator to issue commands to request information, or to change operating parameters. It also provides information to the operator when this becomes necessary.

Allows switch-over to standby system
ALCS is designed so that, if necessary, the operator can switch over to a standby version of ALCS with very little disruption.

Provides online maintenance and customization
Many installations require the ALCS system to be available 24 hours a day, 7 days a week. All ALCS maintenance facilities can be run at the same time as other work, without disrupting applications.

Processing a message

The following series of figures shows how ALCS uses an application program to process a simple enquiry, for example, an agent asking if there are any vacancies on a flight, or a bank officer querying a customer balance in a bank account.

Most units of work are initiated by **messages** received from terminals (Figure 2 on page 3). These units of work are called **entries**. (On many terminals the agent presses the ENTER key to send the message.)

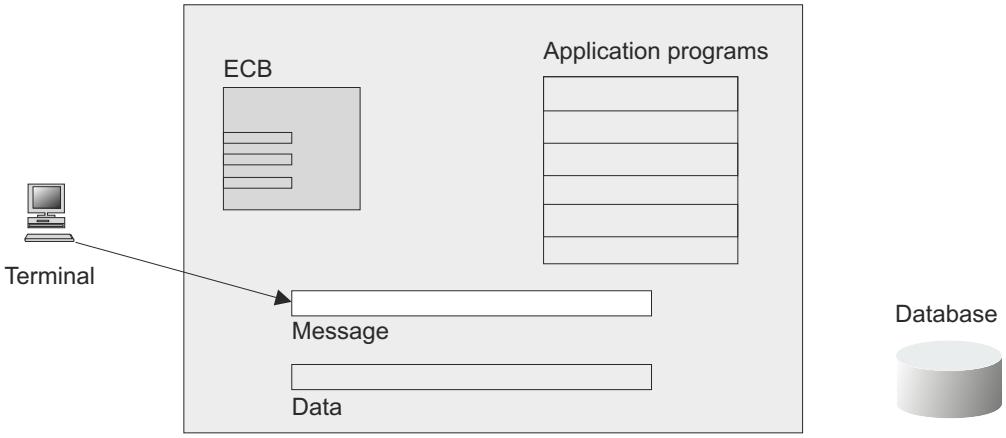


Figure 2. Agent presses the ENTER key to send a message

Entry control block

When ALCS receives a message, it assigns a storage area called an **entry control block** (ECB) to that message (Figure 3 on page 3). The ECB contains a pointer to the message.

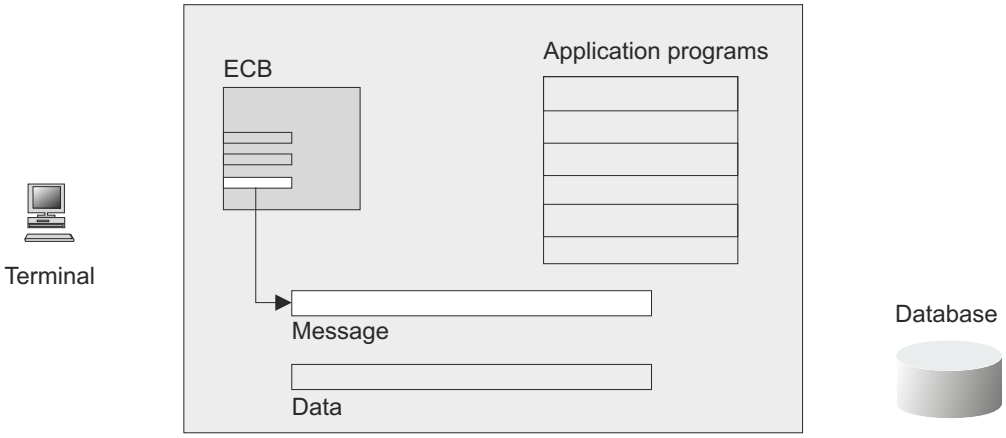


Figure 3. ALCS creates an ECB for the message

The ECB is an interface between ALCS and application programs. ALCS uses part of the ECB for saving information. Other parts are available for use by application programs as small work areas. Another part is used for holding the addresses of areas of storage used by the program.

Selecting a program

ALCS examines the originating terminal address of the message, and message contents, and selects the appropriate application program to process the message (Figure 4 on page 4).

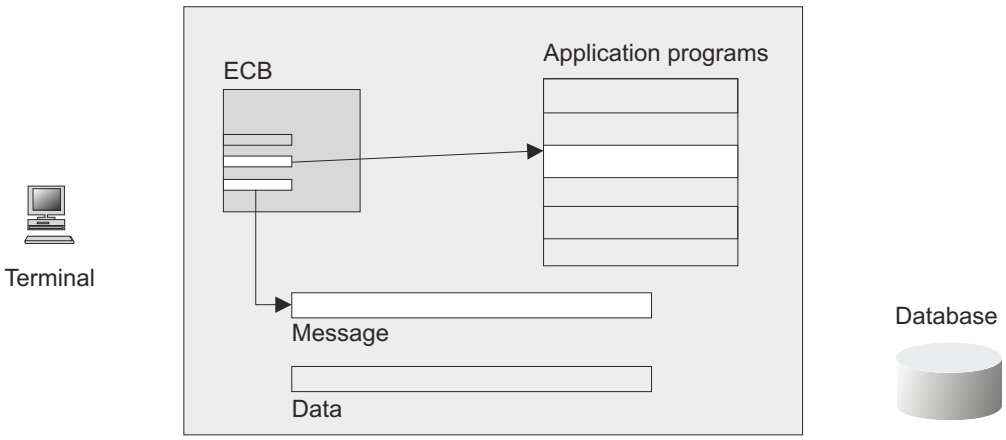


Figure 4. ALCS selects a program for the ECB

Program flow

The program starts processing the message. At some stage it needs to read data from DASD. It does not read DASD directly but requests an ALCS service to perform the read, for example, `FINDC` (`findc`). When it has done this, the application program usually suspends itself, for example by requesting an ALCS wait service, and control returns to ALCS (Figure 5 on page 4).

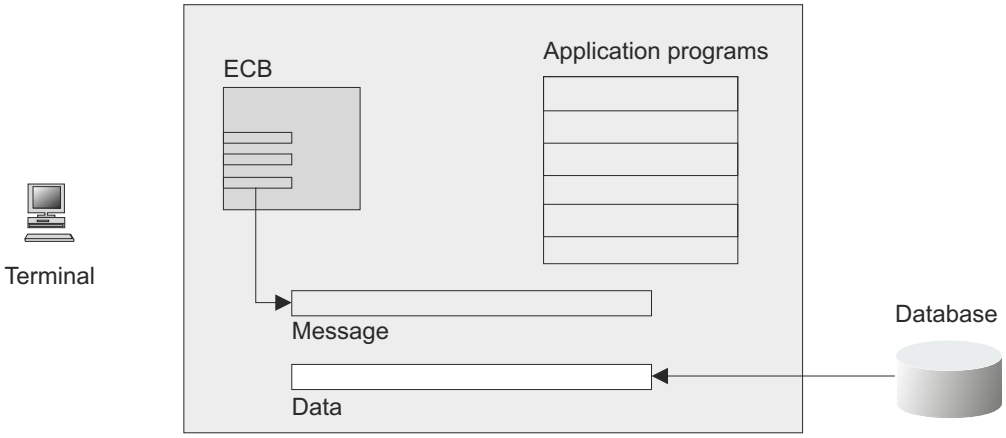


Figure 5. Program requests data from a DASD record

When this happens, ALCS stores in the ECB the contents of the current program registers and information such as the address of the next program instruction.

Data is held on DASD in the form of **records**. Records are explained in "Direct access files" on page 7.

After ALCS has initiated a read of the DASD record, it checks for other entries to start or continue processing.

ALCS reactivates the program

When ALCS detects that the read of the DASD record has completed, it restarts the program from the point stored in the ECB. The program can then continue as if there had been no interruption. ALCS puts a pointer in the ECB to the data it has read (Figure 6 on page 5).

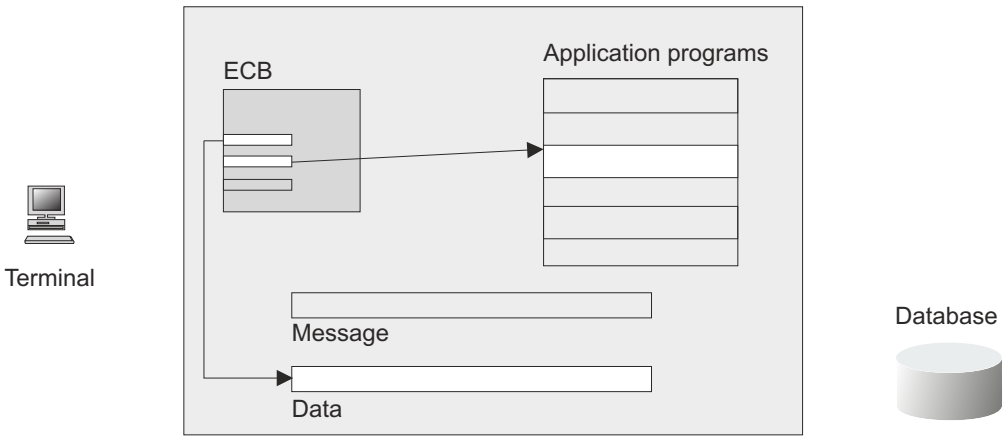


Figure 6. Program receives data from a DASD record

An ECB can have up to 16 pointers to data read from DASD. The 16 fields containing these pointers are called **data levels**.

Transferring control to another program

An application program need not carry out all the processing itself, but can transfer control to another program to perform a specific action or task (Figure 7 on page 5). The called program can return back to the first program or can in turn call other programs.

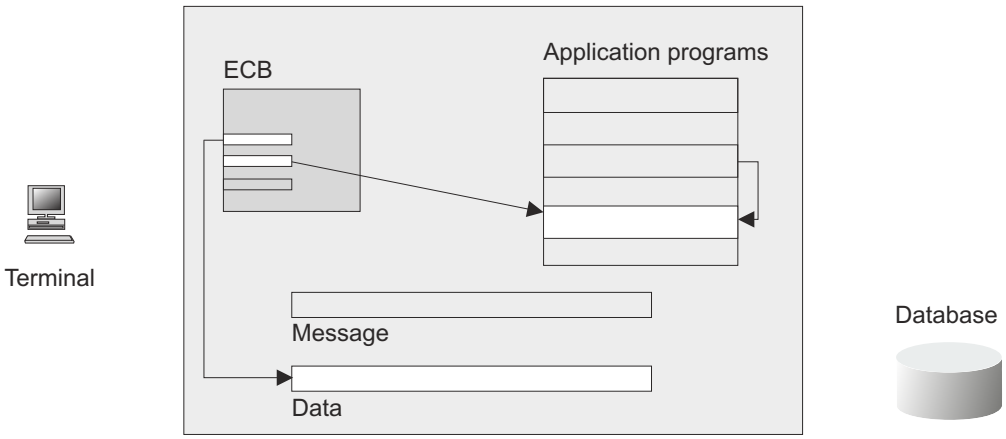


Figure 7. Program transfers control to another program

Application programs transfer control by requesting ALCS transfer services (see "Transferring control between programs" on page 97). In addition, high-level language programs can call functions defined in other application programs without losing control.

Sending a message

The application program, after processing data read from DASD, creates an output message. ALCS displays this message on the terminal where the input originated (Figure 8 on page 6).

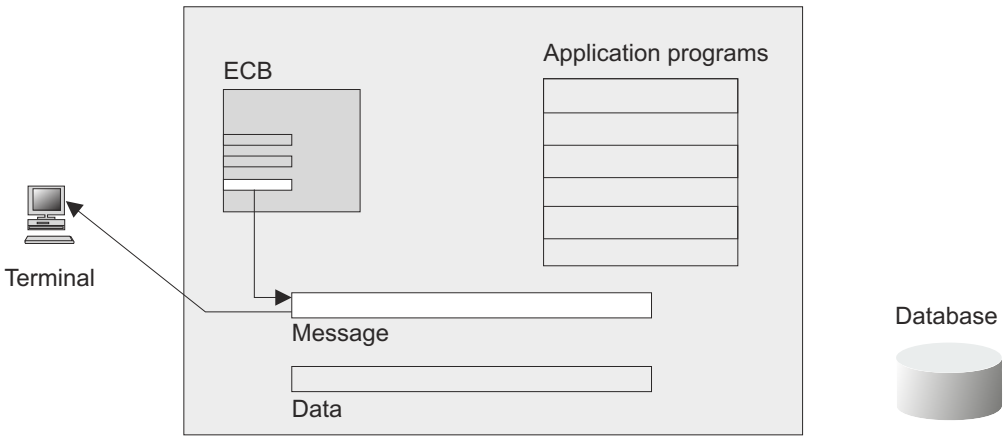


Figure 8. ALCS sends a message to the terminal

After it passes the message to ALCS, the application program normally **exits** the entry. ALCS releases the ECB, and any associated storage used by the entry, as part of the exit processing.

Using information from previous messages

One ECB is associated with each message. Some transactions might require the application program to use information from several messages received from the same terminal. (Messages contain the originating terminal address.)

In these cases, the application must retain information from a series of messages received from a single terminal. It can do this by storing information in DASD records.

The IPARS reservation application (see "IPARS applications" on page 13) holds data relating to a particular terminal in a record called the **agent assembly area** (AAA) record. The IPARS application reads this record, updates the record and then writes it back to DASD (by requesting an ALCS write service, for example `FILEC` (`filec`)).

Application programs in ALCS

ALCS might have to restart the same application program many times, from different points in the program. Application programs must be written in such a way that ALCS can do this, and can also use the same program concurrently for many entries.

Such programs must be **reentrant**. A reentrant program must not modify any storage located within the program itself. Instead, it must reference switches, indicators, or counters in an application work area in the ECB, or other storage owned exclusively by the entry. In this case, ALCS can suspend, abend, initiate, or resume processing of any number of entries at any point in their processing.

Because ALCS uses the ECB to control reentrant application programs, these programs are also called **ECB-controlled programs**.

Obtaining working storage

Most programs need work areas; for example, to hold working data or intermediate results of calculations. In high-level language programs, you obtain working storage automatically, see "Automatically-allocated storage" on page 84.

Assembler programs do not have this facility, so ALCS provides macros that programs can use to get **blocks** of storage. You might also use these in C programs when you are creating DASD records.

When ALCS allocates a block of storage, it sets a pointer to it in the ECB in the same way as to the message area. The block of storage is "owned" by the entry (Figure 9 on page 7).

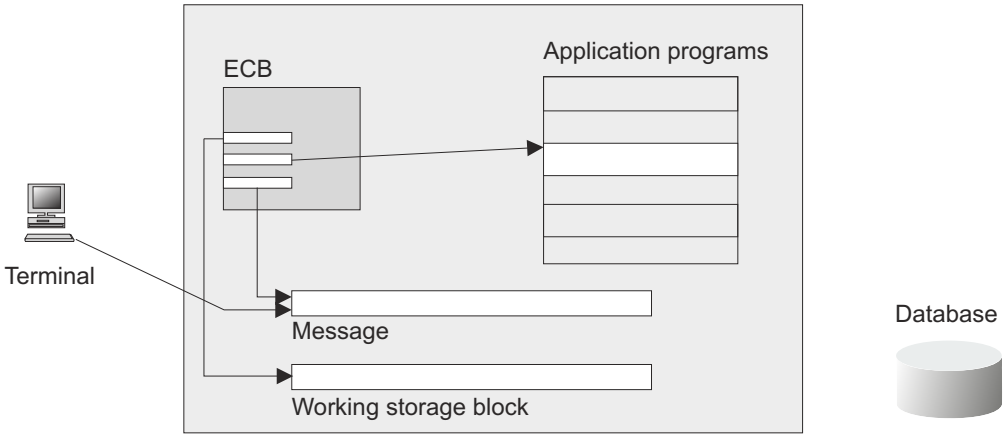


Figure 9. Working storage

ALCS allocates storage in fixed-size pieces called **blocks**. There are up to 9 different block sizes allowed by ALCS (see note "1" on page 7). These are shown in Table 1 on page 7.

Table 1. Block sizes

Block size	Number of bytes of application data	Size recommended for compatibility with TPF
L0	Up to 32K, 127 minimum	127 bytes
L1	Up to 32K, 381 minimum	381 bytes
L2	Up to 32K, 1055 minimum	1055 bytes
L3	Up to 32K, 4000 minimum	
L4	Up to 32K	4095 bytes (required for TPFDF)
L5	Up to 32K	
L6	Up to 32K	
L7	Up to 32K	
L8	Up to 32K	

TPF compatibility:

To be compatible with TPF, all installations should use block sizes L0, L1, L2, and L4 with the recommended values shown in Table 1 on page 7.

An ECB can have up to 16 pointers to blocks of working storage. (This does not include storage that has been allocated automatically.) These pointers are called **storage levels**. ALCS clears the pointers when it releases the storage blocks (for example, when an application exits an entry).

In addition to the 16 storage levels an ECB can also dynamically create one or more DECBs each of which will contain a storage level. For more information see "Data event control blocks (DECBs)" on page 88.

Direct access files

Data is held on DASD in fixed-size records. ALCS supports 8 different record sizes. These are the same as block sizes L1 through L8 shown in Table 1 on page 7. (There are no DASD records of size L0.)

Notes:

1. *(NSYS)* The system programmer defines storage block and DASD record sizes during system generation. Your installation might use only some of the different block and record sizes, for example, L1, L2, L3, and L4.

2. Storage blocks and DASD records are physically larger than the sizes shown in the tables. Records contain additional information that ALCS uses for control purposes.

DASD records are classified into three main classes: **fixed-file records**, **pool-file records**, and **general file records**.

Fixed-file records

The system programmer in your installation allocates some of the records on DASD as **fixed-file records**. Fixed-file records can be used by application programs when the number and size of records needed can be calculated in advance.

Fixed-file record type

The system programmer can allocate records to belong to a particular **fixed-file record type**. A fixed-file record type is identified by a symbol and a number. By convention, the symbol starts with a hash sign (#) [1] followed by from five to seven characters, for example #CONV1, #WORK2. The number can be any number from 1 to 65 535. (ALCS requires a very minor modification to support more than 4094 record types, see your IBM program support representative).

Within a fixed-file record type, the application program can identify a particular record by its **ordinal number** (Figure 10 on page 8).

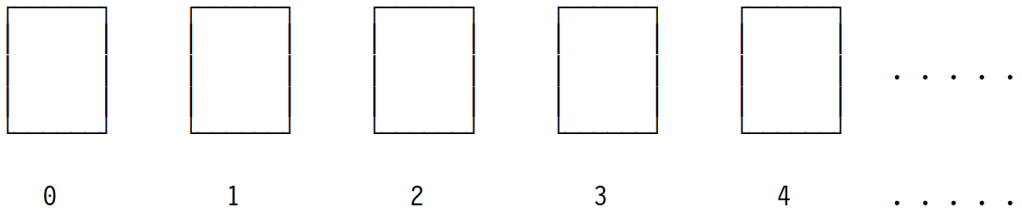


Figure 10. Fixed file ordinals

File addresses

ALCS locates records on DASD by means of a 4-byte **file address**. ALCS provides application programs with services that calculate the file address of a fixed file record and return it as a 4-byte file address (or an 8-byte file address in 4x4 format for use with a DECB). The application supplies the record type and the ordinal number. For more details see “Using direct access files” on page 36 and “Data event control blocks (DECBs)” on page 88.

Miscellaneous file records

To allow programs to use small numbers of fixed-file records, many installations define some miscellaneous fixed-file record types. These are identified by names such as #MISC1, #MISC2, and so on, where the numbers, 1, 2, ..., identify the record size (L1, L2, and so on). Some installations use #MISCS and #MISCL for sizes L1 and L2.

The system programmer can allow individual application programs to use a range of ordinals from one or more of the miscellaneous file records. This saves having fixed-file records allocated specifically for an application.

[1] This character might appear differently on your equipment. It is the character represented by hexadecimal 7B.

Pool-file records

The system programmer also creates pools of records. Each pool contains records of one particular size (L1, L2, and so on). Records in these pools are called **pool-file records**.

The records in a pool file can be either **short-term pool-file records** or **long-term pool-file records**, as follows:

Short-term pool-file records

These are pool-file records (of various sizes) that ALCS makes available ("dispenses") to applications for a short period of time (usually seconds or minutes, typically the life of an entry).

Because an installation only has a limited number of short-term pool records defined, short-term pool-file records are only used to hold data temporarily. When all the short-term records of a particular size become used up, ALCS reuses the oldest ones and the data in these records is overwritten.

Short-term pool-file records are used by applications where the integrity of the data is not critical. For example, an application could use them to build up an output message before sending it to a terminal.

Long-term pool-file records

These are pool-file records (of various sizes) that any application can use when the integrity of the data is critical. Once ALCS has dispensed a long-term pool-file record to an application, it will not dispense the record again unless a number of conditions are met. See “When ALCS redispenses a long-term pool-file record” on page 11.

Long-term pool-file records are suitable for holding long-term data where the amount of data is not known in advance, but where a loss of data would cause a serious problem. For example, the IPARS reservation application uses long-term pool-file records for passenger data in the passenger name record (PNR).

Applications should release a long-term pool-file record when the data is no longer needed; ALCS can then dispense the pool-file record to another application. (But see “Recovering long-term pool-file records” on page 10.)

Chains of pool-file records

An application identifies an individual pool-file record by its file address. ALCS supplies this 4-byte file address when it dispenses a pool-file record to an application.

ALCS keeps track of which pool-file records it has dispensed, but applications share this responsibility. Applications need to record details of each pool-file record that they have used. They normally do this by putting the file address of the pool-file record in another record, thus creating a **chain** of two or more records. The application can put the file address in either:

1. A fixed-file record (Figure 11 on page 9).

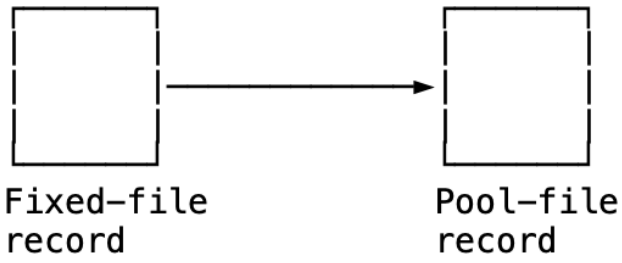


Figure 11. Fixed-file record containing a file address of a single pool-file record

2. A pool-file record that is addressed by another pool file record, or by a fixed-file record (Figure 12 on page 10).

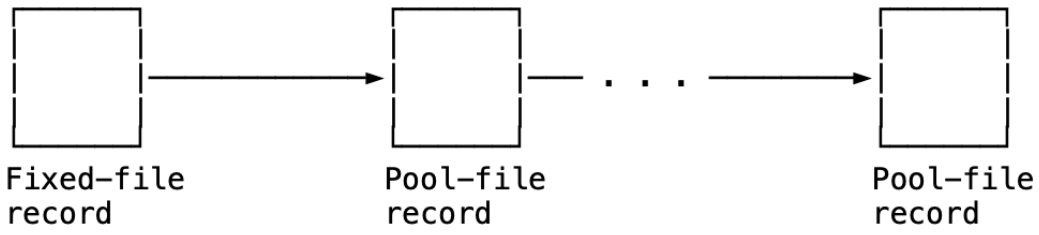


Figure 12. Fixed-file record addressing a chain of pool-file records

A record can contain one or many addresses of other records. This can create very complex structures (see *ALCS Concepts and Facilities*). All such structures must start from a fixed-file record at the top (base) of the structure.

Recovering long-term pool-file records

At regular intervals (every few days, or every week), an installation must recover long-term pool-file records that have been used by applications but which the applications have since released.

ALCS provides a utility program, called **Recoup**, to do this. In addition to checking that the pool-file record has been released, ALCS checks that the record is not part of a chain. This is called **chain chasing**.

Chain chasing

Recoup examines each fixed-file record to see if it contains an address of a long-term pool-file record; if so, it notes that this addressed pool file record is still in use. If the pool-file record contains an address of another pool-file record, it notes that this other pool-file record is also in use, and so on.

It follows chains of pool-file records (see Figure 12 on page 10) until it comes to a pool-file record that does not contain an address of another pool-file record.

When it has examined all the records, Recoup creates a new, up-to-date, table of long-term pool-file records that are available for ALCS to dispense to application programs.

Notes:

1. The system programmer must define where the file address fields are in each type of fixed-file record and in each pool-file record where these are part of a chain. When you use long-term pool-file records you must give the system programmer sufficient information to do this.
2. Recoup can be run while ALCS is operating normally.

Alternative method of recovering long-term pool records

Occasionally, ALCS installations can not or do not run Recoup in time to prevent long-term pool depletion. This can happen if the operations staff simply fail to run Recoup. But it is more likely to happen because unexpectedly high load, or application error, causes pool depletion to occur much sooner than expected.

ALCS already includes facilities to reduce the possibility of unexpected pool depletion. In particular:

- ALCS monitors the long-term pool dispense rate and warns if the current rate will deplete pool in less than 24 hours.
- ALCS controls the maximum number of dispenses that a single entry can perform.

However ALCS installations sometimes require an "emergency" method to recover long-term pool. If an ALCS system does run out of long-term pool, the application will not work.

An ALCS system that has completely run out of long-term pool can continue functioning by redispensing long-term pool records that have been released since the last Recoup.

To do this, ALCS maintains a list of released file addresses, in release order. If a pool is depleted, ALCS will start to redispense file addresses from this list. See *ALCS Installation and Customization* for an explanation of how to implement this facility using the `pdulogstream` parameter of the SCTGEN macro.

When ALCS redispenses a long-term pool-file record

Before ALCS redispenses a long-term pool-file record, all the following conditions must be met:

- An application has released the record.
- The record is not referenced by another record (the record is not in a chain).
- Recoup has been run and has identified the record as fulfilling the two conditions mentioned above.

ALCS dispenses long-term pool-file records on a cyclical basis, so there will be an additional delay before it redispenses a particular long-term pool-file record. These conditions make it almost impossible for ALCS to dispense a long-term pool-file record that is currently in use.

Application global area

Some existing applications written for use with ALCS, TPF, or predecessor systems, keep data in an area of processor storage called the **application global area**. You are recommended not to develop new applications that keep data in the application global area. However, if you are updating existing programs or you are writing new applications that coexist with existing programs, you might need to access data in the application global area. "Global records and fields" on page 56 describes how to do this.

General files and general data sets

General files and general data sets (GDSs) contain DASD records that application programs can read and write. In addition, batch programs can read records from and write records to these files. You can use general files and GDSs when application programs need to pass information to or receive information from, offline programs.

In some predecessor systems general files and GDSs are different, and application programs must use different methods to access them. In ALCS they are physically identical, but ALCS supports the two methods to access them. The application program must access a file as either a general file or a GDS (and not mix the two access methods).

Note: In new applications you might prefer to use APPC, SQL, MQI, or WAS instead of general files. See "APPC communication calls and conversation services" on page 17, "Using Structured Query Language (SQL)" on page 64, "Message Queue Interface (MQI)" on page 23, and "Optimized Local Adapters (OLA) for WebSphere Application Server" on page 28.

General files

Application programs identify a general file by the **general file number** - a decimal number in the range 1 through 255. (One general file, number 0, is reserved for use by ALCS itself.)

ALCS provides application programs with a service that calculates the 4-byte file address of a general file record. (The application program provides the general file number and the ordinal number of the record.)

General data sets

Application programs identify a general data set by the data set name.

ALCS provides application programs with a service that calculates the 4-byte file address of a general data set record. (The application program provides the data set name and the ordinal number of the record.) ALCS also provides services to open and close general data sets.

Virtual file access

ALCS retains records in processor storage after they have been read from DASD or written to DASD (or both). ALCS retains the records in buffers called **virtual file access** (VFA) buffers. (See *ALCS Concepts and Facilities* for more information about VFA.) The system programmer specifies the number of VFA buffers that ALCS allocates.

Normally, there are many more records on DASD than there are VFA buffers. ALCS manages the VFA buffers so that the most recently referenced records remain in buffers. When an application program reads a record, ALCS provides a copy from the VFA buffer (if the record is in a buffer) or reads the record from DASD (if the record is not in a buffer).

The use of VFA buffers is transparent to the application program. The program does not know (and cannot find out) if a particular record is in a VFA buffer.

Sequential files

Sequential files contain records that are written (and subsequently read) in sequence, not randomly. In this way they are equivalent to magnetic tapes. ALCS sequential files are identified by 3-character names.

There are two types of sequential file that application programs can use:

Real-time sequential files
: These files are opened automatically when ALCS is started. Any application program can write records to real-time sequential files, but cannot read records from them. Such files are useful for logging.

General sequential files
: An application must open a general sequential file and must specify whether it is opening the file for input (to read records) or output (to write records). Several applications can share the use of the same general sequential file, but they must do this in a controlled way. See "Using sequential files" on page 52.

Communication with users, other systems, and other applications

An ALCS system does not exist in isolation. There are several areas where ALCS interacts with areas outside the main computer installation:

- Terminals in remote locations, connected over a VTAM network
- Connections to other installations using a Synchronous Link Control (SLC) link, particularly those to a SITA or ARINC network
- Connections to other installations using an LU 6.1 link
- Connections to other installations or devices using Transmission Control Protocol/Internet Protocol (TCP/IP)
- NetView operator identifiers
- Advanced program-to-program communications (APPC), Message Queue Interface (MQI), optimized local adapters (OLA), and so on.

Terminals on a VTAM network are controlled by NetView VTAM commands. Terminals connected using SLC are controlled by ALCS operator commands.

Each communication resource is identified by a **communication resource identifier** (CRI) and **communication resource name** (CRN).

ALCS applications

An ALCS **application** performs some useful user-related task. For example, in an airline environment typical applications are reservations, message switching, departure control, and so on. Each application usually arranges for processing to be carried out by a number of **application programs** which transfer control between themselves, as shown in Figure 13 on page 13.

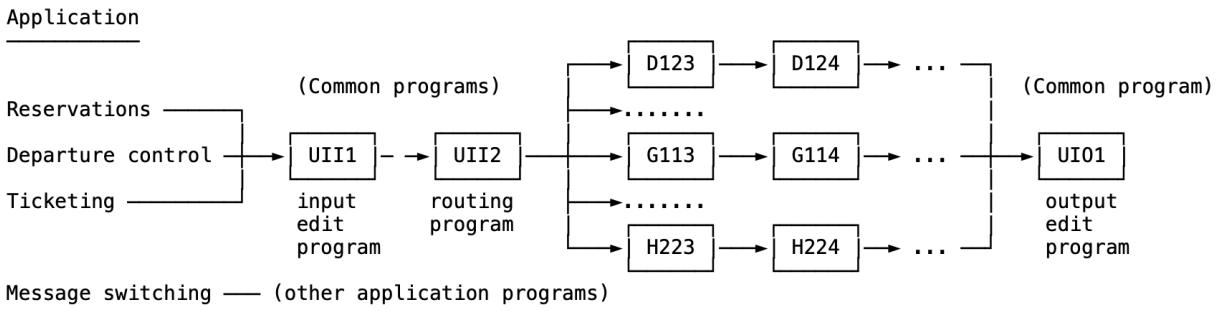


Figure 13. Example ALCS applications and their constituent programs

ALCS applications written in assembler often make use of common programs for input and output editing of messages, as shown in Figure 13 on page 13.

IPARS applications

Two sample applications are supplied with ALCS. Together they constitute the International Programmed Airlines Reservation System (IPARS). They are:

- IPARS reservations application
- IPARS message switching application.

Installations can use one or both of these applications to test that ALCS is working correctly. You can use them as supplied, or as a basis for more comprehensive applications of your own.

Chapter 2. Communication

This chapter describes the communication facilities of ALCS and explains how application programs can use them.

Communication facilities

ALCS allows application programs to communicate with **resources**, where a resource is one of the following:

- Another (or the same) application
- A display or printer terminal controlled by VTAM (the terminal is usually an IBM 3270 terminal or a simulation of an IBM 3270 by an IBM
- A NetView operator ID
- An ATA/IATA synchronous link control (SLC) link
- An airlines line control (ALC) display or printer terminal connected through SLC, X.25, or Airlines Line Control Interconnection (ALCI)
- A world trade teletypewriter (WTTY) link
- An LU 6.1 link or parallel session.

Application programs do not themselves initiate input from terminals on the communication network. ALCS does this, and generates a new entry for each input message. Application programs process these input messages and can send response messages to the originating terminal or to other resources.

How communication resources are identified

A communication resource must be "known" to ALCS. The system programmer tells ALCS about communication resources (during system generation or later by using an operator command) and ALCS holds information about resources in a **communication table**.

An application program usually identifies a particular communication resource by means of a 3-byte **communication resource identifier** (CRI). Other applications use a 8-character **communication resource name** (CRN). Some applications, for example the IPARS message switching application, (see “IPARS applications” on page 13) identify WTTY links and SLC links by 1-byte identifier. ALCS derives the CRI from this 1-byte identifier.

Communication resources on other systems

ALCS can use communication resources belonging to other systems. For example a communication resource can be a 3270 terminal connected to an ALCS, TPF, or CICS® system on a remote processor.

Higher-level communication

In addition to communicating with ALCS communication resources, ALCS application programs can use callable services to:

- Participate in **conversations**, using the APPC/MVS and CPI-C callable services (see “APPC communication calls and conversation services” on page 17).
- Communicate through **message queues**, using the Message Queue Interface (MQI) callable services (see “Message Queue Interface (MQI)” on page 23).
- Communicate using TCP/IP
- Communicate with WebSphere® Application Server for z/OS applications using the optimized local adapters (OLA) interface.

Conversation partners for APPC/MVS and CPI-C, queues for MQI, and registrations and connections with OLA do not have ALCS CRIs or CRNs.

Receiving input messages

In ALCS applications, input messages are usually coded and short. For example, to inquire about the availability of flights between London and Singapore on October 10 of the current year at 1400 hours, an airline reservation agent might enter the following message:

```
A 10 OCT LON SIN 1400
```

Most application programs process messages after they have been examined (and possibly preprocessed) by an **input edit program**. The following descriptions of input message formats would mostly concern the programmer writing the input edit program.

Input message formats

ALCS passes the input message to the input edit program in a one of several message formats. The format in which a particular input edit program receives a message is defined by the system programmer in the ALCS communication generation (described in *ALCS Installation and Customization*).

These message formats are defined in assembler language program DSECT macrodefinitions or in C language header files. See “Message formats” on page 17.

ALCS C language support allows you to use **memory files**. (See *C/C++ Programming Guide* for an explanation of memory files.) You can use the input and output C functions (such as `fopen`, `fread`, `fwrite`, and `fclose`) to create and process memory files.

ALCS supports the standard input and output streams (`stdin` and `stdout`) as memory files. C language programs receive messages from the standard input stream `stdin`.

Routing control parameter list

ALCS puts information about the origin of a message in a **routing control parameter list** (RCPL) in the ECB. Fields in the RCPL are defined in assembler language programs by the RC0PL DSECT macro and in C language programs in a header file `<c$rc0pl.h>`.

For a description of the RCPL contents, see Appendix D, “Messages and message formats,” on page 215.

ALCS services for communication

The following table summarizes assembler macros and C functions that you can use for communication in ALCS application programs.

Assembler macro	C function	Used to
CRASC		Send a message to a CRAS printer terminal, or to the CRAS printer terminal associated with a CRAS display terminal.
DISPC		Build a multiline output message, and optionally send it to a terminal or printer. DISPC can invoke the ALCS scrolling facility.
ROUTC	`routc`	Send a message to a terminal or to another application. The destination terminal or application can be owned (hosted) by the same ALCS system as the originating application, or by another system in the same or in a different processor.
SENDC		Send a message. A macro parameter, the type code, allows a variety of message types to be transmitted.

Assembler macro	C function	Used to
SLMTC		Send a special message to the printer. This macro enables the application to control the printer. The application is notified when the message has been printed successfully. It is the application's responsibility to handle error conditions occurring on the printer.
WTOPC	wtopc	Build and optionally send an output message, or send a preprepared message from a table

As mentioned in “Receiving input messages” on page 15, C language application programs can use standard C functions to "read" the input message from stdin. Similarly, they can use standard C functions to "write" the reply message to stdout.

C language programs cannot directly use the specialized assembler macros that have no equivalent C function. If they need these specialized services, they must call an assembler program to request them.

ALCS does not receive or transmit messages character-by-character. When "reading" from stdin, a C language program is retrieving characters from a complete input message (*not* reading characters directly from the terminal keyboard). Similarly, when "writing" to stdout, a C language program is building a message that ALCS sends when the program ends (*not* writing characters directly to the terminal display).

The following table summarizes assembler macros and C functions used to interrogate and update the communication table.

Assembler macro	C function	Used to
COMCC		Updates an entry in the communication table in storage. Application programs can alter only selected fields.
COMIC	comic	Extract information about a communication resource.

Validating input data

An undetected incorrect or wrongly-formatted input message could result in the real-time database being corrupted. To avoid this, application programs must validate the format of every input message. Where possible, they must also check the type, validity, and consistency of the input message data. For example:

Data type
Check that decimal data is specified in decimal, hexadecimal data in hexadecimal, and so on. (For example, '4B' is not a valid decimal number, and '5Z' is not a valid hexadecimal one.) Failure to check for this type of error can cause data exceptions when converting to internal formats.

Data validity
Check that the data is valid. For example, if an airline flight number is specified, check that the flight number exists. If a date is specified, check that it is a valid date (for example, 56 JUL is not valid).

Data consistency
When two or more values are specified in the same message, check that they are consistent. For example, when a flight number and a flight date are specified, check that the flight is scheduled for that date.

Sending response messages to the originating terminal

Application programs usually send any response to the originating terminal. In C language programs the program simply outputs the message to stdout.

The output message format depends on the destination of the message. If an application program sends a message to the originating terminal, ALCS converts the message into the correct format.

Assembler application programs normally assemble the message and pass control to an **output edit program** which puts the message into a suitable format and sends it to the originating terminal. An assembler edit program can get the CRI of the originating terminal from the routing control parameter list (RCPL) (see "RCPL contents" on page 223) and put this CRI as the destination in the output message. It then calls an appropriate assembler macro to request ALCS to deliver the message.

Conventionally, only the output edit program for an application executes send-type macros. This is because the parameters used with the SENDC macro depend on the type of terminal to which the response is going, and on other criteria not related to the message content.

Sending unsolicited messages

In addition to sending a reply to the originating terminal, an application program can send messages that are not responses. You can send such messages to terminals or other resources; they are called **unsolicited messages**. Examples of unsolicited messages are broadcast messages sent to other terminals.

To send an unsolicited message, an application sets up the message in AMSG (or OMSG) format and requests the ROUTC (`routc`) service to send the message.

The application puts the destination CRI in the RCPL. Applications that must send messages to many terminals can request the ROUTC (`routc`) service many times, changing the destination address (CRI) between calls. Alternatively such an application could create a new entry to send each message. See "Creating new entries" on page 101.

Message formats

ALCS supports several message formats, described in Appendix D, "Messages and message formats," on page 215. The different message formats are associated with different types of terminal (IBM 3270 terminals, teletypewriters, ALC terminals, and so on).

The system programmer can specify in which input format an application program is to receive messages. ALCS converts any input message into this format before passing it to the application. ALCS also converts any reply message created by the application into the appropriate format. However if the application sends messages to several different terminals (for example, a printer and a display terminal) it might have to format at least some of the messages itself.

APPC communication calls and conversation services

ALCS allows application programs to communicate using the Common Programming Interface for Communications (CPI-C) calls. ALCS also supports unauthorized APPC/MVS transaction processing conversation callable services, and (in assembler language programs only) TPPCC macros. The APPC/MVS services enable MVS application programs to communicate with other application programs, using communication protocols provided by the SNA network.

When designing application programs that use these services you need to consider design points such as existence time and conversation integrity.

Note: The CPI-C and APPC/MVS documentation refers to conversations between "transaction programs", "application programs", and "programs". In ALCS, it is the entry which participates in a conversation, not the ALCS application program or programs.

ZSTAT message counts and data collection statistics

You can use the installation-wide monitor exit USRAPPC to identify those CPI-C and APPC/MVS calls which process input or output messages. This enables ALCS to increment the input and output APPC message counts which are displayed by ZSTAT command. When data collection is collecting statistics about messages, this also enables ALCS to increment the statistics for APPC input and output messages.

Program preparation

For information about the procedure for preparing an ALCS application program that includes CPI-C or APPC/MVS calls, see Chapter 7, “Application program management,” on page 124.

Including CPI-C calls in application programs

For more information about including CPI-C calls in application programs, and for information about the characteristics of these programs, see *Common Programming Interface Communications Reference*.

Including CPI-C calls in assembler programs

You can use a CPI-C or APPC/MVS call at any place in an assembler program where you can use an assembler instruction. Code the CPI-C or APPC/MVS call as a normal MVS CALL macroinstruction.

Figure 14 on page 18 shows an CMINIT (INITIALIZE_CONVERSATION) call coded in assembler.

```
         CALL  CMINIT,             CPI-C CMINIT CALL                  X
               (EBW008,            CONVERSATION_ID                    X
               EBW020,             SYM_DEST_NAME                      X
               EBW000),            RETURN_CODE                        X
               VL,MF=(E,EBX000)
```

Figure 14. Example CPI-C call -- assembler

Including CPI-C calls in C language programs

You can include CPI-C or APPC/MVS function calls at any place in a C language program where you can place an executable C statement.

Figure 15 on page 18 shows a CPI-C function call coded in C.

```
#include <atbcmc.h>          /* CPI-C C definitions           */

char sym_dest [10] = "DEST1    ";
char conv_id[10]   = "         ";
long int ret_code = 0;

CMINIT (
  conv_id,                   /* Out - Conversation ID         */
  sym_dest,                  /* In  - Symbolic Dest Name      */
  &ret_code );               /* Out - Return Code             */

if (ret_code != CM_OK)
{
  printf("CMINIT call failed - return code=%d/n", ret_code);
  exit(0);
}
```

Figure 15. Example CPI-C call -- C language

Including CPI-C calls in COBOL programs

You can include CPI-C or APPC/MVS function calls in the PROCEDURE DIVISION of a COBOL program.

Figure 16 on page 19 shows a CPI-C function call coded in COBOL.

```
WORKING-STORAGE SECTION.
  ⋮
01 CONV-ID               PIC X(8) VALUE '        '.
01 SYM-DEST-NAME          PIC X(8) VALUE 'DEST1   '.
  ⋮
* COPY THE COBOL INTERFACE DECLARATION FILE FOR CPI-C
* THIS DEFINES CM-RETCODE AND CM-OK
     COPY ATBCMCOB.
  ⋮
PROCEDURE DIVISION.
  ⋮
*-----------------------------------------------------------------*
*    INITIALIZE CONVERSATION                                      *
*-----------------------------------------------------------------*
CALL 'CMINIT' USING CONV-ID
                    SYM-DEST-NAME
                    CM-RETCODE.
IF CM-RETCODE IS NOT EQUAL TO CM-OK THEN
  DISPLAY 'INITIALIZE CONVERSATION FAILED -- CODE=' CM-RETCODE.
  ⋮
```

Figure 16. Example CPI-C call -- COBOL

Including APPC/MVS calls in application programs

For information about including APPC/MVS calls in ALCS application programs and for information about the characteristics of these programs, see *MVS Programming: Writing Transaction Programs for APPC/MVS*.

Including APPC/MVS calls in assembler programs

Figure 17 on page 19 shows a GET_CONVERSATION call coded in assembler.

```
CALL  ATBGETC,            APPC/MVS GET_CONVERSATION          X
               (EBW008,             CONVERSATION_ID                   X
               EBW020,              TYPE                              X
               EBW030,              PARTNER_LU_NAME                   X
               EBW050,              MODE_NAME                         X
               EBW060,              SYNC_LEVEL                        X
               EBW070,              CORRELATOR                        X
               EBW000),             RETURN_CODE                       X
               VL,MF=(E,EBX000)
```

Figure 17. A GET_CONVERSATION call -- assembler

Including APPC/MVS calls in C language programs

Figure 18 on page 20 shows an APPC/MVS function call coded in C.

```
#include <atbpbc.h>        /* APPC/MVS C definitions           */

/* define fields used in APPC/MVS call */

char * conv_id;           /* pointer to Conversation ID        */
                          /* set by previous call to ATBALLC   */
long int ret_code = 0;
long int send_and_flush = ATB_SEND_AND_FLUSH;
long int msg_length = 0;
long int none = ATB_NONE;
char msg[] = "test message ......................";
long int req_send_received;
long int notify_type_none = ATB_NOTIFY_TYPE_NONE;
#define NOTIFY_NONE ( (char *) &notify_type_none)

msg_length = strlen(msg);

/*-------------------------------------------------------------*/
/* send a message with ATBSEND synchronously                   */
/*-------------------------------------------------------------*/

ATBSEND(
  conv_id,                 /* In  - Conversation ID         */
  SEND_and_flush,          /* In  - Send Type               */
  &msg_length,             /* In  - Send Length             */
  &none,                   /* In  - Buffer ALET             */
  msg,                     /* In  - Buffer                  */
  &req_send_received,      /* Out - Req To Send Received    */
  NOTIFY_NONE,             /* In  - Notify type             */
  &ret_code );             /* Out - Return Code             */

if (ret_code != ATB_OK)
{
  printf("ATBSEND call failed - return code=%d/n", ret_code);
  exit(0);
}
```

Figure 18. APPC/MVS function call -- C language

Including APPC/MVS calls in COBOL programs

Figure 19 on page 21 shows an APPC/MVS function call coded in COBOL.

```
 DATA DIVISION.

   WORKING-STORAGE SECTION.
   77   RC            PIC S9(9) COMP.
   77   EXIT-OK-RC    PIC S9(9) COMP VALUE ZERO.

*  DEFINE PUTM ERROR NUMBER  - X'FFFF01'
   77   ERROR-PUTM-RC PIC S9(9) COMP VALUE 16776961 .

*  DEFINE MESSAGE ISSUED
   77   MSGL01        PIC 9(9) COMP VALUE 30.
   01   MSG01.
   02   FILLER        PIC X(21) VALUE 'ATBSEND - FAILED RC='.
   02   MSG01-RC      PIC 9(9).

*  DEFINE WORKING VARIABLES
   77   CONV-ID       PIC X(8)  VALUE SPACES.
   77   ALET-NONE     PIC 9(9)  COMP VALUE ZERO.
   77   MSG-LENGTH    PIC 9(9)  COMP VALUE 80.
   01   MSG.
   02   MSG-TEXT      PIC X(80) VALUE 'TEST MESSAGE........'.
*-----------------------------------------------------------------*
*  THE ATBPBCOB COPY BOOK MEMBER IS SUPPLIED WITH APPC/MVS AND   *
*  DEFINES THE ATB-.... DATA FIELDS AND CONDITION NAMES          *
*-----------------------------------------------------------------*
   COPY ATBPBCOB.
  ⋮
 PROCEDURE DIVISION.
  ⋮
*-----------------------------------------------------------------*
*    SEND A MESSAGE WITH ATBSEND SYNCHRONOUSLY                    *
*    CONV-ID IS SET BY A PREVIOUS CALL TO ATBALLC                 *
*-----------------------------------------------------------------*
     MOVE 1          TO ATB-SEND-TYPE.
     MOVE 0          TO ATB-NOTIFY-TYPE.

     CALL 'ATBSEND'  USING CONV-ID
                           ATB-SEND-TYPE
                           MSG-LENGTH
                           ALET-NONE
                           MSG
                           ATB-STATUS-RECEIVED
                           ATB-NOTIFY-TYPE
                           ATB-RETCODE.
*-----------------------------------------------------------------*
*    IF ERROR OCCURED - DISPLAY A MESSAGE AND EXIT                *
*-----------------------------------------------------------------*
     IF NOT ATB-OK
       DISPLAY 'ATBSEND FAILED RETCODE=' ATB-RETCODE
       MOVE ATB-RETCODE TO MSG01-RC
       CALL 'DXCQPUTM' USING MSG01 MSGL01 RC
       PERFORM CHECK-RC-PUTM
       CALL 'DXCQSTOP' USING EXIT-OK-RC.
  ⋮
*-----------------------------------------------------------------*
*    PROCEDURE TO CHECK THE RETURN CODE (RC) AFTER USING          *
*    DXCQPUTM AND EXIT WITH ERROR IF RC IS NOT ZERO               *
*-----------------------------------------------------------------*
 CHECK-RC-PUTM.
     IF RC IS NOT EQUAL TO ZERO THEN
       CALL 'DXCQSTOP' USING ERROR-PUTM-RC.
```

Figure 19. APPC/MVS function call -- COBOL

APPC/MVS calls with asynchronous notification of completion

APPC/MVS calls can specify either synchronous or asynchronous notification of completion. This is determined by NOTIFY_TYPE (see *MVS Programming: Writing Transaction Programs for APPC/MVS*).

If NOTIFY_TYPE is nonzero, the application must call the ALCS ATAWAIT (`atawait`) monitor service instead of MVS `WAIT`, to wait for the asynchronous event to complete.

ATAWAIT (`atawait`) causes the entry to lose control if no asynchronous event has yet completed. The entry regains control when any one of these asynchronous events completes. ATAWAIT (`atawait`) does

not cause the entry to lose control if one or more of the asynchronous events has already completed. The format of the ATAWAIT call in assembler language programs is as follows:

```
         CALL  ATAWAIT
```

Figure 20 on page 23 shows an example of the use of ATAWAIT. This example routine waits for two ALLOCATE calls (for two different conversations) to complete. For each ALLOCATE call, NOTIFY_TYPE specifies an 8-byte field comprising a fullword containing the value 1, followed by a fullword containing the address of another fullword in entry storage. This last fullword is the equivalent of an MVS event control block.

APPC/MVS sets the RETURN_CODE when the ALLOCATE call is scheduled, and returns information in the event control block when it has completed the ALLOCATE.

In C language programs, you can invoke the APPC/MVS C functions asy by specifying the Notify_type parameter as ATB_NOTIFY_TYPE_ECB and providing an event control block. ALCS provides the `atawait` C function to enable programs to wait for the completion of asynchronous APPC/MVS calls. On completion, `atawait` posts the event control block and sets the completion code field with the status of the C function. For a description of `atawait`, see *ALCS Application Programming Reference - C Language*. This book also contains an example of use of `atawait`.

Ending the conversation

Applications should deallocate conversations explicitly. If they do not, ALCS takes the following action:

Normal completion

When an application program exits normally, ALCS deallocates any conversations that are still allocated (provided they are in a correct state).

Abnormal completion

When an application program issues a dump with exit, or terminates abnormally, with one or more conversations still allocated, ALCS deallocates the conversations with an abend condition (Deallocate_abend).

```
         ...
         MVC   EBW030(L'PARM0),PARM0  SET UP NOTIFY_TYPE PARAMETER
         XC    EBW038(4),EBW038    CLEAR EVCB TO ZEROS
         LA    R14,EBW038          LOAD ADDRESS OF EVCB
         ST    R14,EBW034          STORE INTO NOTIFY_TYPE PARAMETER

CALL  ATBALLC,            APPC/MVS ALLOCATE 1                 X
               (PARM1,              BASIC_CONVERSATION              X
               PARM2,               SYMBOLIC_DEST_NAME              X
               ...,                                                 X
               ...,                                                 X
               EBW008,              CONVERSATION_ID                 X
               EBW030,              NOTIFY_TYPE                     X
               EBW000),             RETURN_CODE                     X
               VL,MF=(E,EBX000)
         ...
******** Test RETURN_CODE in EBW000-EBW003
         ...
         MVC   EBW050(L'PARM0),PARM0  SET UP NOTIFY_TYPE PARAMETER
         XC    EBW058(4),EBW058    CLEAR EVCB TO ZEROS
         LA    R14,EBW058          LOAD ADDRESS OF EVCB
         ST    R14,EBW054          STORE INTO NOTIFY_TYPE PARAMETER

CALL  ATBALLC,            APPC/MVS ALLOCATE                   X
               (PARM3,              BASIC_CONVERSATION              X
               PARM4,               SYMBOLIC_DEST_NAME              X
               ...,                                                 X
               EBW016,              CONVERSATION_ID                 X
               EBW050,              NOTIFY_TYPE                     X
               EBW000),             RETURN_CODE                     X
               VL,MF=(E,EBX000)
         ...
******** Test RETURN_CODE in EBW000-EBW003
         ...
TEST     EQU   *
         CALL  ATAWAIT             WAIT TILL ONE OR MORE ARE POSTED

         TM    EBW038,X'40'        HAS FIRST ALLOCATE COMPLETED
         BZ    TEST                NO - BRANCH TO WAIT AGAIN

         TM    EBW058,X'40'        HAS SECOND ALLOCATE COMPLETED
         BZ    TEST                NO - BRANCH TO WAIT AGAIN
         ...
         Test return codes in EBW039-EBW041 and EBW059-EBW061
         ...

PARM0    DC    F'1'                NOTIFY_TYPE PARAMETER
PARM1    ...
PARM2    ...
PARM3    ...
PARM4    ...
```

Figure 20. An ATAWAIT call -- assembler

Message Queue Interface (MQI)

ALCS allows application programs to communicate using the common programming interfaces for messages and queuing. MQI provides:

- Additional connectivity between ALCS systems in different regions or under different z/OS systems
- Connectivity to other applications, for example Time Sharing Option (TSO), IMS, and CICS, and with applications running on other platforms, for example, OS/2, OS/400®, AIX®, and a wide variety of non-IBM platforms.

Figure 21 on page 24 provides an overview of MQI.

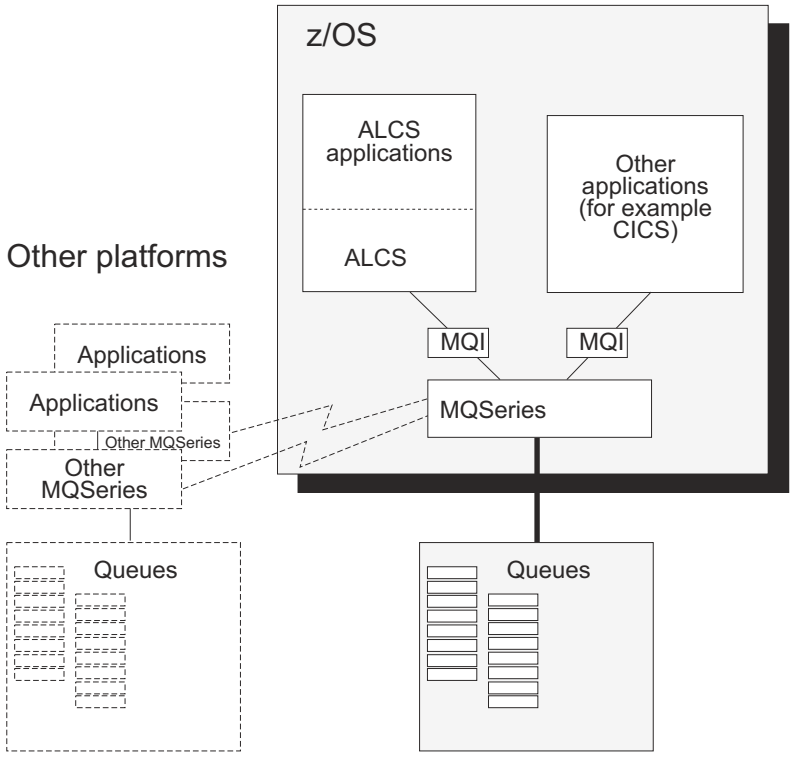


Figure 21. Using MQI for integrated transaction processing

When designing application programs that use these calls you need to consider design points such as existence time, data security, synchronization, and distributed processing techniques.

ZSTAT message counts and data collection statistics

You can use the installation-wide monitor exit USRMQI1 to identify those MQI calls which process input or output messages. This enables ALCS to increment the input and output MQ message counts which are displayed by ZSTAT command. When data collection is collecting statistics about messages, this also enables ALCS to increment the statistics for MQ input and output messages.

Program preparation

For information about the procedure for preparing an ALCS application program that includes MQI calls, see the services in the various sections of "Application programming languages" on page 124.

Including MQI calls in application programs

For detailed information about including MQI calls in application programs, and for information about the characteristics of these programs, see *WebSphere MQ for z/OS Application Programming Reference*. For information about the ALCS initiation queue and trigger queue, see *ALCS Concepts and Facilities*.

ALCS does not support the use of syncpoint management for MQI calls. However, to allow the message-queueing API statements in a program to be ported to or from other environments, an application program can issue MQCMIT and MQBACK calls. ALCS takes no action for these calls and does not pass them to MQSeries®. For backward compatibility, the CSQBCMT and CSQBBAK calls are available as synonyms for MQCMIT and MQBACK.

Application programs do not need to connect to, or disconnect from, a queue manager. ALCS automatically connects them to the queue manager instance that is defined by the ALCS system programmer or selected by the ALCS operator (on the `ZCMQI` command).

Application programs should set the MQHCONN parameter to binary zeroes on MQI calls (ALCS ignores this parameter).

You can include MQI calls at any place in an assembler program where you can place an assembler instruction. Code the MQI call as a normal MVS CALL macroinstruction.

Including MQI calls in assembler programs

Figure 22 on page 25 shows how to code an MQOPEN call in an assembler program.

```
         CALL  MQOPEN,             OPEN OBJECT                      X
               (MQHCONN,           CONNECTION HANDLE                X
               MQOD,               OBJECT DESCRIPTOR                X
               OPTIONS,            OPTIONS                          X
               MQHOBJ,             OBJECT HANDLE                    X
               COMPCODE,           COMPLETION CODE                  X
               REASON),            REASON CODE                      X
               VL,MF=(E,EBX008)
```

Figure 22. An MQI MQOPEN call -- assembler

Including MQI calls in high-level language programs

Similarly, you can include MQI calls at any place in a C language or COBOL program where you can place an executable statement.

Figure 23 on page 25 shows how to code an MQOPEN call in a C language program, Figure 24 on page 25 shows how to code an MQOPEN call in a COBOL program.

```
MQOPEN ( Hconn,            /* connection handle      */
         &ObjDesc,         /* object descriptor      */
         Options,          /* options                */
         &Hobj,            /* object handle          */
         &CompCode,        /* completion code        */
         &Reason           /* reason code            */
       );
```

Figure 23. An MQI MQOPEN call -- C language

```
           CALL 'MQOPEN' USING HCONN
                               MQOD
                               OPTIONS
                               HOBJ
                               COMPCODE
                               REASON.
```

Figure 24. An MQI MQOPEN call -- COBOL

MQI calls with asynchronous notification of completion

When you use an MQGET call to retrieve a message from one or more queues, you can specify either synchronous or asynchronous notification of completion. You do this by using the MQGMO_SET_SIGNAL option and the MQGMO_SIGNAL1 field in the get message options structure (see *WebSphere MQ for z/OS Application Programming Reference*).

When you specify MQGMO_SET_SIGNAL, your application must subsequently call the ALCS `MQAWAIT` (`mqawait`) service, instead of MVS `WAIT`, to wait for the asynchronous completion signal.

`MQAWAIT` causes the entry to lose control if no MQGET call with signal request has completed. The entry regains control when any one of these calls completes. `MQAWAIT` does not cause the entry to lose control if one or more MQGET calls with signal request has already completed. See *WebSphere MQ for z/OS Application Programming Reference* for more information.

The format of the MQAWAIT call in assembler language programs is as follows:

```
        CALL  MQAWAIT
```

Figure 25 on page 27 shows an example of the use of MQAWAIT call in an assembler program. For an example of the use of an mqa`wait` function call in a C language program, see *ALCS Application Programming Reference - C Language*.

WebSphere MQ for z/OS sets the COMPCODE and REASON when the MQGET call is scheduled, and returns information in the event control block when it has completed the MQGET.

Example MQAWAIT call - assembler

This example routine waits for two MQGET calls (for two different queues) to complete. For each MQGET call, it specifies MQGMO_SET_SIGNAL and the MQGMO_SIGNAL1 field in the get message options structure contains the address of a fullword in entry storage. This last fullword is the equivalent of an MVS event control block.

```
         ...
COMPCODE EQU   EBW000
REASON   EQU   EBW004
USERECB1 EQU   EBW008
USERECB2 EQU   EBW012
         ...
******** Open the queues for input
         ...
         XC    BUFFER1(72),BUFFER1 CLEAR GET MESSAGE BUFFER
         MVC   BUFFER1L(4),=A(72)  LENGTH OF BUFFER AREA FOR MESSAGE
         SPACE 1
         L     R03,CE1CR3          LOAD MQMD BASE
         LA    R04,MQMD            LOAD MOVE-TO ADDRESS
         LA    R05,MQMD_LENGTH     LOAD MOVE_TO LENGTH
         LA    R06,MQMD01          LOAD MOVE-FROM ADDRESS
         LR    R07,R05             LOAD MOVE-FROM LENGTH
         MVCL  R04,R06             SET UP MESSAGE DESCRIPTOR
         SPACE 1
         L     R02,CE1CR2          LOAD MQGMO BASE
         MVC   MQGMO(MQGMO_LENGTH),MQGMO01 SET UP GET MESSAGE OPTIONS
         SPACE 1
         L     R00,=A(MQWI_UNLIMITED)  UNLIMITED WAIT
         ST    R00,MQGMO_WAITINTERVAL  SET WAIT INTERVAL
         SPACE 1
         LA    R00,USERECB1        POINT TO SIGNAL EVCB
         ST    R00,MQGMO_SIGNAL1   SET POINTER TO SIGNAL EVCB
         SPACE 1

         CALL  MQGET,                                                  X
               (MQHCONN,MQHOBJ,MQMD,MQGMO,BUFFER1L,BUFFER1,DATAL1,     X
               COMPCODE,REASON),                                       X
               VL,MF=(E,EBX008)
         SPACE 1
         CLC   =A(MQCC_OK),COMPCODE  WAS IT SUCCESSFUL
         BE    MQM0100               YES - BRANCH
         SPACE 1
         CLC   =A(MQRC_SIGNAL_REQUEST_ACCEPTED),REASON OK TO CONTINUE
         BE    MQM0100                                 YES - BRANCH
         SPACE 1
         CLC   =A(MQRC_TRUNCATED_MSG_ACCEPTED),REASON  OK TO CONTINUE
         BE    MQM0100                                 YES - BRANCH
         SPACE 1
         B     MQM0ERR1            OTHERWISE TAKE ERROR BRANCH
         EJECT ,
MQM0100  DC    0H'0'
         XC    BUFFER2(72),BUFFER2 CLEAR GET MESSAGE BUFFER
         MVC   BUFFER2L(4),=A(72)  LENGTH OF BUFFER AREA FOR MESSAGE
         SPACE 1
         L     R14,=A(MQMD_LENGTH) LOAD MQMD LENGTH
         AR    R03,R14             GET MQMD BASE
         LA    R04,MQMD            LOAD MOVE-TO ADDRESS
         LA    R05,MQMD_LENGTH     LOAD MOVE_TO LENGTH
         LA    R06,MQMD02          LOAD MOVE-FROM ADDRESS
         LR    R07,R05             LOAD MOVE-FROM LENGTH
         MVCL  R04,R06             SET UP MESSAGE DESCRIPTOR
         SPACE 1
         L     R14,=A(MQGMO_LENGTH)  LOAD MQMD LENGTH
         AR    R02,R14             GET MQGMO BASE
         MVC   MQGMO(MQGMO_LENGTH),MQGMO02 SET UP GET MESSAGE OPTIONS
         SPACE 1
         L     R00,=A(MQWI_UNLIMITED)  UNLIMITED WAIT
         ST    R00,MQGMO_WAITINTERVAL  SET WAIT INTERVAL
         SPACE 1
         LA    R00,USERECB2        POINT TO SIGNAL EVCB
         ST    R00,MQGMO_SIGNAL1   SET POINTER TO SIGNAL EVCB
         SPACE 1
         CALL  MQGET,                                                  X
               (MQHCONN,MQHOBJ,MQMD,MQGMO,BUFFER2L,BUFFER2,DATAL2,     X
               COMPCODE,REASON),                                       X
               VL,MF=(E,EBX008)
```

Figure 25. Example of an MQAWAIT call -- assembler (Part 1 of 2)

```
SPACE 1
         CLC   =A(MQCC_OK),COMPCODE  WAS IT SUCCESSFUL
         BE    MQM0200               YES - BRANCH
         SPACE 1
         CLC   =A(MQRC_SIGNAL_REQUEST_ACCEPTED),REASON OK TO CONTINUE
         BE    MQM0200                                 YES - BRANCH
         SPACE 1
         CLC   =A(MQRC_TRUNCATED_MSG_ACCEPTED),REASON  OK TO CONTINUE
         BE    MQM0200                                 YES - BRANCH
         SPACE 1
         B     MQM0ERR2            OTHERWISE TAKE ERROR BRANCH
         EJECT ,

MQM0200  DC    0H'0'
         CALL  MQAWAIT             WAIT UNTIL SIGNAL(S) RECEIVED
         SPACE 1
         TM    USERECB1,X'40'      HAS FIRST MQGET COMPLETED
         BZ    MQM0200             NO - BRANCH TO WAIT AGAIN
         SPACE 1
         TM    USERECB2,X'40'      HAS SECOND MQGET COMPLETED
         BZ    MQM0200             NO - BRANCH TO WAIT AGAIN
         SPACE 1
         ...
******** Test completion code in bits 20 through 31 of fields
******** USERECB1 and USERECB2
         ...
```

Figure 26. Example of an MQAWAIT call -- assembler (Part 2 of 2)

Closing MQI objects

You should explicitly close any objects that you opened in an application, using the MQCLOSE call. However, if you do not, ALCS closes any objects that are still open for the entry when an application program: exits normally, issues a dump with exit, or terminates abnormally.

Optimized Local Adapters (OLA) for WebSphere Application Server

Built-in, high-speed, bi-directional adapters for calling between WebSphere Application Server and ALCS in another address space on the same z/OS image allow ALCS customers to support an efficient integration of newer Java-based applications with ALCS-based applications.

A set of callable services can be used by ALCS assembler or C/C++ programs for exchanging data with applications running in WebSphere Application Server for z/OS . For more information on the callable services (with names of the form BBOA1xxx) see the IBM Information Center for WebSphere Application Server - Network Deployment (z/OS) and search for BBOA1. You can use the USRWAS1 installation-wide monitor to verify the authority of the caller and to identify input and output mesages.

WAS Bridge

The ALCS WAS Bridge allows ALCS application programs to send and receive messages using optimized local adapters for WebSphere Application Server for z/OS without the need to code those callable services in ALCS programs. The ALCS WAS Bridge installation-wide monitor exits USRWAS3, USRWAS4, USRWAS5, and USRWAS6 allow you to customize the behavior of the WAS Bridge to suit your applications.

ALCS allows application programs to send and receive messages up to 8 megabytes in size using the large message facility. The large message facility is available for messages to and from ALC terminals connected through WebSphere Application Server for z/OS, with the following restrictions:

- ALCS scroll logging is not allowed.
- ALCS message retrieval is not allowed.

Application programs can send large output messages using the SENDC X or ROUTC (`routc`) service. The message must be in heap storage which the program has obtained using MALOC, CALOC, and RALOC monitor-request macros, or `malloc`, `calloc`, and `realloc` C functions.

ALCS passes large input messages to application programs in heap storage which the program must release using the FREEC monitor-request macro or `free` C function.

For details of the AMSG and OMSG message formats and the RCPL contents see Appendix D, “Messages and message formats,” on page 215.

ZSTAT message counts and data collection statistics

You can use the installation-wide monitor exit USRWAS1 to identify those OLA calls which process input or output messages. This enables ALCS to increment the input and output WAS message counts which are displayed by ZSTAT command.

When data collection is collecting statistics about messages, this also enables ALCS to increment the statistics for WAS input and output messages.

Program preparation

For information about the procedure for preparing an ALCS application program that includes OLA calls, see “Application programming languages” on page 124.

Including OLA calls in application programs

For detailed information about including OLA calls in application programs, and for information about the characteristics of these programs, see the IBM Information Center for WebSphere Application Server - Network Deployment (z/OS) and search for OLA calls.

Note that the register name must be a GRS complex unique name and that ALCS does not support the use of the OLA ASYNC, TRANS, and SEC parameters.

Including OLA calls in assembler programs

Figure 27 on page 29 shows how to code an OLA Register call in an assembler program

```
CALL  BBOA1REG,(Daemon_Name,                                  X
      Node_Name,                                              X
      Server_Name,                                            X
      Register_Name,                                          X
      Minconn,                                                X
      Maxconn,                                                X
      Registerflags,                                          X
      Rc,                                                     X
      Rsn),VL,MF=(E,EBX000)
```

Figure 27. A WAS BBOA1REG call -- assembler

OLA calls in high-level language programs

Similarly, you can include OLA calls at any place in a C language where you can place an executable statement. Figure 28 on page 30 shows how to code an OLA Register call in a C language program.

```
          int  BBOA1REG ();
          ...
          ...
          ...


int alcs_rc = BBOA1REG ( &dgn,        /* daemon group name            */
                         &nodename,   /* node name                    */
                         &servername, /* server name                  */
                         &regname,    /* register name                */
                         &minconn,    /* minimum number of connections */
                         &maxconn,    /* maximum number of connections */
                         &regflags,   /* registration flags           */
                         &rc,         /* return code                  */
                         &rsn         /* reason code                  */
                   );
```

Figure 28. A WAS BBOA1REG call -- C language

Testing the return code

You must test the ALCS return code (alcs_rc). The return code is provided by the OLA callable service to an assembler program in general register 15 (RDB) and to a C program as the integer return value from the function call.

ALCS Return codes are:

00
(Call is executed. You must check the OLA rc and rsn.)

08
WAS is not enabled

12
WAS is not connected

16
Caller is not authorized

20
Not executed as WAS is forcibly disconnected.

Unregistering and releasing connection handles

You should unregister and release connection handles when using OLA calls. Multiple ECBs may use the same register name or connection handle, and therefore ALCS will neither unregister nor release connection handles.

3270 Screen mapping support

Assembler language programs can use MAP3270 macroinstructions to create **map** DSECTs which define input and output fields on a 3270 screen. The MAP3270 macroinstructions specify attributes of the screen (size, and so on) and the start position and length of each field and its initial contents.

Some of these fields are unprotected areas where the user can input data, some are areas on the screen that contain headings, labels, and instructions. If the 3270 screen supports color the different fields can appear in different colors. Some fields (for example, one to contain a password) can be defined so that the input keyed by the user does not display. The program can identify the current contents of each field because each field is associated with a label in a map DSECT.

For more information about 3270 screen mapping, see Appendix E, “3270 Screen mapping support for application programs,” on page 226.

TCP/IP support

ALCS allows application programs to communicate using the Extended Sockets application programming interface provided by Communication Server IP.

Figure 29 on page 31 provides an overview of TCP/IP.

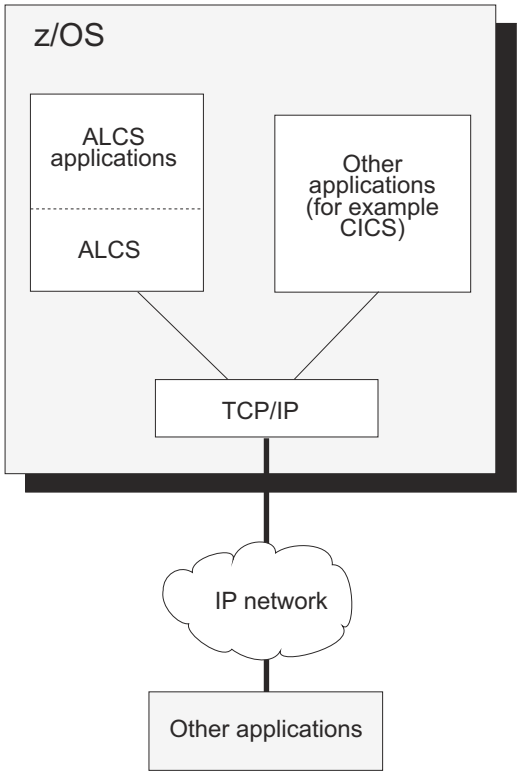


Figure 29. Using TCP/IP to communicate between applications

ZSTAT message counts and data collection statistics

You can use the installation-wide monitor exit USRTCP1 to identify those TCP/IP sockets calls which process input or output messages. This enables ALCS to increment the input and output TCP/IP message counts which are displayed by ZSTAT command. When data collection is collecting statistics about messages, this also enables ALCS to increment the statistics for TCP/IP input and output messages.

Program preparation

Before you begin to use TCP/IP in your application programs see Chapter 7, "Application program management," on page 124, for an overview of other available methods of communication between local and remote applications.

Including TCP/IP sockets calls in application programs

For detailed information about including TCP/IP Sockets calls in application programs and for information about the characteristics of these programs see *z/OS Communications Server IP API Guide*.

You can include TCP/IP Sockets calls at any place in an assembler program where you can place an assembler instruction. Code the TCP/IP Sockets call as a normal MVS CALL.

Figure 30 on page 32 shows how to code a SOCKET call in an assembler program.

```
     CALL  EZASOKET,                                              -
           (SOCKET,           SOCKETS FUNCTION                    -
           SOCKAF,            ADDRESSING FAMILY                   -
           SOCKTYPE,          SOCKET TYPE                         -
           SOCKPROT,          SOCKET PROTOCOL                     -
           SOCKERR,           ERROR NUMBER                        -
           SOCKDESC),         NEW SOCKET DESCRIPTOR               -
           VL,MF=(E,EBX008)
```

Figure 30. Including TCP/IP Sockets calls in an assembler program

ALCS does not support TCP/IP calls in high-level language programs. ALCS does not allow user-written programs to issue GIVESOCKET or TAKESOCKET TCP/IP Sockets calls. These calls are only appropriate for a concurrent server function. If you are designing a server application, you can use the ALCS TCP/IP concurrent server (ALCS Listener) to wait for connection requests to arrive on a port.

ALCS restricts the maximum value of the MAXSOC parameter of the SELECT Sockets call to 256.

By default, ALCS restricts the maximum value of the buffer size parameter to 4000 bytes for some TCP/IP Sockets calls: READ, RECV, RECVFROM. If your installation uses storage blocks which are larger that 4000 bytes, then you can increase the maximum TCP/IP buffer size to match your largest block size. Do this by specifying RCVSZEIP=MAX on the COMGEN macro in your base communication generation load module.

Closing TCP/IP sockets

You should explicitly close down any sockets that you created in an application using the TCP/IP Sockets CLOSE call. However if you do not, ALCS closes down any sockets that are still open for the entry when an application program:

- Exits normally
- Issues a dump with exit
- Terminates abnormally.

TCP/IP large messages

ALCS allows application programs to send and receive messages up to 2 megabytes in size using the **large message** facility. The large message facility is available for messages to and from ALC terminals connected through TCP/IP, and for messages to and from other TCP/IP communication resources, with the following restrictions:

- Unsolicited messages to terminals are not allowed.
- Messages to terminals owned by another system are not allowed.
- ALCS scroll logging is not allowed.
- ALCS message retrieval is not allowed.
- Messages must not exceed any maximum size imposed by the application layer protocol (for example, MATIP or host-to-host).

Application programs can send large output messages using the SENDC X or ROUTC (routc) service. The message must be in heap storage which the program has obtained using MALOC, CALOC, and RALOC monitor-request macros, or malloc, calloc, and realloc C functions.

ALCS passes large input messages to application programs in heap storage which the program must release using the FREEC monitor-request macro or free C function.

The following table summarizes the difference between the application programming interfaces for normal and large output messages.

Normal output messages	Large output messages
Use SENDC A|C|L|M	Use SENDC X

Normal output messages	Large output messages
Use ROUTC (routc) with the RCPLOEXT indicator set off in the RCPL	Use ROUTC (routc) with the RCPLOEXT indicator set on in the RCPL
Message is in a storage block attached on ECB storage level	Message is in heap storage with address in ECB data level
Message is in standard AMSG or OMSG format (2-byte length field)	Message is in extended AMSG or OMSG format (4-byte length field)
ALCS releases the storage block containing the message	ALCS does not release the heap storage containing the message

The following table summaries the difference between the application programming interfaces for normal and large input messages.

Normal input messages	Large input messages
Message is in a storage block attached on ECB storage level	Message is in heap storage with address in ECB data level
Message is in standard AMSG or OMSG format (2-byte length field)	Message is in extended AMSG or OMSG format (4-byte length field)

When defining your ALCS system configuration, the system programmer specifies the number and size of type 3 storage units used for assembler and C/C++ heap storage (see the description of the SCTGEN macro in *ALCS Installation and Customization*).

When defining your TCP/IP connections to ALCS, the system programmer specifies whether or not the TCP/IP connection can support large messages (see the description of the COMDEF macro for a TCP/IP connection in *ALCS Installation and Customization*).

For details of the AMSG and OMSG message formats and the RCPL contents, see Appendix D, “Messages and message formats,” on page 215.

HTTP Client support

The z/OS Web Client Enablement Toolkit can be used to provide HTTP Client services in an ALCS Assembler or C/C++ program.

A set of callable services is provided (HWTHxxxx for assembler and hwthxxxx() for C/C++).

Options can be specified for the connection to the remote HTTP Server such as remote URI, security options, timeout values, etc. The socket calls are handled by USS calls from the HTTP code in the web enablement toolkit. The ALCS application program does not need to make any socket calls for the HTTP client.

Options can also be specified for requests such as redirection, cookies, and specific request headers.

More details can be found in the IBM Knowledge Center or in the MVS Programming: Callable Services for High-Level Languages – SA23-1377.

The GET, DELETE, HEAD, POST, and PUT methods are supported.

One connection can handle different requests (only one at a time). The connections and requests have ECB affinity (assigned on the first HWTHxxxx call - they cannot be transferred to another ECB). An ECB can have more than one HTTP connection and more than one HTTP request active at the same time.

Note that the HTTP client runs in the ALCS address space and will consume storage from the ALCS address space. It might be necessary to increase the storage that is allocated to ALCS.

Program preparation

For information about the procedure for preparing an ALCS application program that includes HTTP Client calls, see “Application programming languages” on page 124.

Including HTTP Client calls in application programs

You can include HTTP Client calls in an assembler program where you can place an assembler instruction.

Including HTTP Client calls in an assembler program

Figure 31 on page 34 shows how to code an HWTHINIT call in an assembler program.

```
CALL HWTHINIT,                                                 X
     (RC,                        Return code                   X
      HANDLE_TYPE,               Handle type                   X
      HANDLE,                    Connect or request handle     X
      DIAGAREA),                 Diagnostic Area               X
      VL,MF=(E,EBX000)
```

Figure 31. An HTTP Client HWTHINIT call – assembler

You can include HTTP Client calls in a C/C++ program where you can place an executable statement.

Including HTTP Client calls in high-level language programs

Figure 32 on page 34 shows how to code an HWTHINIT call in a C/C++ program.

```
include <hwthic.h>          /* HTTP interface declaration file */

HWTHINIT ( &rc,                      Return code                  X
           &handle_type,             Handle type                  X
           &handle,                  Connect or request handle    X
           &diagarea                 Diagnostic Area
         ) ;
```

Figure 32. An HTTP Client HWTHINIT call – C/C++

Releasing Connection and Request Handles

You should explicitly release Connection and Request Handles with the HWTHTERM call. However, If you do not, ALCS releases any handles which are still active when an application program exits normally, issues a dump with exit or terminates abnormally.

JSON Parser support

The z/OS Web Client Enablement Toolkit can be used to provide JSON Parser services in an ALCS Assembler or C/C++ program.

A set of callable services is provided (HWTJxxxx for assembler and hwtjxxxx() for C/C++).

Services provided include validating, searching, creating, adding, and deleting JSON objects.

More details can be found in the IBM Knowledge Center or in the MVS Programming: Callable Services for High-Level Languages – SA23-1377.

A parser instance (work area for the JSON parser) has ECB affinity (assigned on the first HWTJxxxx call). It cannot be transferred to another ECB. An ECB can have more than one parser instance and so parse more than one JSON object at a time.

Note that the JSON parser runs in the ALCS address space and consumes storage from the ALCS address space. It might be necessary to increase the storage that is allocated to ALCS.

Program preparation

For information about the procedure for preparing an ALCS application program that includes JSON Parser calls, see "Application programming languages" on page 124.

Including JSON Parser calls in application programs

You can include JSON parser calls in an assembler program where you can place an assembler instruction.

Including JSON Parser calls in an assembler program

Figure 33 on page 35 shows how to code an HWTJINIT call in an assembler program.

```
CALL HWTJINIT,                                                 X
     (RC,                       Return code                    X
      MAX_SIZE,                 Maximum Work Area Size         X
      HANDLE,                   Parser Instance handle         X
      DIAGAREA),                Diagnostic Area                X
      VL,MF=(E,EBX000)
```

Figure 33. A JSON Parser HWTJINIT call – assembler

You can include JSON Parser calls in a C/C++ program where you can place an executable statement.

Including JSON Parser calls in high-level language programs

Figure 34 on page 35 shows how to code an HWTJINIT call in a C/C++ program.

```
Include <hwtjic.h>         /* HTTP interface declaration file */

HWTJINIT ( &rc,                    Return code                 X
           &max_size,              Max work area size          X
           &handle,                Parser Instance handle      X
           &diagarea               Diagnostic Area
         ) ;
```

Figure 34. A JSON Parser HWTJINIT call – C/C++

Releasing Parser Instance Handles

You should explicitly release Parser Instance Handles with the HWTJTERM call. If you do not, ALCS releases any handles which are still active when an application program exits normally, issues a dump with exit or terminates abnormally.

Chapter 3. Data in ALCS

This chapter describes data that is available to ALCS application programs, including:

- The real-time database
- Sequential files
- Data held on other systems (for example, relational databases).

It also describes how your application program can access and update such data.

Using direct access files

The ALCS real-time database consists of records on direct access storage devices (DASD). The three types of record are:

- Fixed-file records
- Pool-file records
- General file records and general data set records.

This section provides an overview of ALCS DASD support. It describes file addresses within ALCS (fixed file, pool file, and general file) and explains the record hold facility.

File addresses

Each record on DASD is identified by a 4-byte **file address**.

TPF compatibility:

Applications that require compatibility with TPF can use an 8-byte file address, by using a DECB data level. ALCS applications that use a DECB data level must use an 8-byte file address in 4x4 format. You can obtain an 8-byte file address by specifying a DECB address when you request a service to get a file address (see below), or by requesting the FA4X4C (`tpf_fa4x4c`) service to convert an existing 4-byte file address to an 8-byte file address in 4x4 format. For more information about the 8-byte file address in a DECB see “8-byte file address support” on page 91.

When an application program requests a DASD I/O operation, it must tell ALCS the file address of the record it wants to read or write. Application programs can get a file address in one of the following ways:

- For fixed-file records, request the FACE (`face`) or FACS (`facs`) service to obtain a 4-byte file address. Request the FAC8C (`tpf_fac8c`) service to obtain an 8-byte file address in 4x4 format. You provide the fixed file record type and ordinal number of the record you want.
- When writing a pool-file record, request the GETFC (`getfc`) service specifying an ECB level to get the 4-byte file address of an unused pool file record. Request the GETFC (`getfc`) service specifying a DECB address to get the 8-byte file address in 4x4 format of an unused pool file record. When reading a pool-file record that is chained from another record, get the file address from the "refer-from" record (see “Chains of pool-file records” on page 9).
- For general files, request the RAISA (`raisa`) service to calculate the 4-byte file address.
- For general data sets, request the GDSNC (`gdsnc`) and GDSRC (`gdsrc`) services specifying an ECB data level to calculate the 4-byte file address. Request the GDSNC (`gdsnc`) and GDSRC (`gdsrc`) services specifying a DECB address to calculate the 8-byte file address in 4x4 format.

Standard record header

ALCS has a standard format for the first 16 bytes of all records on fixed files, pool files, and general files. This is shown in Figure 35 on page 37.

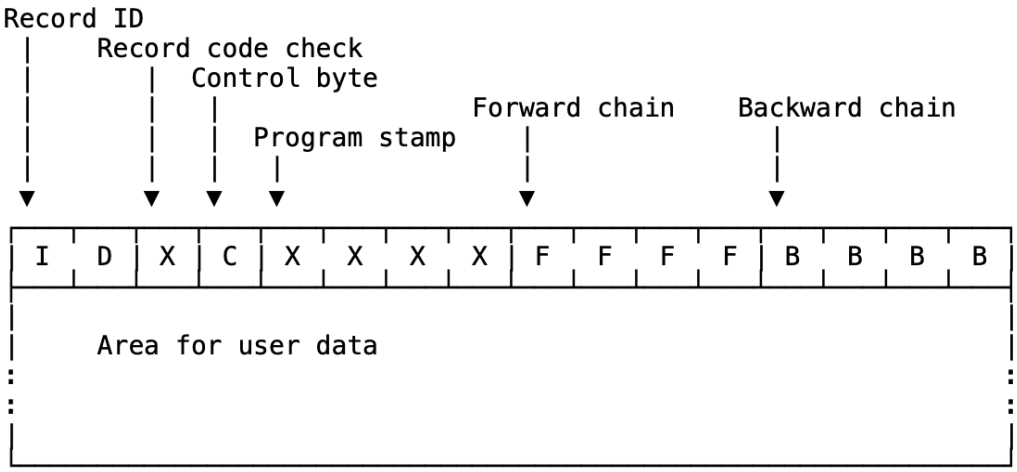


Figure 35. Standard record header

The standard header contains the following fields:

Record ID
This 2-byte field contains the record identifier (record ID). Conventionally, the record ID indicates the kind of data in the record. For example, airlines reservations applications keep passenger name records in long-term pool-file records with an ID of 'PR'.

When application programs read records, ALCS optionally checks that the ID is the same as that which the application program expects. See "Reading and writing DASD records" on page 38. When ALCS dispenses a pool-file record to an application program, it uses the record ID to select the appropriate type of record - that is, a long-term or short-term pool-file record of a particular size (L1, L2, and so on).

Some systems implement logging (using an installation-wide exit) on the basis of the record ID. See *ALCS Installation and Customization* for details of installation-wide exits.

Note: Up to 10 record types can share the same record ID. To distinguish between them, ALCS allows a **record ID qualifier**: a number between 0 and 9.

Record code check
This 1-byte field contains the record code check (RCC). The RCC is intended as an aid in detecting incorrect chaining of records with the same record ID (particularly passenger name records, which can number millions).

A mismatch shows that the chain is broken, probably as a result of an application program releasing a record too soon. A false match cannot be excluded, but the RCC should give early indication of a chaining problem.

Control byte
ALCS ignores this 1-byte field. (There was a standard use for this field in IPARS, but this standard is now obsolete.)

Program stamp
ALCS stores in this 4-byte field the name of the application program that writes the record.

Forward chain
If the file uses standard forward chaining (see "Chains of pool-file records" on page 9), this 4-byte field contains the file address of the next record in the chain. In the last (or only) record in a chain this field contains binary zeros.

Backward chain
If the file uses standard backward chaining (see “Chains of pool-file records” on page 9) this 4-byte field contains the file address of the previous record in the chain. In the first (or only) record in a chain this field contains the file address of the last record in the chain.

In assembler programs, use the STDHD DSECT to reference the standard record header. See the *ALCS Application Programming Reference - Assembler* for details.

In C programs, use the header file <c$stdhd.h> to reference the standard record header. See the *ALCS Application Programming Reference - C Language* for details.

The following sample structure shows the standard record header as it could be defined in a C language program. See *ALCS Application Programming Reference - C Language* for more details.

```
struct sample_record_header
{
  /*
  **   Example of a standard header area
  */
  char              record_ID[2];      /*  Record ID                   */
  unsigned char     rcc;               /*  Record code check           */
  unsigned char     control_byte;      /*  Control byte                */
  char              program_stamp[4];  /*  Program stamp               */
  long int          forward_chain;     /*  Forward chain file address  */
  long int          backward_chain;    /*  Backward chain file address */
};
```

Note: Records used with the IBM database facility (TPFDF) program product use an extended record header. TPFDF references the STDHD DSECT macro and <c$stdhd.h> C header file which describe an extended header containing 4-byte file addresses. TPFDF may also reference the ISTD8 DSECT macro and <c$std8.h> C header file which describe an extended record header containing 8-byte file address fields for compatibility with TPF.

Reading and writing DASD records

An ECB contains 16 **data levels** and 16 corresponding **storage levels**. Each level is identified by a defined symbol: D0 for level 0, D1 for level 1, through to DF for level 15.

An ECB may also dynamically create one or more DECBs, each of which contains a data level and corresponding storage level.

An application uses a data level and a corresponding storage level whenever it reads or writes a DASD record. “ECB data levels and storage levels” on page 78 describes the data and storage level fields in more detail but their use is described here.

Each data level contains the following information about a DASD record:

- 2-byte record ID
- 1-byte record code check (RCC) character
- 4-byte file address (ECB level) or 8-byte file address in 4x4 format (DECB level).

When an application has read a DASD record, or when it is preparing a record for writing to DASD, it holds the record image in a storage block which is attached to the ECB or a DECB (see “Storage blocks” on page 85).

Information about an attached storage block is held in a storage level, which can contain the following information:

- 4-byte storage block address
- 2-byte storage block size field (length in bytes of the user area of the block)
- 2-byte storage block size indicator (L1, L2, ..., L8).

Reading DASD records

When an application program requests an ALCS service to read a DASD record, it specifies the file address of the record it wants to read in any data level which is free (either from the 16 in the ECB (D0 through DF) or from a DECB). (An application can get a DASD file address in various ways, see “File addresses” on page 36.)

The application can optionally also specify the record ID, or the record ID and the RCC value of the DASD record. If the application does not require ALCS to check these fields, it sets them to binary zeros. See Figure 36 on page 39.

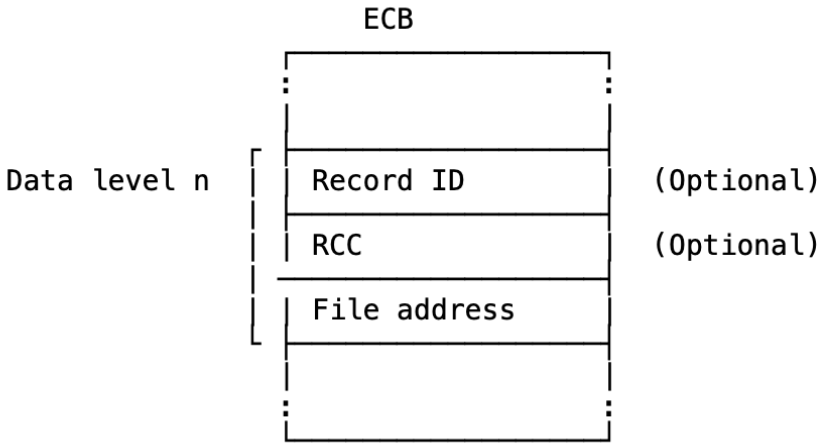


Figure 36. Contents of an ECB data level before a DASD read request

Note: There is a reserved 1-byte field (not shown in the figure) between the RCC and file address fields.

When an application program requests a read service, for example, FINDC (`findc`), it specifies the data level (D0 through DF or the DECB address) that contains the information.

ALCS attempts to read the specified record. If successful, it gets a storage block of the appropriate size (L1, L2, ..., L8) and reads the DASD record into this block. It attaches this storage block to the ECB or DECB and loads the address of the attached storage block, the storage block size, and the storage block size indicator (L1, L2, ..., L8) into the corresponding storage level fields. See Figure 37 on page 40.

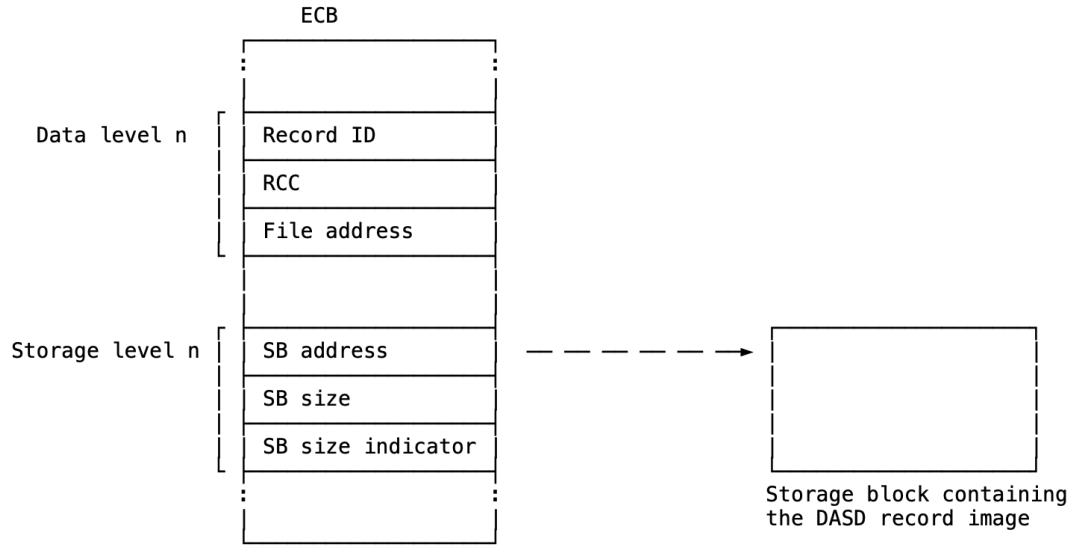


Figure 37. ECB data level and storage level after a successful DASD read

If ALCS cannot read the required DASD record, or if either of the optional fields (record ID and RCC) do not match those in the DASD record, ALCS reports an error. See "Error indicators in the ECB and DECB" on page 115 for a description of how an application tests for these errors.

Writing DASD records

Before an application requests an ALCS service to write a DASD record, it must store the record image in a storage block attached to the ECB or a DECB. (It can get an empty storage block, and attach it at a specified level using a GETCC (`getcc`) service, as described in "Storage blocks" on page 85.)

The GETCC (`getcc`) service puts the address of the storage block, the storage block size, and the storage block size indicator in the storage level fields (see Figure 38 on page 40). The application must also specify the file address and the record ID in the corresponding data level. It can optionally specify the RCC.

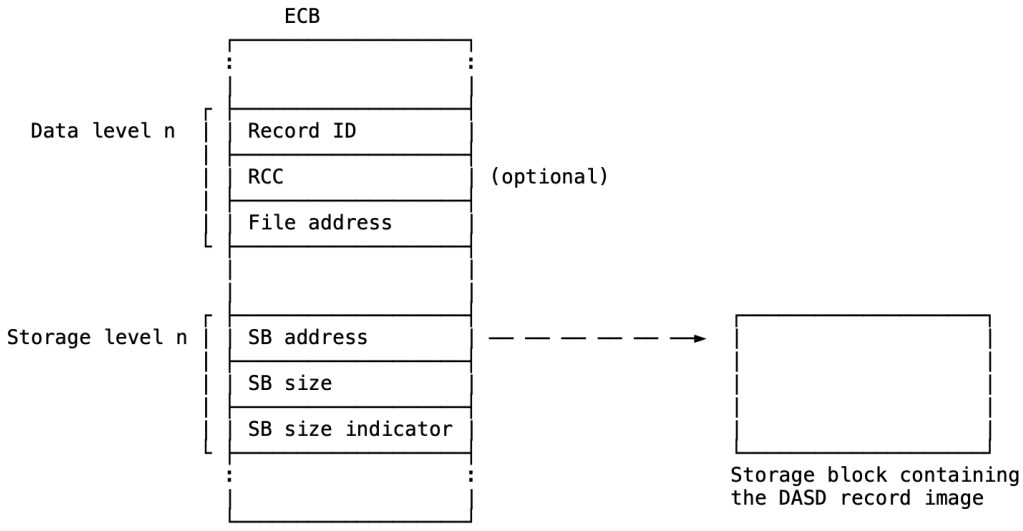


Figure 38. ECB data level and storage level before a DASD write request

When an application program requests a write service, for example, FILEC (`filec`), it specifies the storage level and data level (D0 through DF or the DECB address) that contains the information.

If the file address is not valid, or if the Record ID or (optional) RCC field does not match the corresponding fields in the DASD image in the storage block, this is an error. With most write services, ALCS does not return to the entry, so the application cannot test this. However one write service, FILNC (`filnc`) returns error information which the application can test, see “Error indicators in the ECB and DECB” on page 115.

Sequential files

Some sequential file services also use an ECB data level. The TOUTC (`toutc`) service requires the application to store in the ECB data level the storage address and data length for the write. Also, TDSPC (`tdspc`) can return status information in an ECB data level. See “The ALCS wait mechanism and error processing” on page 111.

Serialization

ALCS is a multiprogramming and multiprocessing transaction processing system. (Systems like this are also called parallel processing or multi-user systems.) This means that there can be many entries accessing the data base at the same time. This requires a serialization process to ensure consistent database updates.

Use of locks

Consider, as an example, two entries which each need to update the same DASD record. Because the two entries can be processing at the same time, one of the updates can be lost, see Figure 39 on page 41.

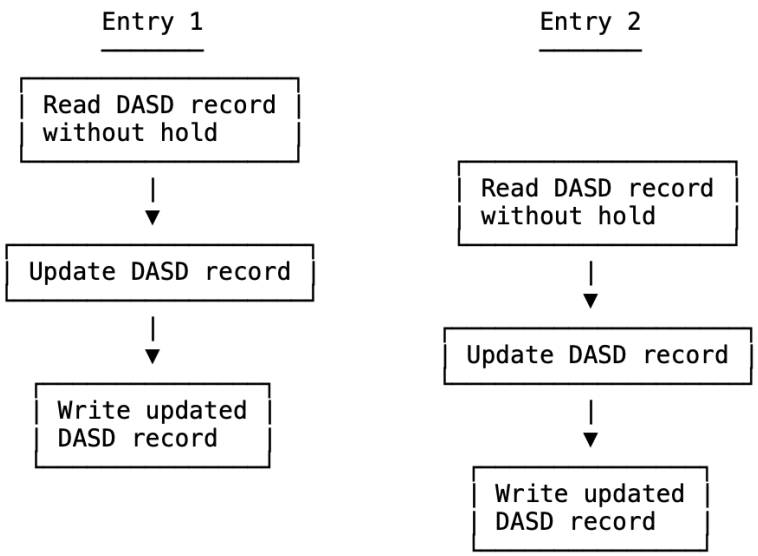


Figure 39. Reading DASD records without holding the file address - updates performed by either entry can be lost.

After such a sequence, the update of entry 1 would be lost.

To prevent this, the program must serialize the updates using a lock, The previous sequence becomes that shown in Figure 40 on page 42.

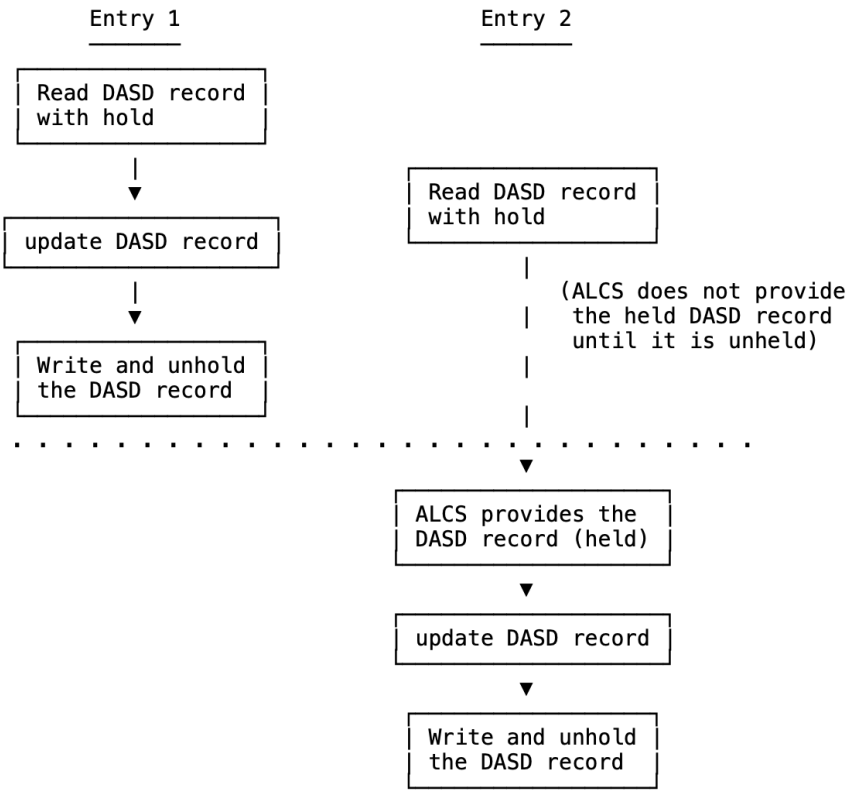


Figure 40. Reading DASD records and holding the file address - updates performed by either entry are recorded.

ALCS suspends entry 2 and does not allow it to read the record and hold the file address until entry 1 has unheld the file address.

To lock a file address, the application program reads the record by requesting a service that has a built-in "hold" option, for example, the FINHC (finhc) or FIWHC (fiwhc) service. When either of these requests completes, the entry is holding the file address. The entry continues to hold the file address until it unholds it with a FILUC (filuc) or UNFRC (unfrc) request.

Serialization is often more complex than this simple example. A single entry often needs to update several records - perhaps deleting information from one record and storing it in another. For example, an airline transaction to cancel a flight might need to transfer the passengers with reservations on that flight to one or more other flights.

Consistent use of locks

In the example just discussed, two entries 1 and 2 ensure that they update an index correctly by using a lock. But for locks to work, it is essential that:

- *All* programs that update a record must request the lock.

  In our example, it is not enough for entry 1 to request the lock. If entry 2 does not request the lock then it is still possible for entry 1's update to be lost.

  In our simple example this mistake is unlikely, but in a real application there may be hundreds of programs that update a particular type of record - if even one program fails to request the lock then updates can be lost.
- All programs that update a record must request the *same* lock.

  In our example, if entry 1 requests lock "A" and entry 2 requests lock "B" then there is no serialization and updates can be lost.

Again, this mistake is unlikely in our simple example. But real applications are more complex. For example, in an airline reservation system, one program might request a lock for a particular flight number when adding a passenger, but another program might request a lock for the flight number *and date*.

Lock contention and granularity

When two entries request the same lock, the first one gets the lock and the second must wait. The second entry cannot resume until the first entry releases the lock. If a third entry requests the same lock then it must wait for both the first entry and the second entry.

Multiple requests for the same lock is called **lock contention**. Lock contention slows transaction responses by forcing entries to wait. To get the best performance from your application, you must design your application *and* your database to minimize lock contention.

There are two main factors that affect lock contention:

- The amount of time that your application holds a lock.

  The longer your program holds a lock, the more likely it is that another entry will request the lock while your program is holding it.
- The amount of the database that the lock controls.

  This is called **lock granularity**. Coarse granularity uses a small number of locks, each controls a large proportion of the database. Fine granularity uses a large number of locks, each controls a small proportion of the database.

  The coarser your lock granularity, the more likely it is that updates to different records will require holding the same lock and therefore the more likely it is that two entries will request the lock at the same time.

Serialization - forcing exclusive access to resources

Each ALCS entry has its own ECB and its own attached and detached storage blocks, which all belong exclusively to that entry. Assembler application programs can therefore use the storage without interference from other entries.

However, all ALCS entries can share other resources, in particular:

- Database records
- The application global area
- Sequential files.

In order to allow application programs to share these resources without corrupting their contents, ALCS allows programs to force exclusive access to a resource while they are using it, as follows:

- Force exclusive access to the resource
- Perform instructions that use the resource
- Release the resource so that other entries can use it.

While one entry has exclusive access to a resource, ALCS queues any other entries that require access to it until the first entry releases it.

Application programmers should therefore keep exclusive access to a resource for the shortest possible time. Otherwise, entries will be kept waiting unnecessarily, which directly affects transaction response times.

For details of how application programs get and release exclusive access of resources, see “Serializing access to the application global area” on page 43.

Serializing access to the application global area

ALCS supports an area of storage that entries can share. This is called the **application global area**[2].

It is complex to use the application global area in a multiprocessor environment for anything except constants. It is much simpler to store variable data in DASD records (appropriate use of virtual file access (VFA) makes physical I/O unnecessary).

However, if you cannot avoid using the application global area, this section describes how assembler application programs can update the application global area in a multiprogramming and multiprocessing environment.

C language application programs:

Application programs written in the C language use their own global functions. These are described in "SYNCC monitor-request macro and global C function" on page 49.

To avoid potential errors, application programs can update the application global area by using the following:

- Block-concurrent references
- COMPARE AND SWAP (CS and CDS) instructions
- ALCS resource hold
- ALCS BEGIN macro SHR and XCL parameters
- SYNCC monitor-request macro. See "SYNCC monitor-request macro and global C function" on page 49.

TPF compatibility:

IBM advises that you use only the SYNCC macro for global serialization in order to minimize changes required to migrate from or to a loosely coupled environment.

When two entries try to update the same global area field at the same time, then the result can be unpredictable. For example, an application program may contain a sequence of instructions such as:

```
L       r1,global_field
LA      r1,1(,r1)
ST      r1,global_field
```

On a multiprocessor, two entries can execute the same sequence of instructions on two different processors. The result, however, is unpredictable:

- If the instructions are executed at exactly the same moment, both entries load the same value into *r1*, increase the value by 1, and store it in *global_field*. The result is that the value in *global_field* increases by 1.
- If the instructions are executed at different times, one entry loads the original value and updates it. Then the other entry loads the updated value and updates it again. The result is that the value in *global_field* increases by 2.

Although the first case is extremely rare, you must write your programs to take account of this possibility.

In general, any sequence of instructions similar to:

1. Load global area field
2. Compute new value
3. Store back new value

can produce unpredictable results. Some single instructions, for example OR (OC and OI), can produce unpredictable results in the same way.

A comparable error can happen if one entry updates a global area field (store accesses) and another entry reads (fetch accesses) the field at exactly the same time. For example, one entry can move data to a global area field. The data replaces the previous field contents. Another entry can move data from the

[2] Actually the ALCS application global area is three areas of storage which can be unprotected (any entry can store information in the global area) or protected (entries must set the protect key to allow access). For details about using global area protection in ALCS, see "Global area protection" on page 62.

same field (for example, into the ECB work area). The two moves can overlap so that the second entry moves a mixture of the original field contents and the new contents.

Note that both types of error can occur on uniprocessors. If an entry loses control during the update, then another entry can refer to or update the same field. Consider this example:

```
L       r1,global_field
DEFRC   ,
LA      r1,1(,r1)
ST      r1,global_field
```

When one entry issues the DEFRC, another entry can update the value in *global_field*. In the same way, an entry that issues DEFRC between two related updates to the global area allows other entries to "see" inconsistent data.

Treat individual fields **that are independent of other fields** according to their size:

- For appropriately aligned fields up to 8 bytes, use either of:
  - Block-concurrent instructions (“S/390 block-concurrent references” on page 45)
  - COMPARE AND SWAP and COMPARE DOUBLE AND SWAP (“S/390 compare and swap instructions” on page 46).
- For other fields, use either
  - The SHR and XCL parameters (“ALCS BEGIN macro SHR and XCL parameters” on page 47). The global area field name can be a suitable token, or
  - The resource hold facility (“ALCS resource hold” on page 46).

S/390 block-concurrent references

Application programs can safely share information in the application global area by means of block-concurrent instructions, provided that:

- For update, the new value does not depend on the old value (the program does not use the old contents of the field to calculate the new contents), and
- The field does not need to be consistent with other fields.

In these circumstances, block-concurrent instructions can be the most efficient way to share access, as they do not require any extra instructions to serialize accesses.

Some assembler instructions make block-concurrent storage references. The *z/Architecture® Principles of Operation* describes block-concurrent references and lists the instructions that make this type of reference. Many instructions that refer to an operand access the operand in a way that is block-concurrent. For example:

- Byte
- Halfword, aligned
- Fullword, aligned
- Doubleword, aligned.

That is, the access (fetch or store) does not overlap with other accesses to the same field. For example, some programs STORE (ST) into a fullword and other programs LOAD (L) from the fullword.

- STORE does not overlap with STORE. For example, an aligned fullword field contains hexadecimal 00000000 and there are two STOREs into the field. One stores hexadecimal 11111111 and the other stores hexadecimal 22222222. The field can contain hexadecimal 00000000, 11111111, or 22222222. It cannot contain, for example, 11112222. (If the field is not fullword aligned, the STOREs can overlap, and the result can be 11112222.)
- STORE does not overlap with LOAD. For example, an aligned fullword field contains hexadecimal 00000000 and there is a STORE into the field; it stores hexadecimal 11111111. LOAD can load

hexadecimal 00000000 or 11111111. It cannot load, for example, 11111100. (If the field is not fullword aligned, the STORE can overlap, and LOAD can load 11111100.)

Use of block-concurrent instructions

An aligned fullword contains a currency exchange rate. The program that updates the field gets the new exchange rate from an external source - an input message. The program uses STORE (ST) to update the field. All programs that use the field use LOAD to load it. In this case there is no need to serialize the accesses.

There can be several fullwords that contain exchange rates. If so, then block concurrent accesses cannot ensure that all the exchange rates are consistent. Instead, a program that updates the exchange rates must have exclusive control of all the exchange rates while it makes the updates. Programs that use more than one exchange rate must have shared control of the exchange rates. But even in this case, a program that uses only one exchange rate can LOAD it while another program is updating exchange rates.

S/390 compare and swap instructions

If block-concurrent access is not suitable, then COMPARE AND SWAP (CS) or COMPARE DOUBLE AND SWAP (CDS) can be the most efficient way to share access to information in the application global area. It does require extra instructions (CS or CSD) to serialize accesses but it does not prevent parallel execution. Two entries can run at the same time on two different processors, even though they both update the same global area field. COMPARE AND SWAP is **not** suitable for information where several fields must contain consistent information.

The *z/Architecture Principles of Operation* describes the CS and CDS instructions and gives examples of how to use them.

In general, use CS to update a global area field as follows:

1. Load global area field into general register *r1*
2. Use *r1* to compute the new value in general register *r2*
3. Use COMPARE AND SWAP to store back the new value
4. If condition code 4, then repeat from step “2” on page 46.

For example, to add 1 to the contents of *global_field*:

```
         L      r1,global_field
RETRY    EQU    *
         LA     r2,1(,r1)
         CS     r1,r2,global_field
         BNE    RETRY
```

Note: A field can contain subfields that application programs access independently. For example, an application program can use TEST UNDER MASK (TM) several times to test different bits in the same fullword. To ensure that these tests give consistent results, the program must use block-concurrent accesses to copy the whole fullword (for example, into the ECB work area). The application program can then test its own copy; no other program can change it between tests.

ALCS resource hold

If block-concurrent access and COMPARE AND SWAP are not suitable, then ALCS application programs can use resource hold to serialize access to information in the application global area.

To use resource hold in this way, define a value (for example, a storage address) that uniquely identifies a field or a group of fields in the application global area. Every application program that uses the field or fields must ensure exclusive control. To do this, application programs can use the resource hold monitor-request macros. These are:

- SYNCC LOCK and SYNCC UNLOCK (recommended)
- CORHC and CORUC

- ENQC and DEQC.

The *ALCS Application Programming Reference - Assembler* describes these monitor-request macros.

Note that the resource hold monitor-request macros allow an entry to keep exclusive access to global area information, even if the entry loses control, for example, when the entry waits for DASD I/O completion.

ALCS BEGIN macro SHR and XCL parameters

TPF compatibility:

This facility is not available in TPF.

If block-concurrent access and COMPARE AND SWAP are not suitable, ALCS application programs can use the BEGIN macro parameters SHR and XCL to serialize access to information in the application global area. Shared access (SHR) is equivalent to read-only access; exclusive access (XCL) is equivalent to read and write access. The format of these parameters is:

```
SHR=(token,...)
XCL=(token,...)
```

where *token* is a user-defined character string that identifies a field or a group of fields in the application global area. In this way the application program indicates that it requires shared or exclusive access to one or more groups of fields.

The token ALL means all fields in the application global area, NONE means no fields. (The next sections describe how your system programmer can set up your tokens.)

The defaults are SHR=ALL,XCL=ALL. The program gets exclusive control of all global area fields. On a multiprocessor, this default means that the program can only run parallel with programs that specify SHR=NONE,XCL=NONE. Many programs do not need exclusive control of all fields in the global area. For these programs, use the SHR and XCL parameters to reset the defaults.

For optimum performance, even on a uniprocessor, you should specify SHR=NONE,XCL=NONE. Do this when the application program:

- Does not refer to any global area fields
- Only refers to global area fields that are constants
- Can share access to all the global area fields that it references (for example, because it uses block-concurrent accesses, or because application programs use COMPARE AND SWAP to update the fields)
- Uses resource hold to control all accesses to the global area.

If the application program requires shared access to some global area fields, but it does not require exclusive access, then specify XCL=NONE. (If possible, also avoid SHR=ALL, see below.) Do this if the application program:

- Does not modify any global area fields
- Can share access to all the global area fields that it modifies (for example, because it uses block-concurrent stores or COMPARE AND SWAP)
- Uses resource hold to control accesses to the global area fields that it modifies.

If the application program requires shared or exclusive access to some but not all of the global area, then do not specify or default to SHR=ALL or XCL=ALL. Instead, specify tokens that identify only those fields the application program references. The following sections describe how you can set up your own tokens.

Note: The BEGIN macro parameters do not allow an entry to keep exclusive or shared control of global area information if the entry loses control. For example, if the entry waits for DASD I/O completion, it loses shared (or exclusive) control while it is waiting. It gets shared (or exclusive) control again when

the wait completes. To keep control (when the entry would otherwise lose control) use the resource hold facility. See “ALCS resource hold” on page 46.

How to allocate BEGIN macro SHR and XCL parameters

The BEGIN macro's SHR and XCL parameters use tokens that are user-defined character strings. Each token identifies a field or a group of fields in the application global area. If required:

- A token can identify a single field.
- More than one token can identify the same group of fields.
- The group of fields that one token identifies can overlap with the group that another token identifies.
- A token can identify some or all of the fields that another token identifies.

You can allocate tokens in the following ways:

- Identify major subsets of the application programs. Each subset is the only (or main) user of a particular subset of global area fields. Allocate a token to each subset of global area fields.

  This can make it easy to modify existing application programs (add SHR and XCL parameters to the BEGIN statement) so that they can exploit multiprocessors.

  For example, an airline application might include the following subsets:

  Passenger seat reservations
  Fare quotation
  Check-in.

  You could allocate a token RES to identify fields in the global area that passenger seat reservations programs use. To identify fields used by fare quotation programs you could allocate a token FQUOTE.
- Identify groups of logically related global area fields. The fields are related because they must contain consistent information. Often a single application program is reponsible for updating all the fields in a group. Allocate a token to each group of logically related fields.

  For example, the global area might contain a number of fields that contain date and time information. The token TIME can identify this group of fields. One application program can be responsible for updating all of these fields (for example, every minute). That program requires exclusive access to the group. Other programs that access some or all of the fields in the group need shared access to the group.

How to define BEGIN macro SHR and XCL parameters

To define tokens for the BEGIN macro's SHR and XCL parameters, modify the ALCS global access serialization macro, DXCSER. The DXCSER macro is described in the *ALCS Installation and Customization*.

DXCSER associates each token with one or more resources. ALCS implements this with two fullwords. One fullword indicates shared control requirements. When a bit is on, the application program requires shared control of the corresponding resource. Similarly, the other fullword indicates exclusive control requirements.

In this way, an application program can specify exclusive control, shared control, or no control of each resource independently.

There can be up to 32 resources. But there can be many more tokens. Allocate resources to tokens as follows:

- If a token represents a group of fields that must contain consistent information, allocate one resource to it.
- If a token represents a major subset of the global area, allocate several resources to it. This allows other tokens to refer to groups of fields within the major subset.
- If more than one token refers to the same group of fields, allocate the same resource to all the tokens.

It can be necessary to "lump" groups of fields (that is, allocate a single resource to more than one group of fields).

SYNCC monitor-request macro and `global` C function

To be compatible with loosely coupled systems, ALCS application programs can use the SYNCC monitor-request macro to serialize access to the application global area. C application programs can **only** use the `global` C function - which provides essentially the same services as SYNCC.

In a loosely coupled configuration (or in a TPF tightly-coupled configuration) there are multiple copies of the application global area. SYNCC and `global` provide services to ensure that updates are correctly and consistently applied to all copies. These services also serialize global area updates in tightly-coupled and uniprocessor configurations. They are:

LOCK
This service gives the requesting entry exclusive control of the specified global field. Entries that request this service are forced to run serially even if they execute in different CECs of a loosely coupled configuration.

(In a TPF loosely coupled configuration, this service also ensures that any updates to the field - including updates done on other CECs in the complex - complete before returning to the requesting entry.)

After requesting the LOCK service, the entry can update the locked field.

SYNC
After updating the global area field, the entry **must** request this service to:

1. Inform ALCS that the field has changed. If the record containing the field is keypointable then ALCS updates the file copy of the record.

   (In a TPF loosely coupled configuration, this service starts the process of updating the field in other CECs of the complex.)
2. Unlock the global field. This allows the next entry (if any) that is waiting to access the field to proceed.

An entry can use the LOCK service of SYNCC or `global` even when the entry does not update the global area field. For example, an entry can use the LOCK service to prevent any other entry from updating a field during some process. An entry may also use the LOCK service to be sure that it accesses absolutely up to date field contents even in a loosely coupled configuration.

If an entry uses the LOCK service without updating the global field, it can not use the SYNC service to unlock the field. Therefore, SYNCC and `global` provide the following additional service:

UNLOCK
This service unlocks the global field. This allows the next entry (if any) that is waiting to access the field to proceed.

ALCS services for accessing DASD records

Application programs can use the following assembler macros and C functions to access DASD records.

Type of DASD record	Assembler macro	function	Used to
Fixed file	FACE (note “1” on page 51)	`face`	Calculate a fixed file address from fixed-file record type number and record ordinal
	FACS (note “1” on page 51)	`facs`	Calculate a fixed file address from fixed-file record type name and record ordinal

Type of DASD record	Assembler macro	function	Used to
	FAC8C	tpf_fac8c	Calculate an 8-byte fixed file address in 4x4 format from fixed-file record type name or number and record ordinal
Pool file	GETFC (note “2” on page 51)	getfc getfc_alt	Get a pool file address (dispensed by ALCS) and optionally get and attach a storage block
	RCRFC	tpf_rcrfc	Release a pool file address and detach and release the storage block
	RELFC	relfc	Release a pool file address, and optionally detach and release the storage block
	RLCHA	rlcha	Release a chain of pool-file record addresses
General file	RAISA	raisa	Calculate the file address of the first record in a general file or increment the relative record number of the current record in a general file and calculate its file address
General data set	GDSNC	gdsnc	Open or close a general data set (GDSNC OPEN also calculates the file address of a general data set record)
	GDSRC	gdsrc	Calculate the file address of a general data set record
All	FILEC	filec filec_ext	File (write) a DASD record
	FILNC	filnc filnc_ext	File (write) a DASD record without detaching the storage block
	FILUC	filuc filuc_ext	File (write) a held DASD record and unhold the file address
	FINDC	findc findc_ext	Find (read) a DASD record
	FINHC	finhc finhc_ext	Find (read) a DASD record and hold the file address
	FINWC	finwc finwc_ext	Find (read) a DASD record
	FIWHC	fiwhc fiwhc_ext	Find (read) a DASD record and hold the file address
	RCUNC	rcunc	Unhold a held DASD record and detach and release the storage block
	RIDIC		Extract information about a record ID
	RONIC	ronic	Extract information about a record
	UNFRC	unfrc unfrc_ext	Unhold a held DASD record
	SONIC	sonic	Extract information about a record

Type of DASD record	Assembler macro	function	Used to
		file_record file_record_ext	File a DASD record
		find_record find_record_ext	Find a DASD record
	FA4X4C	tpf_fa4x4c	Convert a 4-byte file address to an 8-byte file address in 4x4 format, or convert an 8-byte file address in 4x4 format to a 4-byte file address

Notes:

1. *(NOTFAC)* FACE and FACS are not assembler macros but programs that application programs can ENTER.
2. *(NOTGET)* ALCS also provides GETLC, GETSC, GCFLC, and GCFSC macros for compatibility with TPF. IBM recommends that you use GETFC instead of any of these macros.

TPF compatibility:

Do not request the RIDIC or RONIC (ronic) services in programs that must be compatible with TPF.

Summary of ALCS services for DASD processing

Figure 41 on page 52 summarizes the assembler macros (in upper case) and functions (lower case) you can use with records on DASD.

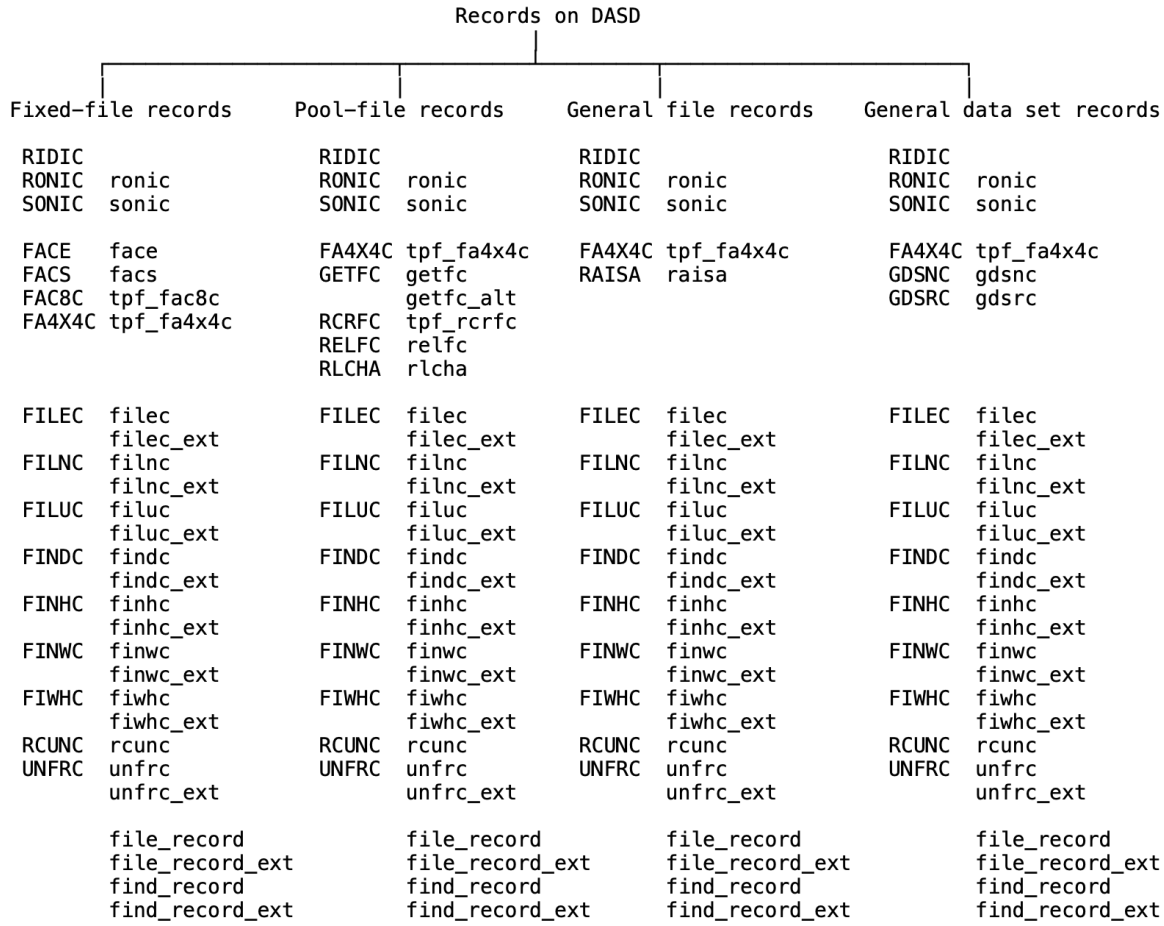


Figure 41. ALCS services for processing DASD records

Offline access to general files and general data sets

Offline programs use standard VSAM facilities to access general files and general data sets. Offline programs must not access the ALCS real-time database.

Using sequential files

In ALCS, sequential files consist of records on any medium that MVS sequential access method (SAM) supports, including DASD. Sequential files are identified by a 3-character name. Records on sequential files can be standard sizes (L0, L1, L2, ..., L8) or nonstandard sizes. C language programs can read and write standard-size records only.

There are three types of sequential file: system files, real-time, and general. Application programs cannot access system sequential files, but can:

- Write records to **real-time sequential files**
- Read records from, or write records to, **general sequential files**.

You can access partitioned data set (PDS) members as sequential files.

System sequential files

Application programs cannot read records from, or write records to, system sequential files. (However, once ALCS has closed a system sequential file, an application could open the same physical file as a general sequential file and read the records from it.)

Real-time sequential files

When ALCS is started, it opens all the real-time sequential files that are available. Any entry can then write data to any of these files.

Real-time sequential files are for output only. The real-time sequential file services TOURC (`tourc`) and TOUTC (`toutc`) allow application programs to write to real-time sequential files.

Note: Because any entry can write data to a real-time sequential file at any time, you cannot normally control the sequence of records on the file. The records that your entry writes may be interspersed with records that other entries write.

General sequential files

Before an application program can use a general sequential file, the entry must first open the file and **assign** it to itself. "Assigned" means that the file is allocated for the use of this entry only.

The TOPNC (`topnc`) service opens and assigns a general sequential file with one call. When it requests this service the program must specify whether the sequential file is to be opened for reading (input) or writing (output).

Following a TOPNC (`topnc`) request to open the file for input, application programs can request the TPRDC (`tprdc`) service to read records from the sequential file. If the file was opened for output, application programs can request the TWRTC (`twrtc`) service to write records to the sequential file. Assembler programs can use the TDTAC service to read and write nonstandard-size records.

When an application has finished using a general sequential file (and before the entry exits), it must unassign the general sequential file. There are two ways to do this. The TCLSC (`tclsc`) service unassigns and closes the general sequential file. The TRSVC (`trsvc`) service unassigns the general sequential file, but it does not close it. Requesting the TRSVC (`trsvc`) service is called **reserving** the sequential file.

Once a general sequential file is reserved, another entry (or the same entry that requested TRSVC (`trsvc`) can assign the file to itself by requesting the TASNC (`tasnc`) service.

Note: The "higher-level" C functions `tape_open`, `tape_read`, and `tape_write` do not follow these conventions. Do not use them if more than one entry can access the general sequential file at the same time.

Several entries using the same general sequential file

The TRSVC (`trsvc`) and TASNC (`tasnc`) services allow several entries to use the same general sequential file, as follows:

1. One entry opens and assigns the file by requesting the TOPNC (`topnc`) service. It reads or writes several records and then reserves the file by requesting the TRSVC (`trsvc`) service.
2. Another entry then assigns the file by requesting the TASNC (`tasnc`) service, reads or writes several records, and then reserves the file again.
3. When these cooperating entries have all finished using the file, one of them assigns the file and then closes it by requesting the TCLSC (`tclsc`) service.

Figure 42 on page 54 shows this process.

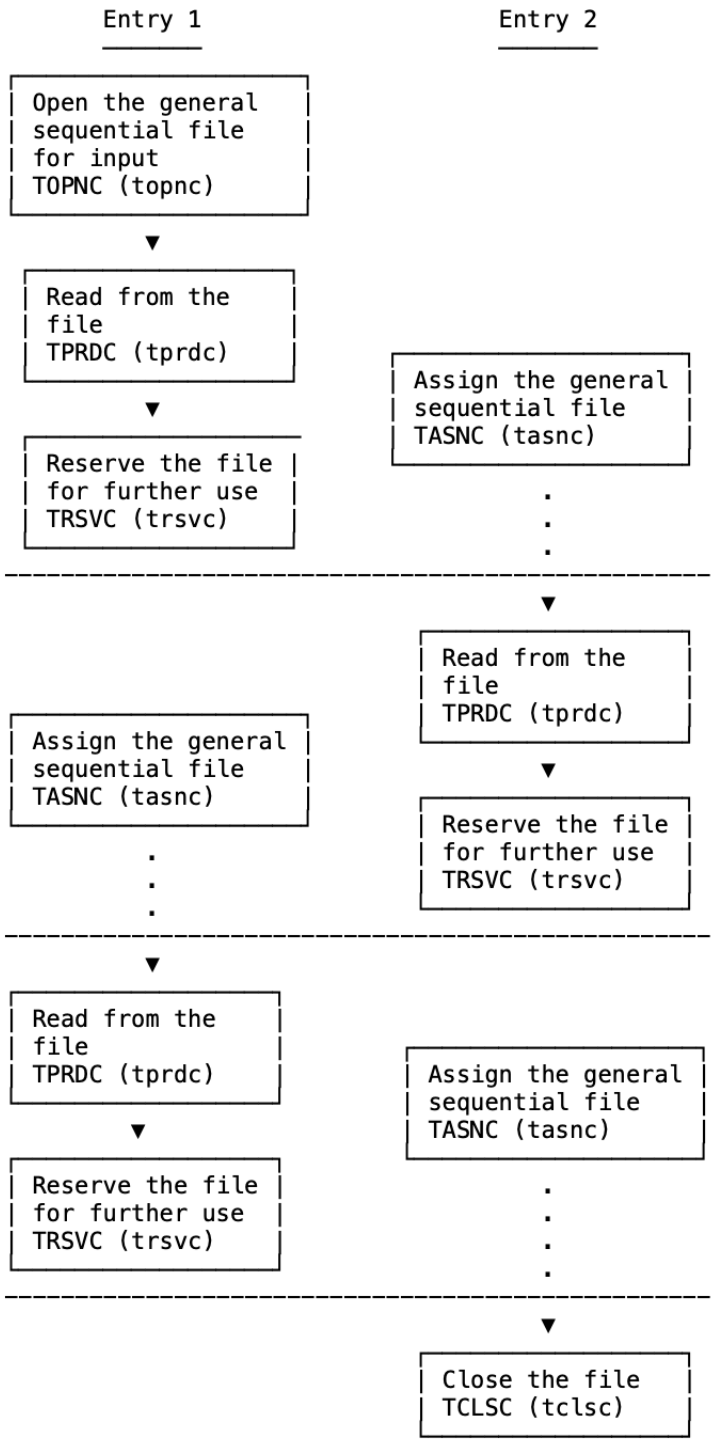


Figure 42. Two entries accessing the same general sequential file

The general sequential file must be open when an application program assigns it by requesting TASNC (`tasnc`) need not be reserved at the time. If the general sequential file is assigned to another entry, the TASNC (`tasnc`) service waits for the other entry to request TRSVC (`trsvc`). If several entries request TASNC (`tasnc`) for a file that is already assigned then they get control in turn; when one entry requests TRSVC (`trsvc`), the next entry gets control, and so on.

ALCS services for sequential file processing

The following table shows the assembler macros and C functions you can use with ALCS sequential files.

Type of sequential file	Assembler macro	Function	Used for
All	TDSPC	tdspc	Extract information about the sequential file
Real-time	TOURC	tourc	Write a standard-size (L0, L1, L2, ...) record
	TOUTC	toutc	Write any size record
General	TASNC	tasnc	Assign a general sequential file to the entry
	TCLSC	tclsc	Close and deallocate, then unassign a file
	TOPNC	topnc	Open and allocate, then assign a file to an entry
	TRSVC	trsvc	Unassign (reserve) a general sequential file
	TPRDC	tprdc	Read a standard-size record (L0, L1, L2, ...)
	TWRTC	twrtc	Write a standard-size record (L0, L1, L2, ...)
	TDTAC		Read or write any size record
		tape_close	Close a general sequential file
		tape_open	Open a general sequential file
		tape_read	Read a record
		tape_write	Write a record

Summary of ALCS services for sequential files

Figure 43 on page 55 summarizes the assembler macros (in upper case) and functions (in lower case) you can use with the three types of sequential file.

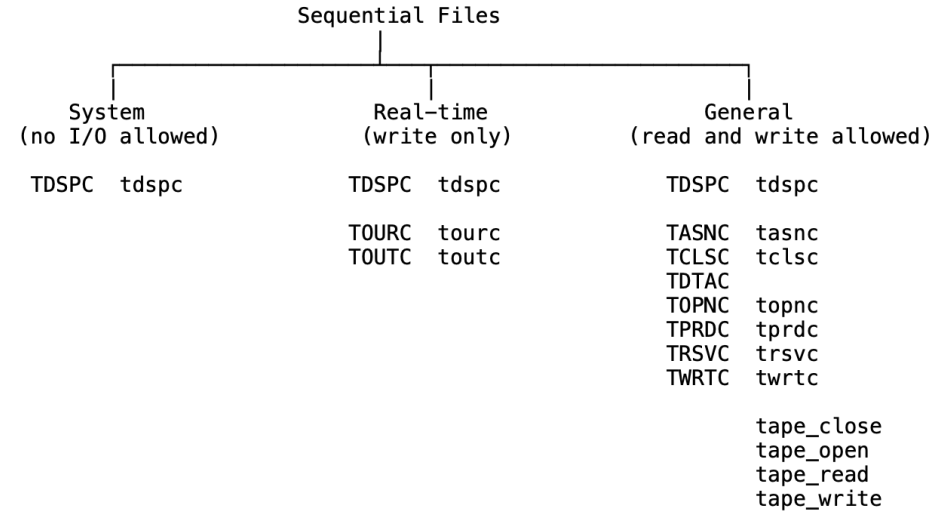


Figure 43. Macros and functions for processing sequential files

Global records and fields

ALCS supports the use of application **global records** and **global fields** (both referred to as "globals"). Global records are fixed-file records that ALCS loads into main storage shortly after start up. Global fields are fields within **directly-addressable** global records (see “Directly-addressable global records” on page 56).

Global records can be read and modified by all application programs.

You are recommended not to use global records (see “Using globals - attention” on page 58). You can get the same benefits in a more controlled way by using VFA. However, some existing applications use global records, so you might need to know how they operate.

By convention, global records and fields are identified in assembler language programs by a name starting with an '@', followed by from 1 to 7 alphabetic or numeric characters (for example, @ABC11). In C language programs the name starts with an underline character (for example, _ABC11). In the following section we use the assembler convention.

Global directories

ALCS holds global records in main storage in 3 **global areas**. The first global area holds up to 16 **global directories**. Each global directory has an identifier @GL*n*BA, where *n* is a hexadecimal number 0 through F. Global directory @GLFBA is reserved for ALCS's use.

Each global directory contains up to 256 directory "slots", each slot contains the storage address of a global record (4 bytes), or the storage address and file address of a global record (8 bytes). Each directory slot is identified by a name of up to 8 characters. Figure 44 on page 56 shows a logical view of a global directory showing how directory entries enable assembler language application programs to locate global records by using directory slots.

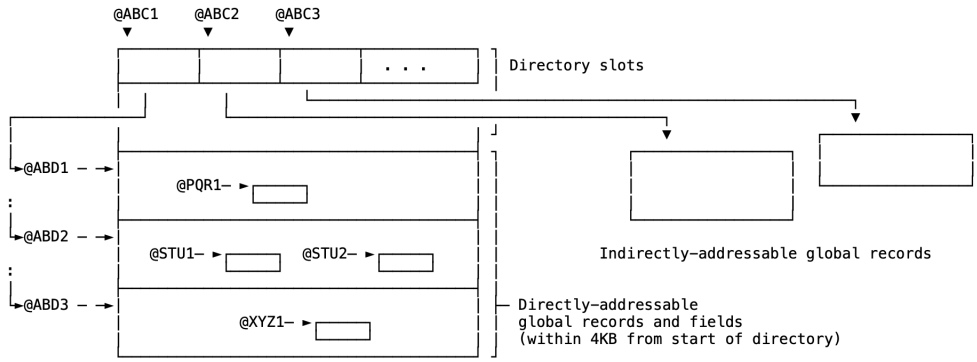


Figure 44. ALCS global area directory - logical view

Directly-addressable global records

As described in “Global directories” on page 56, global area 1 holds up to 16 global directories. Global area 1 can also hold global records, as shown in Figure 44 on page 56. In addition to accessing these records through a directory slot, assembler programs can access fields within these records directly, using the field name (for example @ABD1, @ABD2, @ABD3), so long as they are within 4KB from the start of the directory. These records are called **directly-addressable global records**.

Within directly-addressable global records, application programs can access fields directly using the field name (for example, @PQR1, @STU1, @STU2, @XYZ1). These fields are called **global fields**. They are the only type of "global" that C language programs can update.

Indirectly-addressable global records

A directory can contain storage addresses of up to 256 global records outside the 4KB area. Programs can only access these records indirectly through the directory slot, using the identifier (for example, @ABC1, @ABC2, @ABC3). These global records are called **indirectly-addressable global records**. The program must know the structure of the record to access individual fields in it. (An assembler language program uses a named DSECT, a C language program uses a named `struct`.)

Keypointing global records

Keypointing is the process of writing a record to the database. The system programmer defines individual global records as being **keypointable** or **nonkeypointable**.

Keypointable
: These are global records that are intended to be updated by application programs. After each update, the application program can issue an assembler macro to write the global record to DASD.

Nonkeypointable
: These are global records that are not intended to be be updated by application programs. Application programs can update nonkeypointable records, but even if they request it, ALCS never writes nonkeypointable globals to DASD. They are reinitialized when ALCS restarts.

Note: Updating any global record or field requires care. See “Global area serialization” on page 59.

Logical global records

Physical DASD records include a 16-byte standard header (see “Standard record header” on page 36). ALCS can combine several physical records into one "logical" global record for easier processing by application programs. See Figure 45 on page 58.

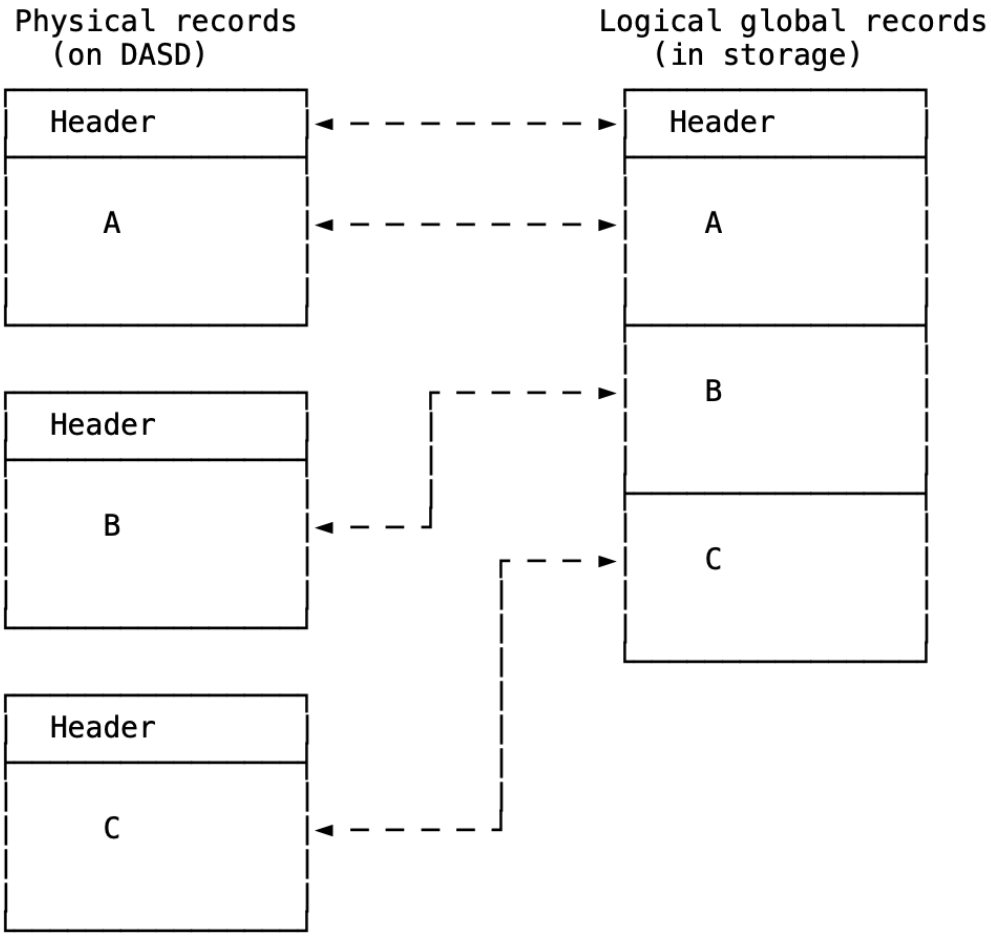


Figure 45. Removing headers to create a logical global record

ALCS can keypoint header-stripped records; this is called **logical global support**.

Using globals - attention

Because global records are permanently in storage, application programs can access the data that they contain without the need for any I/O.

However, if possible, instead of globals you should use fixed-file records defined with the virtual file access (VFA) attribute described in *ALCS Installation and Customization*. In particular, the permanently resident attribute and the time-initiated file attribute can allow application programs to use read and write records with relatively few I/Os.

Using the application global area for records has several disadvantages:

- Applications programs that share access to records in the global area must serialize access to the records. This process can be complex. See "Global area serialization" on page 59.
- It is not easy to add new global records to the global area. (Similarly, it is difficult to remove records from the global area.) By contrast, it is easy to change the VFA options for a record.
- Existing application programs that use read and write macros (or functions) to access a record must be changed to access records from the global area.
- C language programs cannot update global records (they can only update global fields).

- It might become necessary at some stage to run the application programs on a loosely coupled system (for example, under TPF). Application programs that use the global area can require major changes to work in a loosely coupled environment.
- The use of globals greatly reduces the portability of a program.

Global area serialization

The global areas do not belong to any particular entry. They are shared by all entries.

Consider an ECB-controlled program that does the following:

1. Copies a field from a global record (for example, field @ABC1)
2. Updates the copy (for example, adds 1 to it)
3. Stores the updated copy back into the global record.

This sequence works correctly only when no other entry can update the global field @ABC1 during the process. However, with a multiprocessor system this cannot be guaranteed. Even on a single-processor system the first entry could lose control between steps 1 and 3.

When application programs update a global area, access must be **serialized** to avoid data corruption.

There are several methods of serializing access to the global area. The most universally applicable method is for the entry to **hold** the global, using the SYNCC assembler macro or the `global` function during the update. When update is complete the entry releases the hold and other entries can then access the global field or record. See "Serialization" on page 41.

Serialized access inevitably requires extra processing (additional to the update itself), particularly when the application is to run in a loosely coupled environment.

Using resource hold

In assembler programs, the ALCS resource hold facility is controlled by the following assembler macros:

CORHC (used with CORUC) or
ENQC (used with DEQC) or
SYNCC LOCK (used with SYNCC UNLOCK or SYNCC SYNC).

All application programs which access the data must hold (using CORHC, ENQC, or SYNCC LOCK) before accessing the data. After accessing the data, they must unhold (using CORUC, DEQC, SYNCC UNLOCK, or SYNCC SYNC).

Note: Even application programs which only read (fetch access) the data must use resource hold. This ensures that the data does not change while the program is accessing it. (This requirement does not apply to data that **never** gets changed.)

Tightly coupled systems

In a tightly coupled system, SYNCC LOCK holds the referenced field, and SYNCC UNLOCK unholds it. SYNCC SYNC marks the appropriate record as ready for keypointing before unholding the referenced field.

Loosely coupled systems

In a loosely coupled system, SYNCC LOCK holds the referenced field in each memory containing the global area. SYNCC UNLOCK unholds the referenced field in each memory. SYNCC SYNC starts the process of updating the referenced field in other menbers of the complex. It also marks the appropriate record as ready for keypointing before unholding the referenced field.

Note: Current releases of ALCS support tightly coupled systems only.

Other methods

Assembler programs can use other methods to perform global serialization. These include:

1. Block-concurrent referencing

2. Compare and swap instructions
3. Shared access (SHR) and exclusive access (XCL) parameters, together with tokens, on the BEGIN macro.

Methods 1 and 2 only work on certain global fields and only if all other programs access the globals using the same method. They do not work in a loosely coupled environment but some existing applications use them.

TPF compatibility:

Method 3 is not available under TPF.

C functions for processing globals

ALCS provides two functions `glob` and `global` for accessing the global area. These let C language programs read global records or global fields, but C language programs can only update global fields (not records). They are described in detail in *ALCS Application Programming Reference - C Language*.

Accessing global fields

In C language application programs, you can use either the `glob` function or the `global` function to access global fields. You can use the `glob` function to read a global field or record. You can use the `global` function to update global fields (but not global records), to do the necessary serialization for these updates, and perform other actions.

Note: Because all entries share the same global area, you must **not** update variables in the global area directly, using operators such as =, or ++. You must use the ALCS `global` function.

How a C language application program accesses the global area

If you need to read a global record, or read or write a global field, you must first check whether **any** application program modifies the data you want to access. This is important even if your program does not need to modify the data.

You must also check that your system programmer has created the header file <c$globz.h> for your installation. Note that <c$globz.h> differs from installation to installation, it is not provided with ALCS. *ALCS Installation and Customization* describes how to create the header file <c$globz.h>.

Read-only access to global records and fields

If no application programs modify the data you want to access, you can use the `glob` function. The `glob` function returns a `void` pointer to the specified global data. If this is a global field, you can access the data by casting this pointer as appropriate (remember, you must not modify the field). For example, if _BYTE is a 1-byte global field, you might code:

```
#include <tpfglbl.h>
#include <c$globz.h>
  ⋮
my_variable=(unsigned char *) glob(_BYTE);
```

If the data is a global record, you use the pointer returned by the `glob` function to access your `struct` for the record. (There is an example in the description of the `glob` function in *ALCS Application Programming Reference - C Language*.) Again, remember that you must not modify any of the fields in the global record.

Serializing access to globals

If **any** application program modifies the data you want to access, you must use the special serializing facilities provided by the `global` function (see the *ALCS Application Programming Reference - C Language*). You must use these serialization facilities even if your program does not modify the data.

Because the `global` function only provides a subset of the facilities available to assembler language programs, you might need to call an assembler program to access read-write globals. In particular:

- You cannot use the `global` function with global records (only with global fields). If your application needs to access read-write global records, you must call an assembler language program to do this.
- The `global` function serialization facilities involve significant processing overheads. If your application is performance sensitive and requires many global area accesses, then you might need to call an assembler language program which can use more efficient methods to serialize accesses.
- The `global` function serialization facilities only serialize accesses correctly when all programs that access the field use compatible serialization methods. These are:
  - C language programs use the `global` function
  - Assembler programs use the `CORHC`, `CORUC`, or `SYNCCs`.

  If some programs use different serialization methods (available only in assembler programs) then you must call an assembler program to access the data using the appropriate serialization method.

Using the `global` function

To access a read-write global field, you must first lock the field (using the GLOBAL_LOCK value of the *action* parameter of the `global` service). After locking the field, you can access it using the GLOBAL_COPY value of the *action* parameter of the `global` function.

If you do not need to modify the field, you can then unlock it using the GLOBAL_UNLOCK value of the *action* parameter of the `global` function.

If you need to modify the field then you must do so using the GLOBAL_MODIFY value of the *action* parameter of the `global` function. You must then use the GLOBAL_SYNC (not the GLOBAL_UNLOCK) value of the *action* parameter of the `global` function to unlock the field.

TPF compatibility:

If your program must be compatible with TPF, use the `modec` function to ensure that all accesses to the global area (fields or records) are in 31-bit addressing mode.

The `modec` function has no effect under ALCS. All ALCS C programs run in 31-bit addressing mode.

ALCS services for global area handling

ALCS provides the following assembler macros and C functions for use in application programs:

Assembler macro	function	Used to
`FILKW`		Reverse the effect of `GLMOD` and keypoint a global record
`GLMOD`		Set the protect key to allow access to a protected global area
	`glob`	Address application global field or global record; this allows you read-only access to a global field or global record
	`global`	This function allows you to update, modify, keypoint, lock, unlock, and synchronize a global field
`GLOBZ`		Establish the DSECT and the base register for a global area directory and for the directly-addressable global records or records which follow the directory
`GLOUC`		Write keypointable records from the application global area to DASD
`KEYCC`		Set the protect key to allow access to a protected global area
`KEYRC`		Restore the protect key to the entry storage key after access to a protected global area

Assembler macro	function	Used to
KEYUC		Write keypointable records from the application global area to DASD
SYNCC	global	Synchronize access to the application global area

Note: In ALCS, GLOUC and KEYUC both have the same effect.

Global area protection

ALCS provides global area protection, an optional feature that enables different storage protect keys to be used for different parts of the global area.

Applications which run under TPF use global area protection, therefore if you migrate a TPF application to ALCS you may wish to use global area protection when you run that application on your ALCS system.

The GLBLPROT parameter on the ALCS system generation SCTGEN macro determines whether global area protection is enabled or not (see the *ALCS Installation and Customization* manual).

Global area protect key

ALCS can optionally use different storage protect keys for the three different parts of the global area.

If you are not using ALCS global area protection then all parts of the global area have the same protect key as the ECB and working storage blocks.

If you are using ALCS global area protection then parts of the global area have different protect keys as follows:

- Some parts of the global area have the same protect key as the ECB and working storage blocks. These parts are called **unprotected global areas** containing **unprotected globals**.

  Application programs can read from (**fetch access**) and write to (**store access**) unprotected global areas.
- Other parts of the global area have a different key. These parts are called **protected global areas** containing **protected globals**.

  Application programs have fetch access, but not store access, to protected global areas. You can get store access to protected global areas by changing the PSW protect key.

  To change the PSW protect key, use GLMOD or KEYCC. After updating the protected global area, use FILKW or KEYRC to restore the previous protect key.

  While the application program has store access to the protected global area, it cannot update the ECB, working storage blocks, or unprotected global areas. However, it can read them, because it still has fetch access to these areas (they are not fetch protected by ALCS).

Using global area protection

About this task

Table 2 on page 63 describes the TPF global areas.

Table 2. TPF global areas		
Global area	**Protection**	**Contents**
1	Protected	Global area directory 0 (GL0BA) Directly-addressable global records Indirectly-addressable global records
2	Unprotected	Indirectly-addressable global records
3	Protected	Global area directory 1 (GL0BY) Directly-addressable global records Indirectly-addressable global records

Table 3 on page 63 describes the ALCS global areas when global area protection is enabled.

Table 3. ALCS global areas with global area protection		
Global area	**Protection**	**Contents**
1	Protected	All ALCS global area directories Directly-addressable global records Indirectly-addressable global records
2	Unprotected	Indirectly-addressable global records
3	Protected	Indirectly-addressable global records

Note: Global records which are not directly addressable may be either protected or unprotected, depending on the requirements of your installation.

The ALCS global area directories are all held in global area 1. However, to allow for compatibility with TPF, ALCS supports macro usages which refer to global area 3 under TPF. Under ALCS, these macros refer to the second global directory (GL1BA). The REGS parameter of GL0BZ is an example of this kind of macro usage. There are no TPF equivalents to the ALCS directories GL2BA, GL3BA, and so on.

If you are not using ALCS global area protection, you do not need to use the GLMOD or KEYCC macros to update an ALCS global area. Using these macros will not generate any errors, although they will add a small overhead.

However, if you are using ALCS global area protection, or if you must remain compatible with TPF, use GLMOD or KEYCC when you update records in global areas 1 or 3 (the protected global areas) as follows:

Procedure

1. Issue GLMOD before storing into global areas 1 or 3.
2. Between GLMOD and FILKW, **do not**:

   Store into the ECB or working storage blocks
   Store into global area 2
   Issue any monitor-request macro.
3. Issue FILKW after storing into global areas 1 or 3.

Results

Figure 46 on page 64 shows an example of this procedure. It updates fields in global area 1, in a directly-addressable record following directory 0 (GL0BA). The GL0BA directory slot @GBLCC contains the storage address and file address of the record.

```
GLOBZ REGR=R03       ADDRESS GL0BA AND ADJACENT RECORDS
GLMOD GLOBAL1        ACCESS PROTECTED GLOBAL
MVC   @FIELD1,EBW000 UPDATE A FIELD IN @GBLCC
ST    R07,@FIELD2    UPDATE ANOTHER FIELD IN @GBLCC
FILKW R,@GBLCC       RESTORE KEY AND KEYPOINT
```

Figure 46. Updating fields in protected global areas

Using Structured Query Language (SQL)

ALCS application programs can use SQL to retrieve and modify data held in relational databases managed or accessed by DB2® for z/OS. For example, an application can build a record on the real-time database, then use SQL to update a relational database for subsequent use by other applications.

Preparation

The DB2 precompiler produces an output source module that is the same as the input source module, except that SQL statements are replaced by instructions or statements in the appropriate language. For example, in assembler language programs, the DB2 precompiler replaces SQL statements with assembler DSECTs, BALR instructions, and so on. In order to allow reentrant programs, the precompiler puts all the variables and structures it generates into a DSECT (array) called SQLDSECT, and generates a fullword called SQLDSIZ (`sqldsiz`) which contains the length of the DSECT (size of the array).

Assembler programs must allocate an area of size SQLDSIZ (for example, in an attached storage block), set it to zeros, and provide addressability to it as the DSECT SQLDSECT.

Making SQL calls

For detailed information about coding SQL statements in application programs, see *DB2 for z/OS Application Programming and SQL Guide*.

SQL communication area (SQLCA)

When an application program includes executable SQL statements, it must be able to communicate with DB2. For it to do this, you must provide an SQL communication area (SQLCA) in the program by coding an `EXEC SQL INCLUDE SQLCA` statement.

Assembler programs must allocate an area for the SQLCA (for example, in an attached storage block), and provide addressability to that area.

When DB2 completes processing an SQL statement, it sends back a return code in field SQLCODE in the SQLCA. The application program should test the return code to examine the results of the operation. For information about possible return code values, refer to *DB2 for z/OS Messages and Codes* .

In application programs you can use the DSNTIAR service to convert an SQL return code to a text message.

SQL descriptor area (SQLDA)

If the application program includes a varying-list SELECT statement, you must also provide an SQL descriptor area (SQLDA), by coding an `EXEC SQL INCLUDE SQLDA` statement.

Assembler programs must allocate an area for the SQLDA (for example, in an attached storage block), and provide addressability to that area.

Coding SQL calls

For detailed information about including SQL statements in application programs, refer to *DB2 for z/OS Application Programming and SQL Guide*.

You can code an SQL statement at any place in an assembler program where you can use an assembler instruction. The SQL statement must begin with EXEC SQL. You can use a non-blank character in column 72 to continue the statement on a further record. For example, an UPDATE statement would look like this:

```
        EXEC SQL                                                  X
          UPDATE DSN8220.DEPT                                     X
          SET MGRNO = :MGRNUM                                     X
          WHERE DEPTNO = :INDEPT
```

Figure 47. An SQL UPDATE statement -- assembler

You can code an SQL statement at any place in a C language program where you can place an executable C statement. The SQL statement must begin with "EXEC SQL." The remainder of the statement can appear on the same line and on following lines, terminated with a semicolon (;).

All SQL keywords must be in uppercase, and "EXEC SQL" must be on one line. For example:

```
      EXEC SQL
        UPDATE DSN8220.DEPT
        SET MGRNO = :mgrnum
        WHERE DEPTNO = :intdept;
```

Figure 48. An SQL UPDATE statement -- C language

Distributing application logic across several programs

ALCS allows SQL application logic to be distributed across several application programs. While the application is processing an entry, it can execute several separate **units of recovery**. A unit of recovery starts when the first DB2 object updates occur. It ends when the application program issues a subsequent COMMIT or ROLLBACK statement, or when it exits.

If the application program requests the EXITC (`exit`) service before a unit of recovery has ended, ALCS issues an implied COMMIT on its behalf.

If the application program terminates abnormally before a unit of recovery has ended, (for example, it requests a SERRC (`serrc_op`) service with a SERRC_EXIT value of the *status* parameter), ALCS issues an implied ROLLBACK on its behalf.

For more information about SQL, see *SH number*.

Designing databases for use with ALCS

For information on designing databases, see *ALCS Concepts and Facilities*.

Performance and design

The record size that you chose for storing a particular type of data can make a big difference to the performance of your application.

If the record size is too small, many of your prime records require overflow records and your application often needs to do several reads to retrieve the data. Your application will run much faster and use less processor power if it can retrieve the data with a single read.

But if the record size is too large, it will waste space both in the processor memory and on DASD. Also, large records take slightly longer to read than small ones (in practice, the extra time taken to read large records is unlikely to make much difference to the performance of your application).

As a general guide, you should try to chose a record size large enough that it rarely requires an overflow. It is almost always better to chose a size that is too large rather than too small. The reasons for this include:

Product-sensitive Programming Interface

- When your application requests a storage block, ALCS takes that storage from a much larger area that is preallocated for the entry. Using a larger block (for a larger record size) may not actually increase the storage that the entry uses.

End of Product-sensitive Programming Interface

- When your application program completes processing an entry, *all* the storage it uses is released for use by other entries. The effect of keeping smaller amounts of storage (for small record sizes) for a longer time (waiting for reads to complete) can be to *increase* the total amount of storage that ALCS needs to process entries.
- Very small record sizes, such as 381-byte L1 records, use DASD space inefficiently. For typical modern DASDs, 381-byte records "waste" approximately half the available DASD space because of the gaps which separate consecutive records.
- As applications develop over a period of years, it often happens that they need to store more and more information. What seems like an ideal record size today may prove too small in years to come.

  The IPARS seat reservation application was originally designed to store passenger data in 381-byte records. At the time, this was big enough to hold the information for a typical passenger. Increasing sophistication of airline reservation systems means that the information saved for each passenger requires, on average, a chain of two or more 381-byte records.

TPF Database Facility (TPFDF)

IBM's TPFDF product allows a data administrator (instead of the application programmer) to specify the record sizes to use for specified data. Subsequently, the data administrator can change the record sizes if required to improve performance - this does *not* require changes to the application programs that access the data.

Portability

To make your application as portable as possible, avoid organizing your data in a way that makes your application programs contain dependencies on particular record sizes. For example, if your application uses L5 records and depends on each L5 record containing (say) 16KB of application data then it may be difficult to port the application to an ALCS system that has a different size for L5 records or to a TPF system (TPF does not support size L5).

The IBM TPFDF product makes this particularly easy to do. TPFDF allows your program to store and retrieve data without knowing the record size that TPFDF is using for the data. TPFDF automatically handles overflow and chaining for you. (This is only one benefit of using TPFDF - there are many others.)

Even if you do not use TPFDF, you can program to reduce or eliminate dependencies on record sizes. For example, you should avoid using a constant as the size (number of bytes) of a record. One way to do this is to define a symbol for the record size in an assembler data macro, C header file, or COBOL WORKING-STORAGE SECTION.

This allows you to change the value of the symbol and reassemble or recompile the program if a different installation chooses to use a different record size. Better still is to have your program determine the record size at execution time. Your programs can do this by reading the record and loading the size from the storage level (see “Reading and writing DASD records” on page 38).

Organizing data

The way that you organize the data in your ALCS database - how you use lists, indexes, and other structures - can make a big difference to the performance of your application.

As a general rule, you should aim to minimize the number of records that your application must read to access the data that it needs.

You must also consider how your application will serialize access to the data, see “Serialization” on page 41.

Standard structures

The ALCS Recoup facility (described in *ALCS Installation and Customization*) supports a number of "standard" structures for lists, indexes, and so on. IBM recommends you to use these standard structures, where possible, rather than inventing new ones.

TPF Database Facility (TPFDF)

Designing and coding application programs to access data through structures of lists, indexes, and so on can be a complex and time-consuming task. Also, over time the amount of data that you store can change. If it increases, you may need to introduce indexes to improve performance. If it decreases, existing index structures may become an unnecessary overhead. The cost of changing existing application programs to do this can prevent you from optimizing your application's performance.

TPFDF makes the initial designing and coding of the application, and subsequent optimization, particularly easy to do. TPFDF allows your program to store and retrieve data without knowing the indexes, lists, and so on that TPFDF is using for the data. It also allows your database administrator to change the structures, for example by adding or deleting indexes, without requiring any changes to the application programs.

See the “Bibliography” on page 270 for information about TPFDF.

Record classes

There are three different **classes** of records for application use on the ALCS real-time database: **fixed file**, **long-term pool file**, and **short-term pool file**. The long-term pool file and short-term pool file are sometimes referred to collectively as "pool file" or just "pool".

Fixed files

ALCS application programs access fixed files in much the same way that any application accesses direct access (sometimes called "random access") files. To access a particular record, the application specifies the file and the relative record number of the record within the file.

ALCS application programs do not use data set names or file names for fixed files. Instead, they use a special token called the **fixed-file record type** (in TPF, this is called the "FACE ID"). In ALCS, the relative record number is called the **fixed-file record ordinal number** (in TPF, it is called the "FACE ordinal"). Before actually reading or writing a fixed-file record, ALCS application programs use an ALCS service to convert the fixed-file record type and ordinal into a 4-byte token called the **file address**.

ALCS application programs cannot create or delete fixed files, and they cannot change the number of records in a fixed file. (Your system programmer or database administrator does these things.)

Pool files

In addition to the fixed files, ALCS supports a number of large pools of records. ALCS application programs do not specify the file and the relative record number of a particular pool-file record. Instead, they use an ALCS monitor service to allocate a record from one of these pools. This service is called **dispense**.

The monitor service returns a file address that the application stores as a pointer in another record. For example, an application program might use a pool-file record as an overflow record - storing its file address in the prime record. Alternatively, the application program might store the file address in a list or index record. Both these methods are described in *ALCS Concepts and Facilities*.

Note that to the application program it is unimportant *which* pool record ALCS allocates. However it is important that the record is not already in use for some other purpose.

Subsequently, application programs access the record using the stored file address.

Note:

An exception to this general rule is the **PNR locator** used in airline seat reservation systems. These systems store passenger information in passenger name records (PNRs). PNRs are pool records, and the application saves the file address of the PNR in another record (the passenger name index) in the normal way. But the application also generates an encoded form of the file address of the prime PNR. This encoded file address is called the PNR locator. The passenger is provided with the PNR locator (it is, for example, printed on the ticket).

Since the PNR locator is effectively the file address of the prime PNR, the application can use it to retrieve the PNR directly, without first accessing the passenger name index.

Eventually, if the data contained in the record is no longer needed, an application program can clear the saved file address to zeros and use an ALCS monitor service to return the record to the available pool. This service is called **release**.

Short-term pool file

Application programs can use short-term pool-file records to store data for short periods of time - a few seconds or minutes.

Your application must release a short-term pool-file record within, at most, a few hours after the dispense. If it does not release a short-term pool record within a reasonable time (typically 24 hours), ALCS releases the record itself.

The actual amount of time before ALCS releases a short-term pool record depends on the rate at which applications dispense and release short term pool. If many applications fail to release short-term pool records, then the short-term pool becomes depleted (there are no available records left). To prevent short-term pool depletion, ALCS dynamically adjusts the amount of time before it releases records itself. IBM recommends that your system programmer or database administrator allocates enough short-term pool records to ensure that this amount of time is 24 hours or more. But under conditions of exceptional load, the actual time can be less than 24 hours.

A short-term pool-file record becomes available immediately after the release which means that the record can be reused, destroying its previous contents. It is important to ensure that your application clears any pointer to a short-term pool record to zeros *before* it releases the record.

Your system programmer or database administrator can allocate one short-term pool file for a record size. For example, your installation may have four short-term pool files, one for for each of the sizes L1, L2, L3, and L4.

Long-term pool file

Unlike short-term pool-file records, application programs can use long-term pool-file records to store data for long periods of time - days, weeks, or years.

Like short-term pool-file records, your application should release a long-term pool record when the information it contains is no longer required. And it is important to ensure that your application clears any pointer to a long-term pool record to zeros before it releases the record.

But unlike short-term pool-file records, there is no time limit within which your application must release a long-term pool record. Also, a long-term pool record does not become available immediately after the release. Instead, an ALCS utility called Recoup must check that there are no pointers to the record saved in the ALCS database before ALCS allows the record to be reused.

Your system programmer or database administrator can allocate one long-term pool file for a record size. For example, your installation may have four long-term pool files, one for for each of the sizes L1, L2, L3, and L4.

Choosing which class of record to use

For most ALCS applications, you will need to organize records in a way that combines efficient access (minimizing the number of DASD I/Os) with efficient use of space (minimizing the number of "wasted" records on the database).

You may need to design records in a way that allows overflow from a prime record to a chain of one or more overflow records. In general, always use pool files for overflow records, even when the prime record is a fixed-file record. Your application will only need to access an overflow record after it retrieves the prime record (which contains the file address of the overflow record).

You may need to use list or index records to contain the addresses of the prime records. If so, use pool file for both the prime records and the overflow records. Your application does not need to calculate the file address of a particular prime record, because it gets the address from the list or index record.

It is often appropriate to use fixed file for list or index records (but only for the *prime* record - use pool file for overflow list or index records).

When you are deciding between using long-term or short-term pool file, be aware that short-term pool file is only suitable for transient data. You cannot safely store information in short-term pool for more than a few hours. If your application routinely stores data for many hours in short-term pool records, you risk not only losing that data, but also depleting the pool. This may force ALCS to reuse other short-term pool-file records containing data belonging to other applications.

Also, short-term pool is much less secure than long-term pool. If an application error prematurely releases a short-term pool record, or if ALCS itself releases the record while it still contains data then that data is lost.

Protecting data

Data can be encrypted and stored on the ALCS database, a relational database (DB2), or a sequential file. It can be sent through a communication link to another host system. Up to 4KB of data can be processed in one operation by using the CRYPC macro.

Symmetric key encryption and decryption is used; that is, the same key is used for encrypting the data and decrypting the data.

If the encrypted data is stored only on the ALCS system, decryption is easy as the encryption key and processing rules (which encryption process is used and other details) is known to ALCS.

If the data is sent to another host system, some way must be found of sending the key information and processing information without compromising security.

If the host is on the same z/OS system sharing the same Cryptographic Key Data Set (CKDS), this is trivial as the key information can be retrieved from the CKDS and the processing rules easily determined.

If the data is to be sent to another system not on the same z/OS image, the key information can be sent to that system using functions provided by ICSF. The processing rules can be sent by that method as well.

The encryption and decryption routines use the CFB processing rule. This rule is used because it allows any length of data to be processed (it does not have to be a multiple of 8) and the processed data takes up the same space as the input data.

A user exit is provided to allow the user to handle a simple key change policy whereby the keys that are used to encrypt or decrypt the data can be changed, as is required by many security standards bodies.

If data needs to be sent to another system, a different encryption key and processing rule can be used if necessary. See the *ICSF Administrators Guide* and *ICSF Application Programming Guide* for more information about which keys and processing rules are supported.

The keys that are used are accessed by name. If the correct hardware and software is available, encrypted keys can be used. These keys cannot be read from the ALCS address space and do not appear in any dumps or core displays.

If the correct hardware or software is not available, clear keys can be used and RACF® can be used to provide protection against unauthorized access.

Commit and backout

When an entry needs to update several records, it can be important to be sure that either all the updates complete (the entry completes normally) or none of them do (the entry fails). An example is electronic funds transfer in banking applications. A transfer consists of debiting one account and crediting another. If the program that is doing this debits one account and then abnormally terminates then money "disappears". This situation can arise because of application programming errors, or it can happen because of equipment (hardware) errors or failures.

Single-phase commit

Many transaction processing systems include facilities that help protect the database against incomplete or inconsistent updates when a transaction fails part way through. These systems allow an application to identify a group of related database updates. For each group of related updates, the system *guarantees* one of the following actions:

Commit

The system writes *all* of the updated records to the database. There are two types of commit:

- The application requests the commit service from the transaction processing system.
- The application completes normally (without error). This is called **implied commit**.

Backout

The system writes *none* of the updated records to the database. There are two types of backout:

- The application requests the backout service from the transaction processing system.
- The application completes abnormally (with error). This is called **implied backout**.

Some systems write each record as it is updated and perform backout by restoring the records to their previous contents. Other systems do not write any records until commit time.

Two-phase commit

If a single set of related updates affects two or more separate databases, a more complex process called **two-phase commit** is sometimes used. Two phase commit uses a **synchpoint manager**, and works as follows:

1. The synchpoint manager asks each database to commit its part of the set of related updates. Each database tells the synchpoint manager when it successfully completes this (phase 1) commit process,
2. When *all* the databases successfully complete the phase 1 commit process, the synchpoint manager tells all the affected databases. This is phase 2.

   If any database does not successfully complete that phase 1 commit, the synchpoint manager asks all the databases to backout the change.

ALCS commit and backout services

ALCS provides single-phase commit services (commit, implied commit, backout, and implied backout) for SQL updates to relational databases.

ALCS does *not* provide commit services for updates to the ALCS database, ALCS general files (GDSs), or sequential files. Also, ALCS does not provide commit services for MQI `MQPUT` and `MQGET`.

It is important to remember this when designing and writing ALCS applications. In particular, you should try to design your applications so that:

- Each update should leave the database in a consistent form. There is a particular technique (see "Sequential update technique" on page 71) that you can use to help achieve this.
- Errors and inconsistencies in the database do not cause your application programs to "crash". Instead, your applications should detect and report this type of problem (possibly by requesting a system error dump) in a way that helps your database administrator, end user, or whoever is appropriate to take the required corrective action.

Note: On a typical ALCS system, only a very small proportion of entries (less than one in a million) fail leaving inconsistent updates. However you can significantly reduce this proportion, and reduce the undesirable consequences, by following the guidelines described here.

Keeping a transaction log

Many ALCS users find it useful to keep a record of some or all input messages. You may be able to use a log of this type to help correct errors or inconsistencies in your database.

If you want to do this, you can write the input messages to an ALCS real-time sequential file.

Sequential update technique

By planning the sequence in which you update records on the database you can minimize, or even or eliminate the possibility of being left with inconsistent data should a hardware or software error occur.

Most ALCS databases store the vast majority of the data in long-term pool records. Applications access long-term pool records using a pointer (file address) stored in another record (the "refer-from" record). You can exploit this to avoid introducing inconsistencies in the database when adding or changing complex structures.

Adding a new structure

You can *add* a new structure, which can be either a chain of records or a more complex structure including lists and indexes. See Figure 49 on page 72 for a list of actions.

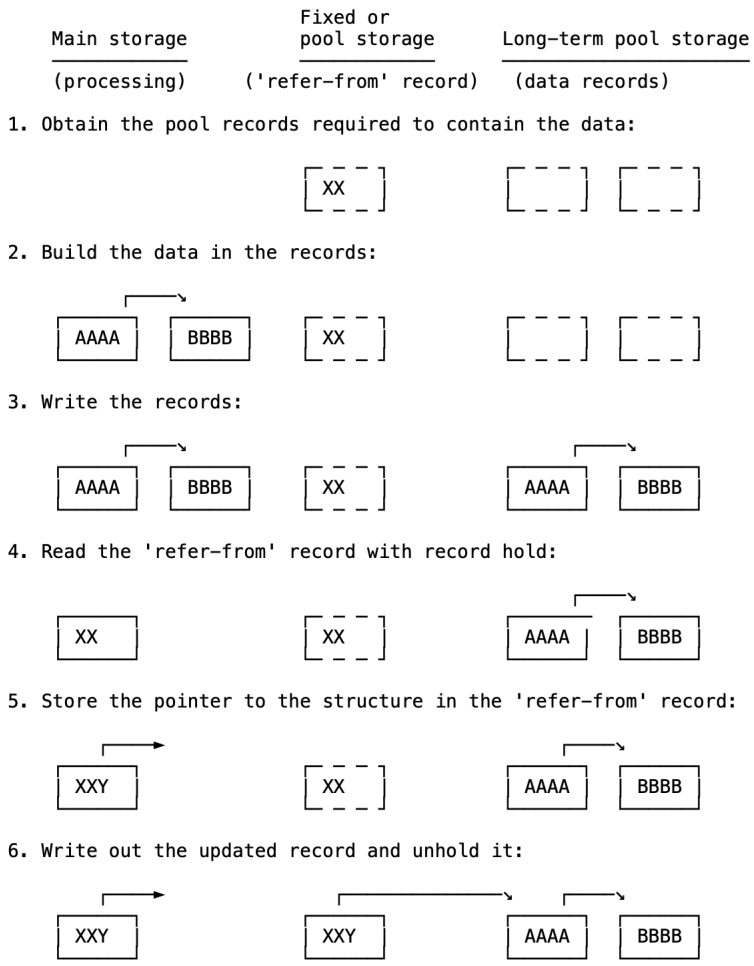


Figure 49. Adding a new structure

If your application (or the entire ALCS system) fails before this process is complete, then the database does not contain a pointer to the partially built structure. (The records in the structure are "lost", but will eventually be recovered for reuse by Recoup.)

Note that the above sequence only requires the entry to lock one record (the "refer-from" record), and only for the short time taken to update the pointer.

Updating a structure

You can *update* a structure using a sequence of actions as shown in Figure 50 on page 73. (The starting structure is assumed to be the structure added in Figure 49 on page 72.)

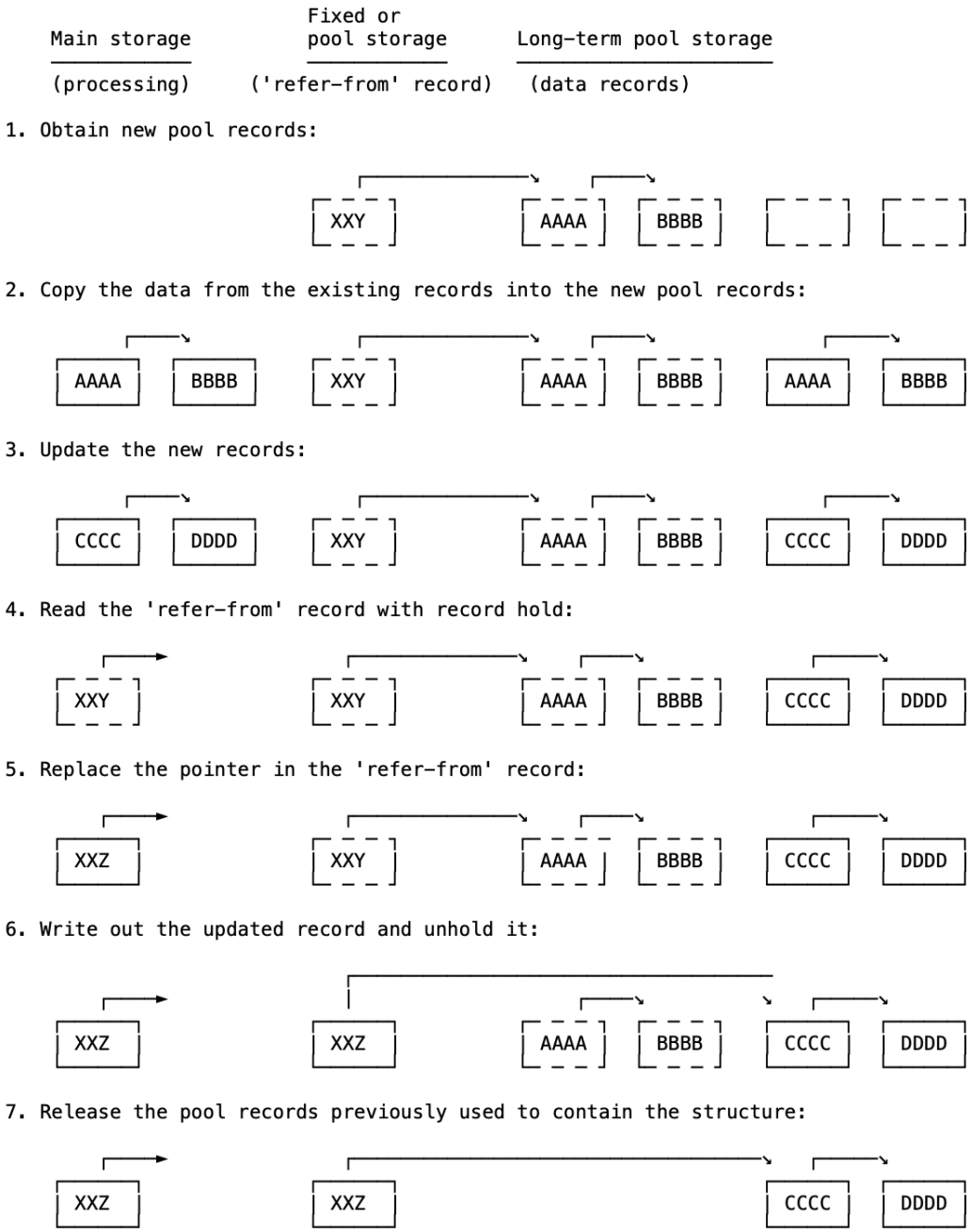


Figure 50. Updating a new structure

Your application can fail at any stage in this process without leaving any reference to a partially updated structure. Again, the above sequence only requires the entry to lock one record (the "refer-from" record), and only for the short time taken to update the pointer.

Why ALCS does not provide a synchpoint manager

If you are happy that ALCS does not provide commit and backout services for its database and that it only provides single-phase commit for SQL, you can skip this section. If you think that ALCS should provide comprehensive commit and backout services, or if you are interested to know why it doesn't, then you should read this section.

It might appear obvious that any transaction processing system should provide services to prevent inconsistent or partial updates to the database - particularly since many systems do provide these services.

These services provide the following main benefits:

- They safeguard your data by backing out partly completed transactions when an equipment (hardware) or system program (software) failure terminates the entries.
- They safeguard your data by backing out partly completed transactions when an application programming error terminates the transactions.
- They simplify application program design and coding by making the *system* responsible for database consistency instead of the application program.

However, commit and backout facilities impose significant overheads. Transaction processing systems that provide these facilities typically require several times as much processor power as ALCS systems do for the same transaction rate. Similarly, the restart times for systems that guarantee to backout partly completed transactions are dramatically longer than ALCS restart times.

Other considerations

Although commit and backout facilities can help reduce inconsistencies in databases, they cannot guarantee the correctness of the data. Consider the following:

- It is clearly undesirable if a system failure results in a debit to one bank account without a corresponding credit to another. However, it is just as serious if a human error causes the wrong account to be credited or debited (or both).
- It is clearly undesirable if an application program error terminates a transaction part way through, leaving inconsistent or incorrect data in the database. However, it is just as serious if an application programming error stores incorrect or inconsistent data without terminating the transaction abnormally.

  And generally, the first type of error is much more quickly identified and corrected than the second type of error. Abnormal termination is a more obvious symptom than incorrect data on the database.

ALCS is designed to satisfy the requirements of systems where the costs of commit and backout facilities outweigh the advantages.

Suppose your system processes 1000 transactions per second during the peak hour - equivalent to perhaps 10 000 000 transactions per day. Suppose also that it runs for 3 months without a system failure. Your system will probably process something like 2 000 000 000 transactions during the 3 month period. When the system fails, perhaps 200 transactions will fail part way through. This means that roughly one transaction in ten million would benefit from a full-blown transaction backout facility.

Admittedly, application programming errors can cause abnormal termination that leaves part-completed updates. But even a full two-stage commit facility will not protect your database against application programming errors. Two-phase commit only guarantees to write (or not write) the data provided by the application. It does not guarantee that the data "makes sense".

Utility programs

In addition to the services (assembler macros and C functions) which are used by ECB-controlled programs, ALCS provides some utility programs that can be used by programs running under MVS:

DXCCDDLD
Load DASD configuration table load module.

DXCMRT
Return the highest used general file number, pool interval number, or fixed-file record type number.

DXCFARIC
Return information about a supplied file address.

These programs are described in more detail in *ALCS Application Programming Reference - Assembler*.

Chapter 4. The entry control block (ECB) and other entry storage

This chapter describes the ECB and other entry storage.

For each entry, ALCS creates a unique 4KB storage area called the **entry control block** (ECB). Application programs can use the ECB to hold data and variables needed by a particular entry.

Format of the ECB

The areas of the ECB which concern the application programmer are shown diagrammatically in Figure 51 on page 76.

Most areas in the ECB consist of several fields. Each field has a label and the area as a whole usually has a label. In assembler programs labels are in upper case (for example, EBW000). C language labels are in lower case (for example, `ebw000`).

Alternative names for fields in the ECB

Some fields in the ECB have two different names. One of these names starts with the characters 'EB' (`'eb'` in C language programs), the other starts with the characters 'CE1' (`'ce1'`).

The two named fields are not necessarily exactly the same (their lengths might be different). In this book, the most common field name is mentioned first.

© Copyright IBM Corp. 2003, 2019

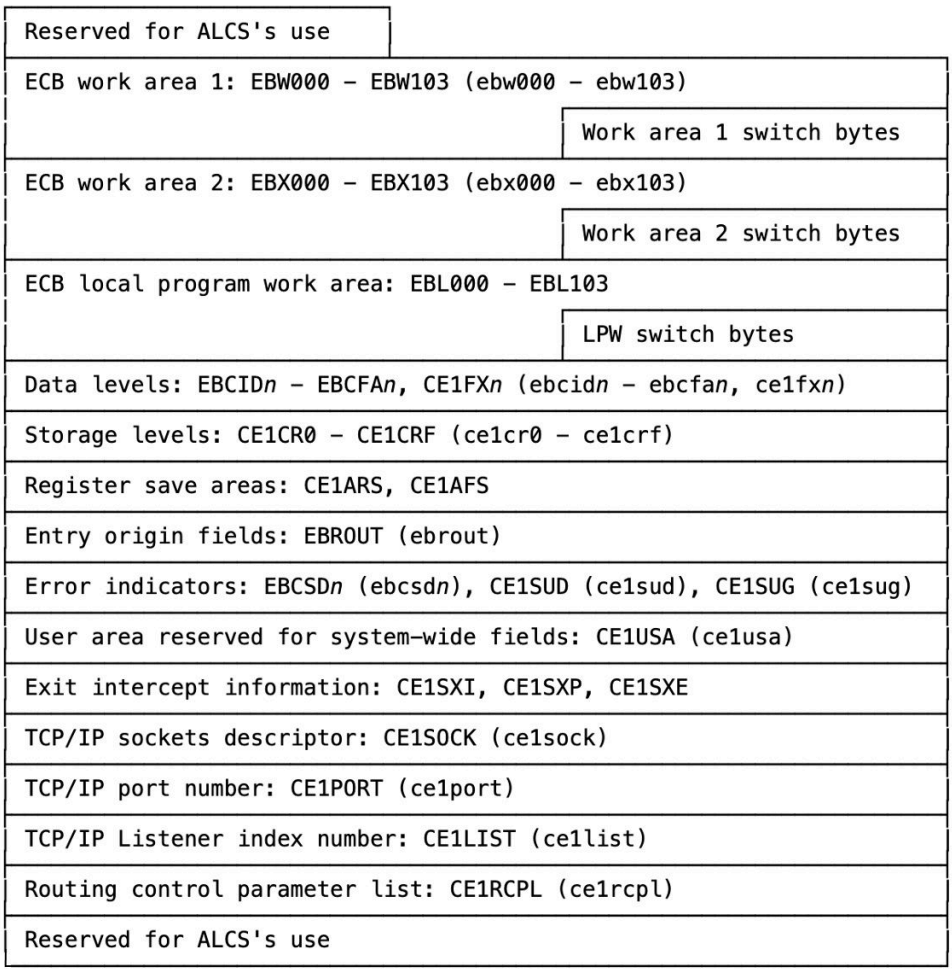


Figure 51. Format of the ALCS ECB (schematic)

ECB work areas 1 and 2

In C language programs you do not normally need to use these work areas, since you can define storage automatically, as described in "Automatically-allocated storage" on page 84. However, you might need to use one or the other of the work areas to pass data to an assembler program called from a C program.

The ECB contains two similar work areas, each 104 bytes long, plus 8 bytes of switch data, which application programs can use. When ALCS creates an ECB, it sets both work areas to binary zeros. ALCS does not use either work area itself, nor does it check the use that programs make of each work area.

TPF compatibility:

TPF systems might not set the work areas to binary zeros, this is installation-dependent.

Work area fields

You can access any of the first 104 bytes in either work area by using the names shown in Table 4 on page 76. The assembler name is shown first followed by the C language name (in parentheses).

Table 4. Names of work area fields

First work area	Second work area	Byte referenced
EBW000 (ebw000)	EBX000 (ebx000)	1st

Table 4. Names of work area fields (continued)		
First work area	**Second work area**	**Byte referenced**
EBW001 (ebw001)	EBX001 (ebx001)	2nd
EBW002 (ebw002)	EBX002 (ebx002)	3rd
. . . . .	. . . . .	. . . . .
EBW103 (ebw103)	EBX103 (ebx103)	104th

The first byte of each work area starts on a doubleword boundary. Each field is 1 byte long (type `char`).

Switch bytes in work area 1

You can access the fields in this 8-byte area as follows:

EBSW01, EBSW02, EBSW03 (ebsw01, ebsw02, ebsw03)

Conventionally, these three 1-byte fields are used for 1-bit switch data.

EBRS01 (ebrs01)

This 1-byte field is intended for 1-bit (switch) data. Application programs conventionally use it to control application program linkage to assembler programs.

EBCM01, EBCM02, EBCM03 (ebcm01, ebcm02, ebcm03)

These three 1-byte fields are conventionally used for 1-bit (switch) data that is to be transferred between application programs.

EBER01 (eber01)

This 1-byte field is conventionally used as an error indicator. Application programs use it when one program detects an error and another program carries out the error processing. See Chapter 6, “Error conditions and error processing,” on page 110.

Switch bytes in work area 2

Work area 2 also contains 8 switch bytes, as follows:

EBXSW0 - EBXSW7 (ebxsw0 - ebxsw7)

These eight 1-byte fields occupy positions in the second work area corresponding to fields EBSW01 through EBER01 (ebsw01 through eber01). Application programs can use these fields in any way.

Alternative names

Both work areas have alternative names:

CE1WKA, CE1WKB (ce1wka, ce1wkb)

These are alternative names for the whole of work area 1 or work area 2. Each has a length of 112 bytes. Use CE1WKA (ce1wka) to refer to work area 1 and CE1WKB (ce1wkb) to refer to work area 2.

ECB local program work area

Use of the local program work area is supported in assembler language programs that specify the BEGIN macro parameter LPW=YES. It is not supported in C language programs.

TPF compatibility:

Do not use the local program work area in programs that must be compatible with TPF.

This work area differs from work areas 1 and 2 in that it is "local" to the program. On entry to the program ALCS clears the local program work area to binary zeros, so data in this area cannot be passed between application programs.

When the program issues an ENTRC macro (return expected to this program), ALCS saves the contents of the local program work area. It restores the contents of the local program work area when control returns to the program following the ENTRC macro.

When the program issues an ENTNC macro (return to the program that last issued an ENTRC macro), ALCS does not save the contents of the local program work area.

When the program issues an ENTDC macro (no return expected), ALCS discards all saved copies of the local program work area.

Notes:

1. Transferring control to a transfer vector has the same effect on the local program work area as described above (that is, depends on the type of transfer). This is true even if the transfer vector is an entry point in the same program, because the local program work area actually relates to the program nesting level.
2. Transferring control to an exit intercept program (set by SXIPC macro) when an entry terminates has the same effect on the local program work area as ENTDC.

Work area fields

You can access any of the first 104 bytes in the local program work area by using the names shown in Table 5 on page 78.

Table 5. Names of local program work area fields

Name of field	Byte referenced
EBL000	1st
EBL001	2nd
EBL002	3rd
.....	.....
EBL103	104th

The first byte of the local program work area starts on a doubleword boundary. Each field is 1 byte long.

Switch bytes

The local program work area contains 8 switch bytes, as follows:

EBLSW0-EBLSW7:
These eight 1-byte fields occupy bytes 105-112 in the local program work area.

Alternative name

The alternative name for the whole of the local program work area is CE1WKC. This has a length of 112 bytes.

ECB data levels and storage levels

There are 16 data levels and storage levels in the ECB, represented by the defined symbols D0 for level 0, D1 for level 1, ..., DF for level 15. Data levels and storage levels, used together, are called **ECB levels**.

Note: The data level and storage level in a DECB may be used as an alternative to an ECB level. For more details about the use of DECBs see “Data event control blocks (DECBs)” on page 88.

ECB data levels

Applications (and ALCS) use each data level to identify a DASD record that is about to be read or written. Each data level contains the following information:

- 2-byte record ID (optional)
- 1-byte record code check (RCC) character (optional).
- 4-byte file address (required).

Note: There is also a 1-byte reserved field in each data level.

Data level fields

The 16 data levels have identical formats. The last character of the field name identifies the level number (hexadecimal 0 through F). In the following description of the data level labels this last character is indicated by *n*.

EBCID*n* (`ebcidn`)

When required (or allowed) by the DASD I/O operation, the application program stores the 2-byte record ID in this field.

EBCRC*n* (`ebcrcn`)

Optionally for most DASD I/O operations, the application stores the record code check (RCC) in this 1-byte field.

EBCFA*n* (`ebcfan`)

For most DASD I/O operations, the application program stores the file address of the record it is reading or writing in this 4-byte (`long int`) field. The 4 bytes have individual names:

```
EBCFMn (ebcfmn)
EBCFCn (ebcfcn)
EBCFHn (ebcfhn)
EBCFRn (ebcfrn)
```

For more details of the use of data levels, see "Reading and writing DASD records" on page 38.

CE1FX*n* (`ce1fxn`)

These 16 8-byte (`unsigned char`) fields are known as the **data level extensions**. They are used with the two macros (C functions) that process general data sets: GDSNC (`gdsnc`) and GDSRC (`gdsrc`).

Note: Data levels are used in a different way by the sequential file macros TOUTC (`toutc`) and TDTAC and by the communication macros SENDC X and ROUTC (`routc`) when TCP/IP or OLA large messages are used.

ECB storage levels

ALCS uses 16 storage levels to attach storage blocks (see "Storage blocks" on page 85) to the ECB. Application programs can access storage level fields, but must not modify the contents, except as described in *ALCS Application Programming Reference - Assembler* for the ATTAC and some event-processing macros.

Storage levels contain the following information:

- 4-byte attached storage block address
- 2-byte block size field
- 2-byte block size indicator.

Storage level fields

The 16 storage levels in the ECB each contain 3 fields. As with data level field names, the last character of the field name identifies the level number (hexadecimal 0 through F). The 3 fields are named CE1CR*n* (`ce1cr`*n*), CE1CC*n* (`ce1cc`*n*), and CE1CT*n* (`ce1ct`*n*).

CE1CR*n* (`ce1cr`*n*)

ALCS stores in this 4-byte (`long int`) field the address of the storage block that it allocates to the program.

CE1CC*n* (`ce1cc`*n*)

ALCS normally stores in this 2-byte field the length of the attached block. However, when a tape read operation produces the error condition "incorrect size (small)" (see Table 11 on page 116), this field contains the length of the data read (see “Recovery from wait-completion errors” on page 121).

CE1CT*n* (`ce1ct`*n*)

ALCS stores in this 2-byte field the block size value. It contains the block type indicator value, L0, L1, L2, ...L8.

For more details of the use of storage levels, see “Reading and writing DASD records” on page 38 and “Storage blocks” on page 85.

Entry origin fields

The ECB contains a number of entry origin fields. EBROUT (`ebrout`) is the only one which application programs can use. It is also known as CE1OUT (`ce1out`).

Application programs can use this 3-byte field to contain the communication resource identifier (CRI) of the originating terminal (not WTTY). ALCS sets up this field for input messages.

User register save areas

The ECB contains two register save areas that assembler application programs can use. One area is provided for general registers and one for floating-point registers. ALCS does not modify these areas.

General registers

Application programs can use the general register save area to save the contents of general registers 0 through 7 (RAC through RGF), 14 (RDA), and 15 (RDB). Table 6 on page 80 shows the labels that application programs should use to refer to the general register save areas:

Table 6. Labels for the general register save area

Label	Area referred to
CE1ARS	Entire save area (General registers 14, 15, 0 through 7)
CE1URA	General register 14
CE1URB	General register 15
CE1UR0	General register 0
CE1UR1	General register 1
CE1UR2	General register 2
CE1UR3	General register 3
CE1UR4	General register 4
CE1UR5	General register 5

Table 6. Labels for the general register save area (continued)	
Label	**Area referred to**
CE1UR6	General register 6
CE1UR7	General register 7

Figure 52 on page 81 shows an example of using the general register save area.

```
        STM   R14,R07,CE1ARS      SAVE REGISTERS
⋮
        L     R06,CE1UR6          RESTORE REGISTER 6
⋮
        LM    R14,R07,CE1ARS      RESTORE REGISTERS
```

Figure 52. Using the general register save area

Floating-point registers

Table 7 on page 81 shows the labels that application programs should use to refer to the floating-point register save area:

Table 7. Labels for the floating-point register save area	
Label	**Area referred to**
CE1AFS	Entire save area (floating-point registers 0 through 6)
CE1UF0	Floating-point register 0
CE1UF2	Floating-point register 2
CE1UF4	Floating-point register 4
CE1UF6	Floating-point register 6

Figure 53 on page 81 shows how you can use the floating-point register save area.

```
        STD   FP0,CE1UF0          SAVE FLOATING POINT REGISTERS
        STD   FP2,CE1UF2
        STD   FP4,CE1UF4
        STD   FP6,CE1UF6
⋮
        LD    FP0,CE1UF0          RESTORE FLOATING POINT REGISTERS
        LD    FP2,CE1UF2
        LD    FP4,CE1UF4
        LD    FP6,CE1UF6
```

Figure 53. Using the floating-point register save area

Only the floating-point registers 0, 2, 4, 6 can be used.

TPF compatibility:

TPF does not support the standard labels for the user floating-point save register area in the ECB. If your program must be compatible with TPF, see the *ALCS Application Programming Reference - Assembler*, which explains how you can define extra labels for TPF.

ECB error indicators

The ECB contains 16 1-byte (`unsigned char`) error indicators, one for each ECB data level. They are called EBCSD*n* (`ebcsdn`) where *n* is the data level (0 through hexadecimal F). When a program requests a `WAITC` (`waitc`) service, ALCS might set one or more of these indicators.

For a description of these error indicators and how to test them, see "Error indicators in the ECB and DECB" on page 115.

ECB user area reserved for system-wide fields

Your system programmer can use this 2248-byte field CE1USA (`ce1usa`) in the ECB to define system-wide fields, as described in *ALCS Installation and Customization*.

Exit intercept information

You can use the SXIPC monitor-request macro to set up an exit intercept program for an entry. SXIPC passes control to the exit intercept program whenever the entry terminates. You can test the following fields in the exit intercept program to determine whether a system error occurred:

CE1SXI
A 1-byte field containing error indicators. Bit CE1SXF is set on if the entry terminated abnormally.

CE1SXP
A 4-byte field containing the name of the failing program, if error indicator CE1SXF is set on and the program name is valid.

CE1SXE
A 3-byte field containing the system error code, if error indicator CE1SXF is set on.

TCP/IP socket descriptor, TCP/IP port number, and Listener index number

When the ALCS TCP/IP concurrent server (Listener) is started, it waits for connection on a specific TCP/IP port. When a client connects to that port, the ALCS concurrent server creates a new ECB and passes control to the installation-wide exit program ATCP if it exists. ALCS puts the TCP/IP socket descriptor for this connection in a fullword field called CE1SOCK (`ce1sock`) in the ECB. It puts the TCP/IP port number in a fullword field called CE1PORT (`ce1port`) in the ECB. Up to eight concurrent servers can be started at the same time, each waiting for connection on a different TCP/IP port. ALCS puts the index number of the concurrent server Listener which received the connection (from 1 to 8) in a fullword field called CE1LIST (`ce1list`) in the ECB.

The routing control parameter list (RCPL) area

With each input message, ALCS provides a **routing control parameter list** (RCPL). The RCPL contains information which identifies the origin and characteristics of the message. ALCS puts this RCPL in a 112-byte area called CE1RCPL (`ce1rcpl`) in the ECB.

An output message sent by requesting a ROUTC (`routc`) service also has an RCPL associated with it. This identifies the destination (and characteristics) of the message. You can preserve the RCPL received with the input message and change it to be used as an output message RCPL (for example, by swapping the origin and destination addresses).

C language programs that only need to return a message to the originating terminal do not need to use RCPLs, instead they use `stdin` and `stdout` as described in "Input and output messages" on page 86. However, by using an RCPL in combination with the C function `routc` an application program can:

Send messages to terminals other than the originating terminal.
Send messages to other application programs (or to itself).
Send unsolicited messages to one or more terminals.

For details of the RCPL contents, see Appendix D, “Messages and message formats,” on page 215.

ECB areas reserved for ALCS use

The ECB contains a number of areas which are reserved for ALCS use. Application programs must not use (or even refer to) fields unless they are part of the general use programming interface.

Accessing the ECB

Assembler programs access the ECB using the DSECT EB0EB.

The C data structure describing the ECB is called `eb0eb`, and is defined in the header file `<c$eb0eb.h>`. Use the C function `ecbptr` to obtain access to fields in the ECB. (The `ecbptr` C function returns the address of the ECB.)

Examples

The following examples show how you could use `ecbptr` to access three storage level fields.

ce1cr*n*

```
struct cm1cm *cm;           /* pointer to message structure */

/* set pointer to message in block at level 1 */
cm = ecbptr()->ce1cr1;
```

ce1cc*n*

```
int size;

 size = ecbptr()->ce1cc1;      /* get size of block on level 1 */
 printf("size of block on level 1 is %d bytes\n",size);
```

ce1ct*n*

```
if (ecbptr()->ce1ct1 == L2)   /* is block on level 2 size L2 ? */
  puts("L2 size block on level 1");
```

You can also use the `levtest` C function to test whether a storage level is in use and to get the size of the block. See *ALCS Application Programming Reference - C Language*.

Other entry storage

As explained at the start of this chapter, ALCS allocates a unique 4KB ECB for each entry. This means that ALCS application programs can process several entries at the same time. An application program can safely store information (for example, an intermediate result) in an ECB field; the same program, storing into the same ECB field for a different entry stores into a different ECB.

In addition to the ECB itself, ALCS can allocate other unique areas of storage for each entry. Like the ECB, application programs can store into these areas without interfering with other entries. Storage that has this property in ALCS is called **entry storage**.

ALCS application programs can use the following entry storage:

The ECB
ECB user data collection area

Automatically allocated storage
Storage blocks
Local save stack
Data event control blocks (DECBs).

ECB user data collection area

Product-sensitive Programming Interface

ALCS provides services that allow application programs to update an area of entry storage called the user data collection area. This area is an optional extension to the ALCS data collection area, and does not exist unless the size is defined using the DCLECBU= parameter of the SCTGEN macro, as described in *ALCS Installation and Customization*. Application programs can use the DCLAC macro to read or update fields in the user data collection area.

End of Product-sensitive Programming Interface

Heap storage used by assembler programs

Assembler programs may use the CALOC, MALOC, RALOC, and FREEC monitor-request macros to obtain or release heap storage.

Whenever an assembler program obtains heap storage a type 1 entry storage block is used, if such a storage block can contain that heap storage. When an entry storage block can not be used (for example because the largest entry storage block size is too small), a type 3 storage unit is allocated, in addition to any other storage units that are already in use.

If the program requires more heap storage than fits in the first type 3 storage unit, ALCS allocates additional type 3 storage units if entry storage blocks can not be used.

You need to be aware that the largest contiguous amount of heap storage that a application can use is a whole type 3 storage unit. You must choose the size of your type 3 storage units so that they are large enough to hold the largest variable (which can be an array or structure) that your application uses.

Note: The system programmer allocates the number and size of type 3 storage units using the NBRSU= and SUSIZE= parameters of the SCTGEN system generation macro, as described in *ALCS Installation and Customization*.

Automatically-allocated storage

High-level language programs usually obtain and use storage by defining variables. In addition, C language programs can obtain storage by using the `calloc`, `malloc`, and related functions, PL/I programs can obtain storage by using the ALLOCATE function.

At execution time, high-level language programs obtain and manage storage to contain dynamic variables and to satisfy C language `malloc`, `calloc`, and related function calls. They also obtain and manage storage for register save areas, for library routines to use as work areas, and so on.

As an application programmer, you do not need to understand in detail how your programs obtain and manage this storage at execution time. The compiler generates code that obtains the storage, as required, by requesting ALCS services. The services allocate *entry* storage so that different transactions use different storage even when the same program is executing. Note that these services are intended for use *only* by compiler-generated code - you must not attempt to invoke the services directly.

Initial storage allocation

When an ALCS ENTER-type service transfers control to a high-level language program, the high-level language application program obtains a storage area called the **initial storage allocation** (ISA).

The size of the ISA depends on the version and release of the compiler and library environment, but is of the order of 100KB.

The ISA storage is released when the high-level language program returns to the calling program, or if it exits without returning control.

Note that the compiler automatically generates code to obtain and release the ISA. The high-level language source code does not include any instructions to do these things.

Stack and heap storage

High-level language programs use **stack** and **heap** storage to contain dynamic variables. C language programs can also obtain heap storage using the `malloc`, `calloc`, and related functions.

During execution, the programs obtain and release stack and heap storage when required. The compiler generates code that manages this storage. This code does not request storage separately for each variable. Instead, it requests storage in amounts that are a multiple of 64KB for stack storage and 256KB for heap storage. (In other environments the amounts can be multiples of a different constant.) Typically a program can store many variables within 256KB, but it is possible that a single variable (for example a large array) can require more than 256KB. For a description of how stack and heap are used, see *z/OS Language Environment Programming Guide*.

When the program requires stack, it requests the required amount (in multiples of 64KB and 256KB respectively) from ALCS. ALCS allocates stack storage in **type 2 storage units** (also called **HLL storage units**). ALCS allocates heap storage in **type 3 storage units** if no entry storage block can be used.

Whenever an ALCS ENTER-type service transfers control to an HLL program, the entry needs at least one **type 2 storage unit** (to hold the ISA) additional to any storage units it is already using. The type 2 storage unit size must be at least large enough to hold the ISA - any remaining space in the first type 2 storage unit is available for use as stack storage.

If the program requires more stack storage than fits in the first type 2 storage unit, ALCS can allocate additional type 2 storage units. If the program requires more heap storage than fits in the first type 3 storage unit, ALCS can allocate additional type 3 storage units if no entry storage block can be used.

You need to be aware that the largest contiguous amount of stack storage that an application can use is a whole type 2 storage unit. You must choose the size for type 2 storage units so that they are large enough to hold the largest stack frame that any HLL program requires (the stack frame includes storage for all function-scoped variables defined in the function, plus the save area and other work areas used by the compiled code). Similarly, the largest contiguous amount of heap storage that an application can use is a whole type 3 storage unit. You must choose the size for type 3 storage units so that they are large enough to hold the largest area requested by `malloc` or a similar C language function.

Note: The system programmer allocates the number and size of type 2 and 3 storage units using the NBRSU= and SUSIZE= parameters of the SCTGEN macro, as described in *ALCS Installation and Customization*

Storage blocks

Assembler and C language programs can request blocks of storage. One typical use of a storage block is to build up a record before writing it to DASD.

An application program can request a block of storage by requesting a GETCC (`getcc`) service. ALCS allocates a block of storage of the specified size (L0 through L8). It saves the address and size of that block in the storage level that the program specifies in the assembler macro or C function call. The application program can request up to 16 storage blocks to be attached to the ECB at any one time (one on each of the 16 ECB storage levels), and can additionally request one storage block to be attached to each allocated DECB.

When an application program has finished using a storage block, it can request ALCS to release the block by requesting the RELCC (`relcc`) service. The storage then becomes available for reuse and ALCS clears

the storage level in the ECB or DECB to indicate that the level is free. ALCS automatically releases all storage blocks when an entry exits.

A program can test if a storage level is currently in use by requesting the `levta` (`levtest`) service.

Input and output messages

ALCS stores an input message in a block and attaches this at storage level 0 (D0) before it enters the first application program that will process the message. For TCP/IP or OLA large messages, ALCS may store an input message in a heap storage area instead of a storage block; in which case ALCS puts the address of the heap storage area in data level D0 before it enters the first application program that will process the message.

ALCS treats an input message from a terminal as coming from the **standard input stream** (`stdin`) Application programs can access data from this message using the `gets` and `sscanf` C functions.

Application programs can send output to the originating terminal by using C functions that direct output to **standard output stream** (`stdout`). Applications can send messages to other terminals as described later. See "The routing control parameter list (RCPL) area" on page 82.

Entry storage on exit

The stack, the heap, the ECB, storage blocks and DECBs are all examples of entry storage. ALCS allocates entry storage automatically as and when your program requires it.

When an entry exits, ALCS releases all this entry storage so that other entries can use it. When ALCS releases the storage, any information stored there is lost. Of course, if your program has written this information to DASD, the DASD copy is not lost and can be read by another entry.

Swapping storage levels

Sometimes an application program might need to attach a storage block to a level where there is already an attached block. If there is an unused ECB storage level, then the application program can request the FLIPC (`flipc`) service to swap the contents of the two ECB storage levels.

The application program can then attach a new storage block at the chosen ECB level. Later, it can release the new storage block and use `flipc` again to restore the ECB storage levels to their original contents. For example, if an application needs to use level 0 (D0) that is already in use, but level 6 (D6) is not in use, the program could use `flipc` as follows:

```
flipc(D0,D6);
```

An application program can also swap the contents of an ECB storage level and a DECB storage level by requesting the `DECBC  FUNC=SWAPBLK` (`tpf_decb_swapblk`) service.

Detaching and attaching storage blocks

As an alternative to requesting the FLIPC (`flipc`) service, you can request the DETAC (`detac`) service. This detaches a block from a specified level but does not release the block. The program can then attach a new storage block at the detached level.

Later the application program can release the new storage block and request the ATTAC (`attac`) service to reattach the original storage block.

In the example in Figure 54 on page 87, an application needs to use ECB level 2 (D2), which is already in use. It uses DETAC to detach the block at level 2 (D2), and later uses ATTAC to reattach the block on the same level.

```
        MVC   EBW000(8),CE1FA2    SAVE FILE ADDRESS
        DETAC D2,ADDRESS          DETACH BLOCK AT LEVEL 2
⋮
        MVC   CE1FA2(8),EBW000    SET UP FILE ADDRESS
        ATTAC D2,FA               REATTACH BLOCK
```

Figure 54. Detaching and attaching a storage block - assembler

If an application needs to use ECB level 0 (D0), which is already in use, a C language program could use the `detac` and `attac` functions as shown in Figure 55 on page 87.

```
detac(D0);
/*   program uses level D0  */
⋮
attac(D0);
```

Figure 55. Detaching and attaching a storage block - C language

A block detached by DETAC (`detac`) service does not use a storage level, but ALCS still regards the block as belonging to the entry.

Storage block activity control

ALCS assumes certain storage block usage characteristics. If an application includes entries that use more than 10 or 15 storage blocks simultaneously, the application program might need to request the SLIMC (`slimc`) service to increase the amount of storage that the entry is allowed to use.

Note: The maximum number of blocks that you can use is system-dependent; see *ALCS Installation and Customization* for details. A C language application program might also need to call the `slimc` function if it uses a large amount of heap or stack storage.

It may also be necessary to change the activity control parameters. *ALCS Operation and Maintenance* describes how to do this.

ALCS services for handling storage blocks

ALCS provides the following assembler macros and C functions for handling storage blocks:

Assembler macro	C Function	Used to:
ATTAC	attac	Reattach a storage block after a DETAC.
DECBC FUNC=SWAPBLK	tpf_decb_swapblk	Swap the contents of an ECB storage level and a DECB storage level. DECBC FUNC=SWAPBLK does not exchange the contents of data levels and detail error indicators.
DETAC	detac	Detach the storage block from a level but do not release it.
FLIPC	flipc	Flip (exchange) the contents of two ECB storage levels. FLIPC also exchanges the contents of ECB data levels and detail error indicators.
GETCC	getcc	Get a storage block and attach it to a level.
levta	levtest	Test if a storage block is attached to a specified level.
RELCC	relcc	Detach a storage block from a level and release it.

Note: In addition to the above, some ALCS services either get and attach, or detach and release, a storage block as part of the requested service. For example, FINDC (`findc`) automatically gets and attaches a storage block and reads a DASD record into it.

Local save stack

An application program can request a SAVEC service to save and restore the contents of one or more of:

- The general purpose registers
- The floating point registers 0, 2, 4, 6
- ECB work area 1 (CE1WKA)
- ECB work area 2 (CE1WKB)
- ECB local program work area (CE1WKC).

SAVEC uses a last-in first-out (LIFO) stack, called the local save stack, to save the information. The local save stack uses entry storage and is "local" to your program. For full details of the SAVEC service see *ALCS Application Programming Reference - Assembler*.

Note: ALCS does not provide SAVEC support for C language programs.

TPF compatibility:

Do not use the SAVEC service in programs that must be compatible with TPF.

Data event control blocks (DECBs)

An application can use a data event control block (DECB) as an alternative to using an ECB level. An ECB level is used to identify a DASD record that is about to be read or written and to attach a storage block. Although a DECB does not physically reside in an ECB, the DECB fields specify the same information without requiring the use of a level in the ECB. The same requirements and conditions that apply to the fields in the ECB level also apply to the equivalent fields in the DECB.

DECB name: IDECNAM (idecnam)
Reserved
Storage level: IDECCRW (ideccrw) IDECDAD (idecdad), IDECCT0 (idecct0), IDECDLH (idecdlh)
Data level: IDECFRW (idecfrw) IDECRID (idecrid), IDECRCC (idecrcc), IDECFA (idecfa)
Error indicator: IDECSUD (idecsud)
Reserved
Detached block count: IDECDET (idecdet)
Data level extension: IDECFX0 (idecfx0)
User data: IDECUSR (idecusr)
Reserved for ALCS's use

Figure 56. Format of the ALCS DECB application area (schematic)

DECB fields

The DECB fields which concern the application programmer are described below. The data level and storage level, when used together, are called a **DECB level**.

DECB name (IDECNAM (idecnam))

Optionally when creating a DECB, the application program can give it a name. ALCS stores the name in this 16-byte field. If no name is given, this field contains binary zeros.

DECB storage level (IDECCRW (ideccrw))

The DECB storage level corresponds to an ECB storage level and contains the following information:

- 4-byte attached storage block address
- 2-byte block size indicator
- 2-byte block length.

IDECDAD (idecdad)

ALCS stores in this 4-byte (`long int`) field the address of the storage block that it allocates to the program.

IDECCT0 (idecct0)

ALCS stores in this 2-byte field the block size value. It contains the block type indicator value, L0, L1, L2, ...L8.

IDECDLH (idecdlh)

ALCS stores in this 2-byte field the length of the attached block.

For more details of the use of storage levels, see "Reading and writing DASD records" on page 38 and "Storage blocks" on page 85.

DECB data level (IDECFRW (idecfrw))

The DECB data level corresponds to an ECB data level and contains the following information:

- 2-byte record ID (optional)
- 1-byte record code check (RCC) character (optional)
- 8-byte file address (required).

Note: There is also a 1-byte reserved field in the data level.

IDECRID (idecrid)

When required (or allowed) by the DASD I/O operation, the application program stores the 2-byte record ID in this field.

IDECRCC (idecrcc)

Optionally for most DASD I/O operations, the application stores the record code check (RCC) in this 1-byte field.

IDECFA (idecfa)

For most DASD I/O operations, the application program stores the file address of the record it is reading or writing in this 8-byte (defined type TPF_FA8) field.

For more details of the use of data levels, see "Reading and writing DASD records" on page 38.

DECB error indicator (IDECSUD (idecsud))

This 1-byte field corresponds to an ECB detail error indicator. When a program requests a `WAITC` (`waitc`) service, ALCS might set this indicator.

For a description of the error indicators and how to test them, see "Error indicators in the ECB and DECB" on page 115.

DECB detached block count (IDECDET (idecdet))

ALCS stores the number of storage blocks detached from this DECB in this 2-byte field.

For more details about detaching and attaching storage blocks, see "Detaching and attaching storage blocks" on page 86.

DECB data level extension (IDECFX0 (idecfx0))

This 8-byte field corresponds to an ECB data level extension. It is used with the two macros (C functions) that process general data sets: GDSNC (gdsnc) and GDSRC (gdsrc).

DECB user area (IDECUSR (idecusr))

The application can store user data in this 8-byte field.

Managing DECBs

An application program can reference 16 storage blocks concurrently by using the storage levels of the ECB. DECBs enable an application to reference more than 16 storage blocks concurrently. An application program can dynamically acquire a DECB by requesting the DECBC FUNC=CREATE (tpf_decb_create) service. The number of DECBs that the application is restricted to is limited only by the amount of entry storage available to the ECB. An application program can release a DECB by requesting the DECBC FUNC=RELEASE (tpf_decb_release) service.

The first time the DECBC FUNC=CREATE (tpf_decb_create) service is requested for an entry, ALCS allocates a new overflow storage unit and reserves the first section of that storage unit for use as part of a DECB frame. The remaining space in this overflow storage unit is available to satisfy storage block requests.

A single DECB frame will contain a header and multiple DECBs. Each DECB is comprised of an application area that is accessible to application programs and a system area that is accessible only to ALCS. The application area resides in the overflow storage unit. The frame header and system areas reside in the trap page preceding the overflow storage unit and are referred to as the DECB descriptor.

It is possible for an entry to have more than one DECB frame. ALCS will allocate a new overflow storage unit for each DECB frame.

An application program can release a DECB by requesting the DECBC FUNC=RELEASE (tpf_decb_release) service. When the final DECB is released by the application, ALCS leaves the DECB frame attached to the ECB in preparation for creating the next DECB. ALCS releases the DECB frame when the entry exits.

Using symbolic names

An application program can associate a symbolic name with each DECB. This allows different components of an application to easily pass information in storage blocks attached to a DECB. Each component only needs to know the name of the DECB to access it. However ALCS services that support the use of a DECB (such as FILEC, FINDC, file_record_ext, find_record_ext, and so on) will only accept a DECB address as a valid reference to a DECB. If an application does not maintain the address of a particular DECB and, instead, maintains the name of the DECB, the caller must first request the DECBC FUNC=LOCATE (tpf_decb_locate) service to obtain the address of the DECB. The DECB address can then be passed on to the subsequent ALCS service call.

See the *ALCS Application Programming Reference - Assembler* for more information about the DECBC service and the *ALCS Application Programming Reference - C Language* for more information about the tpf_decb_locate function.

ALCS services for managing DECBs

ALCS provides the following assembler macro and C functions for managing DECBs.

Note: ALCS C/C++ applications that use DECBs must be compiled with the C++ compiler.

Assembler macro	C Function	Used to
DECBC FUNC=CREATE	tpf_decb_create	Create a DECB.
DECBC FUNC=LOCATE	tpf_decb_locate	Locate a DECB.

Assembler macro	C Function	Used to
DECBC FUNC=RELEASE	tpf_decb_release	Release a DECB.
DECBC FUNC=SWAPBLK	tpf_decb_swapblk	Swap the contents of an ECB storage level and a DECB storage level. Does not exchange the contents of the data level and detail error indicator.
DECBC FUNC=VALIDATE	tpf_decb_validate	Validate a DECB.

8-byte file address support

Although an ECB level and a DECB level are alike, there is a difference in the data level. The IDECFA (idecfa) field in the DECB, which contains the file address, is 8 bytes in length.

TPF compatibility:

The 8-byte file address is required for compatibility with TPF which allows 8-byte file addressing in either 4x4 format or FARF6 format. In ALCS, if you want to use a DECB, you must use an 8-byte file address in 4x4 format only.

The 4x4 format enables a standard 4-byte file address to be stored in an 8-byte field. In 4x4 format the low-order 4 bytes of the IDECFA (idecfa) field contain an ALCS band format file address. The high-order 4 bytes of the IDECFA (idecfa) field contain an indicator (a full-word of zeros) that classifies it as a valid 4x4 format file address.

TPF compatibility:

In TPF the high-order 4 bytes of a file address in FARF6 format contain a non-zero value.

An application may request the FAC8C (tpf_fac8c), GETFC (getfc), GDSNC (gdsnc) or GDSRC (gdsrc) services to obtain the 8-byte file address in 4x4 format of the record it wants to read or write.

An application may also request the FA4X4C (tpf_fa4x4c) service to convert a 4-byte file address to an 8-byte file address in 4x4 format, or to convert an 8-byte file address in 4x4 format to a 4-byte file address.

When there is DASD I/O activity the data level must contain an 8-byte file address and may also contain a record ID (2 bytes) and record code check (1 byte). One byte is unused.

Accessing DASD records and using storage blocks with a DECB

Application programs can access DASD records and reference storage blocks using a DECB instead of an ECB data level. However only a subset of the assembler macros and C functions that reference ECB levels accept a DECB address in place of an ECB level.

Applications can request the following assembler macros and C functions specifying a DECB address or an 8-byte file address in place of an ECB level. When these assembler macros and C functions are used, and they reference the specified file address or the file address in the data level of the specified DECB, ALCS will first verify that it is a valid 8-byte file address in 4x4 format.

Note: ALCS C/C++ applications that request any of the following C functions using DECBs must be compiled with the C++ compiler.

Assembler macro	C Function	Used to
ATTAC	attac_ext attac_id	Reattach a storage block after a DETAC
CRUSA	crusa	Release a storage block from specified ECB levels or DECB
DETAC	detac_ext detac_id	Detach the storage block from a level but do not release it

Assembler macro	C Function	Used to
FILEC		File (write) a DASD record
FILNC		File (write) a DASD record without detaching the storage block
FILSC		File (write) a single copy of a DASD record
FILUC		File (write) a held DASD record and unhold the file address
	file_record_ext	File (write) a DASD record with extended options
FINDC		Find (read) a DASD record
FINHC		Find (read) a DASD record and hold the file address
FINSC		Find (read) a single copy of a DASD record
FINWC		Find (read) a DASD record
FIWHC		Find (read) a DASD record and hold the file address
	find_record_ext	Find (read) a DASD record with extended options
GDSNC	gdsnc	Open or close a general data set
GDSRC	gdsrc	Calculate the file address of a general data set record
GETCC	getcc	Get a storage block and attach it to an ECB level or DECB
GETFC	getfc getfc_alt	Get a pool file address (dispensed by ALCS) and optionally get and attach a storage block
levta	levtest	Test if a storage block is attached to a specified ECB level or DECB
LISTC		Generate a storage list for SNAPC or SERRC
RCRFC	tpf_rcrfc	Release a pool file record address and storage block
RCUNC	rcunc	Release a storage block and unhold a file address
RELCC	relcc	Detach a storage block from an ECB level or DECB and release it
RELFC	relfc	Release a pool file address, and optionally detach and release the storage block
SONIC	sonic	Extract information about a record
UNFRC	unfrc_ext	Unhold a held DASD record

Notes:

1. You can request the FAC8C (tpf_fac8c) service to obtain the 8-byte file address in 4x4 format for a fixed-file record.
2. You can request the FA4X4C (tpf_fa4x4c) service to convert a 4-byte file address to an 8-byte file address in 4x4 format, or to convert an 8-byte file address in 4x4 format to a 4-byte file address.

Error handling

Each DECB has a detail error indicator, IDECSUD (idecsud). The gross error indicator CE1SUG (ce1sug) in the ECB will include any errors that occur on a DECB-related DASD I/O operation.

For more information about the error indicator and how to test it, see “Error indicators in the ECB and DECB” on page 115.

Accessing a DECB

Assembler programs access a DECB using the DSECT IDECB.

The C data structure describing the DECB is defined as type TPF_DECB, and is defined in the header file <c$decb.h>.

Functional flow

About this task

The DECB acts as an interface between the application and ALCS, and contains relevant information about storage block and DASD I/O requests. The following example shows DECBs being used by a sample application.

When an airline passenger purchases a one-way ticket to Bermuda, the sample application is activated to update a database that maintains a list of all passengers with one-way flights.

The application consists of 2 programs, PGMA and PGMB. Program PGMA will validate the database update request and verify that the passenger record does not already exist in the current passenger list. Once the request has been validated, PGMA will forward the request to program PGMB which will add the name of the new passenger to the current passenger list.

Procedure

1. PGMA is entered from a reservation application with a copy of the passenger record attached on data level 0 (D0) of the ECB. After verifying that there is a passenger record attached to D0, PGMA allocates a DECB which will be used to hold the primary database record for Bermuda. The following examples show the request to allocate a DECB.

```
         IDECB REG=R01             DATA EVENT CONTROL BLOCK
         DECBC FUNC=CREATE,        CREATE PRIMARY DECB        X
               DECB=(R01),                                    X
               NAME=DECBPRIM
           ⋮
DECBPRIM DC    CL16'BERMUDA.PRI'
```

Figure 57. Allocating a DECB - assembler

```
  TPF_DECB *decb;
  DECBC_RC rc;
    ⋮
  decb = tpf_decb_create ("BERMUDA.PRI", &rc);
```

Figure 58. Allocating a DECB - C++ language

   There were no DECBs previously allocated to this entry, therefore ALCS allocates a new overflow storage unit to contain the DECB frame.
2. Now that a DECB is available for use, program PGMA attempts to obtain the file address of the primary record for the Bermuda database. The following examples show a request for the FAC8C (`tpf_fac8c`) service to compute the correct file address.

```
         IFAC8 REG=R04             FAC8C PARAMETER BLOCK
         LA    R04,EBX000
         MVC   IFACORD,=XL8'150'   ORDINAL
         MVC   IFACREC,=CL8'#BERMUDA' RECORD TYPE SYMBOL
         MVI   IFACTYP,IFACFCS     FACS TYPE CALL
         FAC8C PARMS=(R04)         COMPUTE FILE ADDRESS
```

Figure 59. Computing an 8-byte file address - assembler

```
  TPF_FAC8 *fac8_parms;
    :
  fac8_parms          = (TPF_FAC8 *)ecbptr()->ebx000;
  fac8_parms->ifacord = 0x150;
  fac8_parms->ifactyp = IFAC8FCS;
  memcpy (fac8_parms->ifacrec, "#BERMUDA", 8);
  tpf_fac8c (fac8_parms);
```

Figure 60. Computing an 8-byte file address - C++ language

3. PGMA now has the file address for the primary record and issues a FIND request to bring the record into working storage and obtain a lock on the record for this ECB. The following examples show a request for a FIND-type service.

```
         MVC   IDECFA,IFACADR      FILE ADDRESS
         MVC   IDECRID,=CL2'AB'    RECORD ID
         XC    IDECRCC,IDECRCC     RCC
         FIWHC DECB=(R01),ERROR=ERROR_BRANCH   READ RECORD
```

Figure 61. Requesting a FIND-type service using a DECB - assembler

```
  decb->idecfa = fac8_parms->ifacadr;
  find_record_ext (decb, NULL, "AB", '\0',
                   HOLD_WAIT, FIND_DEFEXT);
```

Figure 62. Requesting a FIND-type service using a DECB - C++ language

4. When the FIND request ends either successfully or unsuccessfully, ALCS updates the DECB accordingly. For a successful call, ALCS attaches a storage block at the DECB storage level.
5. Now that the primary record has been read from the database, PGMA enters program PGMB to complete the processing. PGMB searches the primary record for an available entry slot to which the new passenger record can be added. If there are no available entries in the primary record, an overflow record is obtained. To obtain the overflow record a new DECB must be created. The following examples show a request to create a new DECB.

```
         DECBC FUNC=CREATE,        CREATE OVERFLOW DECB       X
               DECB=(R02),                                    X
               NAME=DECBOFLW
           :
DECBPRIM DC    CL16'BERMUDA.PRI'
DECBOFLW DC    CL16'BERMUDA.OVR'
```

Figure 63. Creating a new DECB - assembler

```
  TPF_DECB *prime, *overflow;
  DECBC_RC rc;
    :
  overflow = tpf_decb_create ("BERMUDA.OVR", &rc);
```

Figure 64. Creating a new DECB - C++ language

ALCS allocates the first available DECB in the DECB frame to the entry. There are now two DECBs in the single DECB frame, which are marked as in use by the ECB.

6. Program PGMB must obtain a pool-file record to use as the new overflow record in the database. PGMB must locate the DECB created by PGMA (which contains the primary record) so that the pool-file record can be chained to it. The following examples show the requests to obtain a pool-file record address and locate the DECB containing the primary record.

```
         GETFC DECB=(R02),         GET POOL-FILE ADDRESS      X
               ID=C'AB',                                      X
               BLOCK=YES
         DECBC FUNC=LOCATE,        LOCATE PRIMARY DECB        X
               DECB=(R03),                                    X
               NAME=DECBPRIM
           ⋮
DECBPRIM DC    CL16'BERMUDA.PRI'
DECBOFLW DC    CL16'BERMUDA.OVR'
```

Figure 65. Locating a DECB - assembler

```
  TPF_FA8 pool_addr;
    ⋮
  pool_addr = getfc (overflow, GETFC_TYPE0, "AB",
                     GETFC_BLOCK, GETFC_SERRC);
  prime = tpf_decb_locate ("BERMUDA.PRI", &rc);
```

Figure 66. Locating a DECB - C++ language

The GETFC (`getfc`) request specifies that a storage block is required. ALCS attaches the storage block at the DECB storage level for the overflow record.

7. Program PGMB must now copy the relevant information from the passenger record of the new passenger, which is attached on ECB data level 0 (D0), to an available entry in the overflow record attached to the DECB. After copying the data a FILE request must be issued to update the database with the details of the new passenger. The following examples show the request for the FILE-type service.

```
         FILEC DECB=(R02)          WRITE RECORD
```

Figure 67. Requesting a FILE-type service using a DECB - assembler

```
  file_record_ext (overflow, NULL, "AB", '\0',
                   NOHOLD, FILE_DEFEXT);
```

Figure 68. Requesting a FILE-type service using a DECB - C++ language

8. Now that the overflow record is no longer being referenced by the ECB, program PGMB may choose to release the DECB that was allocated to contain the record. The following examples show a request to release the DECB.

```
         DECBC FUNC=RELEASE,       RELEASE DECB               X
               DECB=(R02)
```

Figure 69. Releasing a DECB - assembler

```
  tpf_decb_release (overflow);
```

Figure 70. Releasing a DECB - C++ language

9. Program PGMB can now update the primary record with the file address of the overflow record to chain them together. After performing this update, PGMB can now issue a FILE request to update the primary record in the database and release the lock, which program PGMA had previously obtained on the record. The following examples show the request for the FILE-type service.

```
         FILUC DECB=(R03)          WRITE AND UNHOLD RECORD
```

Figure 71. Requesting a FILE-type service using a DECB - assembler

```
  file_record_ext (prime, NULL, "AB", '\0',
                   UNHOLD, FILE_DEFEXT);
```

Figure 72. Requesting a FILE-type service using a DECB - C++ language

10. Program PGMB may now choose to release the final DECB before it exits. If it does not, ALCS releases the DECB frame when the entry exits.

Chapter 5. Application programming considerations

This chapter describes:

- Reentrant requirements of application programs
- Linking between programs
- Creating new entries
- Events.

Reentrant requirements of application programs

One ALCS application program can process more than one entry, often simultaneously. For example, several end users can request a time display at the same time. Each request (input message) is a separate entry, but there is only one program that generates the time display response (output message).

When one application program processes several entries, ALCS uses the same copy of the program for all entries. Because of this, ALCS application programs must be **reentrant**. A reentrant program does not modify itself; for example, it does not contain work areas or switches that it modifies during execution.

A reentrant application program makes use of **entry storage** (of which the ECB is an important part). Entry storage is described in Chapter 4, “The entry control block (ECB) and other entry storage,” on page 75.

Links between programs

Application programs running under ALCS can be up to 32KB in size (high-level language (HLL) programs can be any size). However, it is usually convenient to process an entry by a sequence of smaller programs which transfer control from one to another (or in the case of HLL programs, use common functions).

TPF compatibility:

If your programs must be compatible with TPF, you must restrict the size of each program to a maximum of 4KB.

Transferring control between programs

C language and COBOL programs can call other programs using facilities of the language (for example CALLs in COBOL). See Chapter 8, “Assembling, compiling, and link-editing application programs,” on page 145 for a description of this in a ALCS environment.

ALCS provides services that allow an application to transfer control from one program to another. Request the ENTDC (`entdc`) service when you want to transfer control and do not expect the called program to return to the calling program; request the ENTRC (`entrc`) service when you expect the program to return.

When a program enters another program, the ECB and other entry storage is available to the entered program, as shown in Figure 73 on page 98.

© Copyright IBM Corp. 2003, 2019

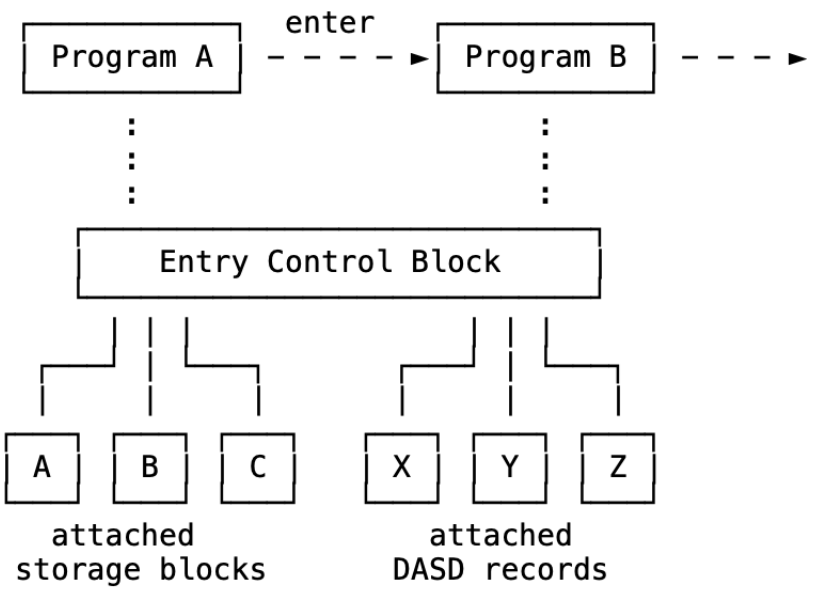


Figure 73. Program entering another program

1. Program A enters Program B by requesting the ENTRC (`entrc`) or other ALCS enter service.
2. The ECB, and any attached storage blocks or DASD records are accessible to both programs (and to any other programs that are entered).

Program nesting levels

The ENTRC (`entrc`) service saves return control data in an ECB area called a **program nesting level**. It then transfers control to the other application program. Later, a BACKC macro (`return` C function) can use this data to return to the next instruction following the ENTRC (`entrc`) request.

The ENTNC macro (for which there is no corresponding C function) allows a program to enter another program without an expected return to the calling program.

ALCS supports up to 32 program nesting levels (numbered 0 through 31), so 32 sets of return control data, saved by ENTRC (`entrc`), can exist at any one time. That is, an entry can request a maximum of 32 ENTRC (`entrc`) services without calling BACKC (`return`).

Figure 74 on page 99 illustrates the ENTER/BACK mechanism in assembler programs and Figure 75 on page 99 shows the same (slightly more restricted) mechanism in C programs. (Note that the names used in ENTRC and similar service requests are actually 4 characters long, as explained in “Naming conventions” on page 146.)

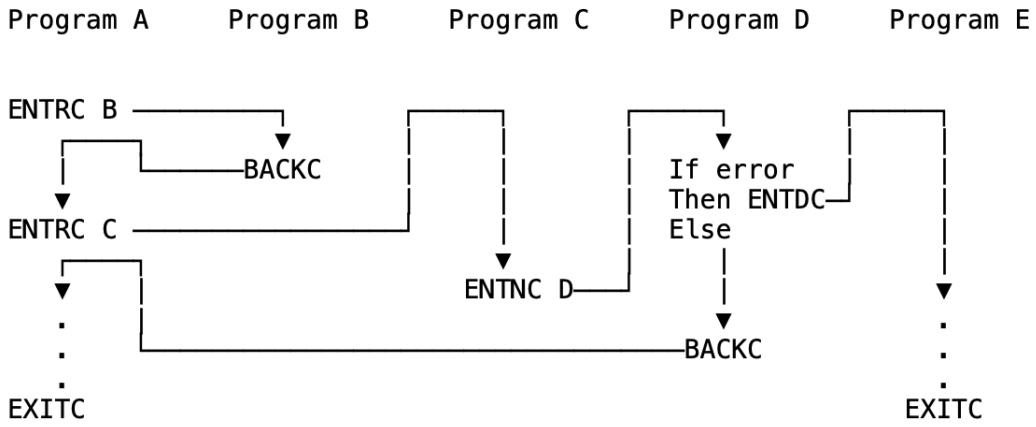


Figure 74. The ENTER/BACK mechanism in assembler programs

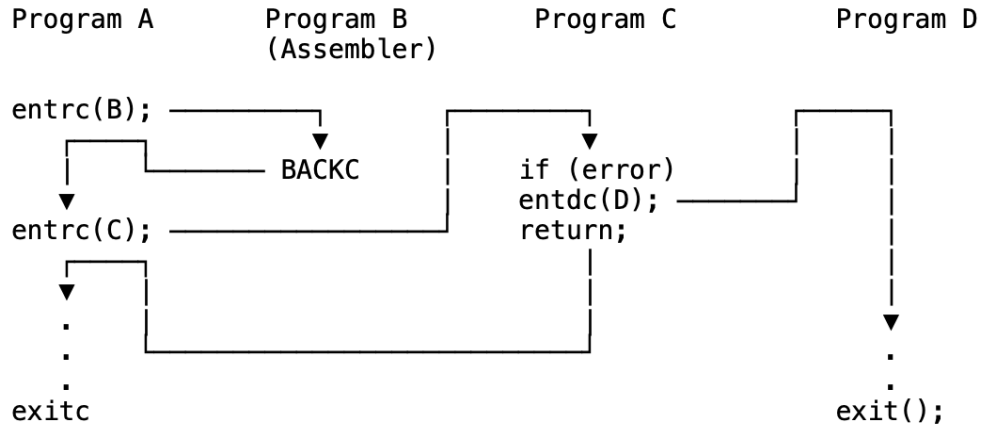


Figure 75. The ENTER/BACK mechanism in C programs

ALCS services for transfer of control

ALCS provides the following assembler macros and C functions to enable transfer of control between programs.

Assembler macro	C Function	Used to:
BACKC	`return`	Returns control to the next sequential instruction in the program that last requested an ENTRC (`entrc`) service. If there is no such program, this is equivalent to an EXITC (`exit`) request. (See below for a description of EXITC (`exit`).)
ENTDC	`entdc`	Transfers control to another program when control is not to be returned to the issuing or any previous program.
ENTNC		Transfers control to another application program when control is not to be returned to the issuing program.
ENTRC	`entrc`	Transfers control to another program when control is to be returned by BACKC or `return` to the issuing program.

Assembler macro	C Function	Used to:
EXITC	exit	Terminates processing of an entry, releases all entry storage, and returns control to ALCS
FINPC		Gets the address of an application program
WHOCC		Identifies a program that called this program (you can specify an absolute or relative nesting level)

ALCS provides the following utility services for use with assembler application programs:

ENTRC CAP1
Scan data base records.

ENTRC CCNV
Convert between binary and character representation of data.

ENTDC CFMS
Process ALCS commands.

ENTRC CFMT
Format the complete display buffer of a 3270 screen.

ENTRC CONF
Request command confirmation.

ENTRC CPL3
Start an ALCS application.

ENTRC CPL4
Stop an ALCS application.

ENTRC CQS7
Check message queues for a communication resource.

ENTRC CSMI
Convert the input data stream received from a 3270 display terminal into the format described by a map DSECT.

ENTRC CSMO
Convert the data supplied in a format described by a map DSECT into an output data stream for a 3270 display terminal.

ENTRC CSMS
Send an e-mail message using SMTP.

ENTRC CVEP
Check the availability of pool based on a pool interval number.

ENTRC CVEQ
Check the availability of pool based on a record ID.

ENTRC CXA0
Generate an output message that conforms to the ALCS numbered message format.

ENTRC FACE
Calculate the file address of a fixed-file record from the fixed-file record type number and the ordinal number of the record.

ENTRC FACS
Calculate the file address of a fixed-file record from the fixed-file record type name and the ordinal number of the record.

Two C functions (`face` and `facs`) provide the equivalent of the callable services FACE and FACS. C language programs use standard C library functions in place of the other callable services.

Note: When a C language program is activated through an `entrc` or `entdc` function, ALCS creates a new C environment. This environment contains storage for stack, heap, `static`, and `extern` data.

ALCS releases this storage when the application exits the environment using the `return`, `exit`, or `entrc` C function. Do not pass references to this storage back to the calling program.

Creating new entries

An entry is an item of processing (unit of work) that can proceed independently of other entries. Typically, an entry is processed by a series of application programs that transfer control to each other, using the program linkage macros (C functions) described in "Transferring control between programs" on page 97.

When an application program is processing an entry, there might be associated items of processing which can be considered additional to the main objective of the entry. For example, an entry might need to send an informative message to several other terminals in addition to a normal response message. These associated items of work can be processed using independent entries.

In this case, an application program requests an ALCS service to create a new entry. The newly-created entry operates with its own ECB (separate from the original ECB). See Figure 76 on page 101.

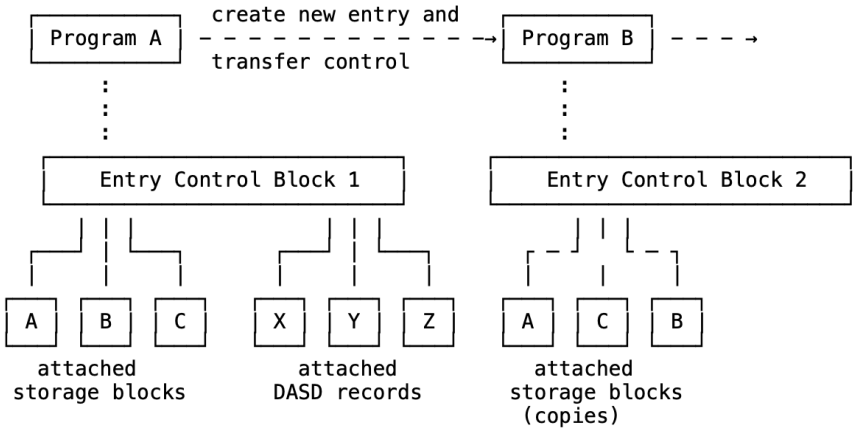


Figure 76. An application program creates a new entry

1. Program A creates another entry by requesting an ALCS service (CREEC (`creec`), CREMC (`cremc`), and so on). ALCS creates a new ECB.
2. Program A must specify the program (Program B) that is to process the new entry. The new ECB is accessible only to Program B.
3. The first ECB, and any attached storage blocks or DASD records are not accessible to Program B and the second ECB is not accessible to Program A. However, if Program A has requested CREDC (`credc`) or CREMC (`cremc`) to create the new entry, it can request ALCS to attach **copies** of the storage blocks to the new ECB. CREEC (`creec`) can request ALCS to transfer a copy of only 1 storage block.

   CREDC (`credc`) and CREMC (`cremc`) services can pass up to 112 bytes of data. The CRETC (`cretc`) service can pass 4 bytes of data. ALCS puts this data into the work area of the new ECB.

   TPF compatibility:

   In TPF:

   - Only the CREEC (`creec`) service can transfer a storage block to the created entry.
   - The CREDC (`credc`) and CREMC (`cremc`) services only allow you to pass a maximum of 96 bytes of data.

Priority of created entries

The CREMC (cremc) and CRETC (cretc) services give the newly created entry the same priority as the creating entry. The CREMC (cremc) service creates the new entry immediately, the CRETC (cretc) service creates the new entry after a specified time delay.

The CREDC (credc) service creates the new entry immediately, and gives it lower priority than new input messages.

After a create service request, ALCS normally returns control immediately to the application program that requested it. However, to create a new entry, ALCS must allocate storage for the new ECB. Extensive use of create services can use up available storage.

To reduce the impact of this, ALCS suspends processing of an entry that calls a create service when there is not enough storage available. So a create service (but not CRETC (cretc)) can cause the entry that calls the service to lose control. This also delays the creation and eventual processing of the new entry. In the case of CRETC (cretc), ALCS only suspends creation of the new entry.

Use the CRETC (cretc) service to do the same processing repeatedly at regular intervals. After doing the processing once, the application program can request a CRETC (cretc) service to create a new entry, to be processed by the same program, after a fixed interval. The program can then terminate the current entry by requesting an EXIT (exit) service. After the interval, ALCS creates the new entry and the process repeats. This type of process is called **time-initiated**.

Note: This process is quite different from the IPARS application function of creating "time-initiated" entries from a time-initated function table.

Transactions that create multiple entries

Many transactions can be processed using a single entry - they do not need to call create services. However, you might find it more efficient to implement some complex transactions by creating a number of entries that can process in parallel.

For these transactions, the original entry (the "parent" entry) creates a number of "child" entries, which can run parallel with each other and with the parent entry. The child entries can in turn create more "grandchild" entries, and so on.

Note: Attention

Application programs that create child and grandchild entries in this way can efficiently exploit ALCS's multiprogramming and multiprocessing capabilities - but these types of program can cause serious problems unless they are carefully designed. When designing new programs or installing purchased programs of this type, note the following:

- In most ALCS configurations, ALCS services create requests before it starts processing new input messages. This is to ensure that current transactions complete even when there is a high input message rate.

  Because of this, a parent entry that creates many child and grandchild entries can flood ALCS with entries, and so lock out new transactions. Application programs can avoid this by regulating their use of create services.

  To avoid creating so many entries that performance is degraded, you can call the LODIC (lodic) service before each create service. LODIC (lodic) can tell you if creating the new entry would degrade performance.
- If a parent entry is holding a record or other resource (or has a general sequential file assigned) while it is looping creating child entries, it may lose control (see “The ALCS wait mechanism and error processing” on page 111) because there are already too many entries in the system.

  Generally, it is advisable for application programs to unhold records or other resources (or reserve general sequential files) before calling create services.

ALCS services for creating entries

ALCS provides the following assembler macros and C functions for creating new entries:

Assembler macro	C Function	Used to:
CREDC	credc	Create a new entry for deferred scheduling. Pass up to 112 bytes of data and the contents of one or more storage blocks.
CREEC	creec	Create a new entry and pass the contents of one storage block
CREMC	cremc	Create a new entry for immediate scheduling. Pass up to 112 bytes of data and the contents of one or more storage blocks.
CRETC	cretc	Create a new entry for deferred scheduling. Pass 4 bytes of data to the new ECB.
CREXC	crexc	Create an entry for deferred scheduling (for TPF compatibility only)

Events

When a parent entry creates one or more child entries it might need to suspend itself until all the child entries have completed processing. It can do this by requesting ALCS **event** services. Currently these services are available only as assembler macros.

Counter-type events

In a counter-type event, the parent entry sets a counter, typically to the number of child entries it has created. Each child entry requests a POSTC service when it has finished processing. When a child entry requests POSTC, ALCS reduces the counter by 1. When the counter reaches zero, ALCS restarts the suspended parent entry. See Figure 77 on page 104.

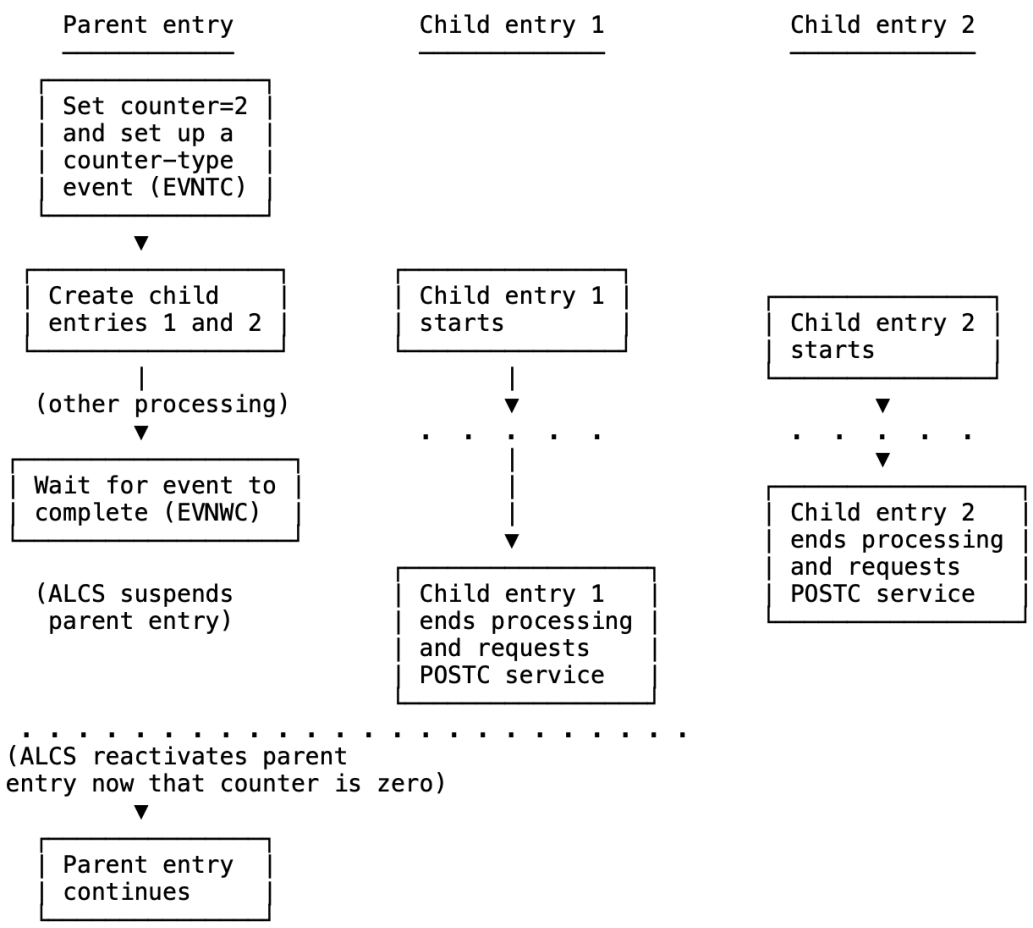


Figure 77. How parent and child entries use a counter-type event

Other types of event in ALCS

In addition to counter-type events, applications can use two other types of event. Mask-type events allow both the parent and child entries to set bits in a mask (to indicate various conditions). Storage block-type events allow the (single) child entry to return the contents of a storage block to the parent entry.

As with counter-type events, the parent entry sets up the event by requesting an EVNTC service, then suspends itself by requesting an EVNWC service. Child entries can request POSTC services. ALCS returns control to the parent entry when the event has completed.

Table 8 on page 104 describes how the 3 types of event are set up and used in ALCS.

Table 8. How the 3 types of event are used in ALCS

Type of event	Action by parent entry	Action by child entry	ALCS returns control to parent entry
Counter-type	Sets counter before requesting EVNTC	Each POSTC request by a child entry reduces the counter by 1	When counter is zero
Mask-type	Sets up mask before requesting EVNTC	Set or unset bits in the mask before each POSTC request	When mask becomes all zeros

Table 8. How the 3 types of event are used in ALCS (continued)			
Type of event	**Action by parent entry**	**Action by child entry**	**ALCS returns control to parent entry**
Storage-block type	No special action before requesting EVNTC	Attaches a storage block before requesting POSTC	After the first POSTC request

In addition, ALCS always returns control to the parent entry (with an error condition) after a timeout period. This is to ensure that the parent entry is not suspended indefinitely.

Use of the ECB levels in event processing

Normally, the parent application sets up the 8-character event name in a data level field in the ECB. When the event completes, ALCS returns information about the event in the corresponding storage level. (The storage level must not have an attached storage block when the parent application requests EVNTC.) See Figure 78 on page 105.

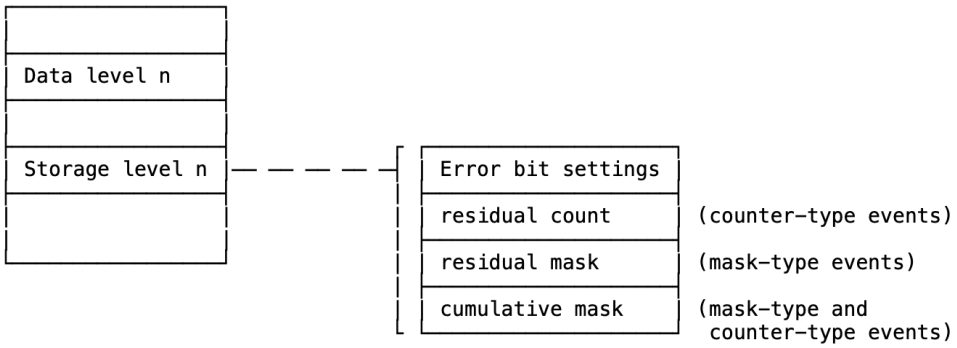


Figure 78. How events use ECB levels

With storage block-type events, the child entry provides an attached storage block with the POSTC service request. (If the storage block event completes on a timeout, ALCS provides an error-reporting storage block as shown in Figure 78 on page 105.)

Using a resource control block with events

Instead of using ECB levels, applications can can use a **resource control block** to store information about events. See the event control macro descriptions in the *ALCS Application Programming Reference - Assembler* for a description of how to do this.

Summary of event services

In addition to the services already described, ALCS allows individual entries to increment the count on counter-type events and to query the state of an event. The full list of ALCS services is shown in Table 9 on page 105.

Table 9. Event services	
Assembler macro	**Used to**
EVINC	Increment the count associated with a counter-type event
EVNQC	Interrogate the status of an event
EVNTC	Define an event

Table 9. Event services (continued)	
Assembler macro	**Used to**
EVNWC	Wait for an event to complete
POSTC	Complete (or part-complete) an event

Condition code and program mask

ALCS uses a 1-byte field in the ECB to manage the condition code and program mask for an entry.

You must not attempt to change the program mask directly, either by using the SET PROGRAM MASK (SPM) assembler instruction or by updating ECB field CE1ECC. Normally the program mask is all zeros; changing this can result in ALCS system errors.

You can use the CC macro to set the condition code to a value which can be tested in a later part of the application program. CC is described in *ALCS Application Programming Reference - Assembler*.

Floating-point registers

ALCS does not preserve the AFP (additional floating-point) registers to avoid the overhead of saving and restoring them as they are not used in ALCS applications.

Only the floating-point registers 0, 2, 4, 6 can be used in assembler programs.

The C compiler FLOAT(NOAFP) option for C/C++ programs must be used to limit the compiler to generating code using only the original four floating-point registers 0, 2, 4, and 6.

Calling other functions and programs

About this task

The following information explains how to call other functions and programs in your applications.

Procedure

1. Calling a C program (from assembler)
2. Calling an assembler program (from C)

Calling a C program (from assembler)

There can be times when you want to call a segment written in C from a segment written in assembler. Use the CALLC macro if the CSO that is being called passes parameters.

You can continue to enter a C/C++ program with the ENTDC, ENTNC, or ENTRC macro and use traditional TPF 4.1 parameter passing when you are migrating to the z/TPF system as long as you code a CPROC macro with the correct C/C++ function definitions. Add the CPROC statement to the assembler program directly; it applies to all of the matching CALLC statements that follow. Alternatively, you can add the CPROC statement to the UPROC macro to avoid updating the assembler program.

Note: The preceding information applies to C functions and only those C++ functions that use extern "C" linkage.

See z/TPF General Services for more information about the CALLC, CPROC, ENTDC, ENTNC, ENTRC, and UPROC macros.

Calling an assembler program (from C)

A special protocol is required when C language programs call basic assembler language (BAL) programs because the conventions to pass parameters are different. This protocol is used to pass parameters to the

BAL program and receive the return values. BAL programs run in 31- or 64-bit addressing mode and use registers (R0-R7) to receive parameters and return values.

TPF_regs structure

The prototype statement that declares a BAL program call from a C language program must contain one parameter, which is a pointer to the TPF_regs structure. The return data type must be void.

For example, the following shows the prototype statement when BAL program ABCD is called and instructs the compiler to check the data type of the parameter and not to expect a return value:

```
void ABCD(struct TPF_regs *);
```

The TPF_regs structure must be included in the prototype statement even if the BAL program that is called does not receive any parameters (in registers) or return any values (in registers). For this, the calling C language program must pass NULL instead of a pointer to a TPF_regs structure or errors can result.

Note: Attention: Because BAL programs do not recognize how values are returned to a C function, the results of declaring a return data type other than void cannot be predicted.

The following example shows how the TPF_regs structure is defined in tpf/tpfregs.h. Registers r0┘r7 are 8 bytes each and correspond to registers R0┘R7. The tr_mode field indicates the mode in which TPF_regs is operating.

```
struct  TPF_regs {
      long    tr_mode;
      long    r0;
      long    r1;
      long    r2;
      long    r3;
      long    r4;
      long    r5;
      long    r6;
      long    r7;
  };       Parent
```

TPF_regs structure

The prototype statement that declares a BAL program call from a C language program must contain one parameter, which is a pointer to the TPF_regs structure. The return data type must be void.

For example, the following shows the prototype statement when BAL program ABCD is called and instructs the compiler to check the data type of the parameter and not to expect a return value:

```
void ABCD(struct TPF_regs *);
```

The TPF_regs structure must be included in the prototype statement even if the BAL program that is called does not receive any parameters (in registers) or return any values (in registers). For this, the calling C language program must pass NULL instead of a pointer to a TPF_regs structure or errors can result.

Note: Attention: Because BAL programs do not recognize how values are returned to a C function, the results of declaring a return data type other than void cannot be predicted.

The following example shows how the TPF_regs structure is defined in tpf/tpfregs.h. Registers r0┘r7 are 8 bytes each and correspond to registers R0┘R7. The tr_mode field indicates the mode in which TPF_regs is operating.

```
struct  TPF_regs {
      long    tr_mode;
      long    r0;
```

```
      long    r1;
      long    r2;
      long    r3;
      long    r4;
      long    r5;
      long    r6;
      long    r7;
  };
```

TPF_regs in 64-bit addressing mode

To use TPF_regs in 64-bit addressing mode, a pointer to the TPF_regs structure must be passed as a parameter to the TPF_regs_Call64 function. The TPF_regs_Call64 function must be coded instead of TPF_regs in the parameter list of the BAL program. The prototype for the TPF_regs_Call64 function is defined in tpf/tpfregs.h.

When a C language program calls a BAL program, the linkage takes the data from fields r0┘ r7 in the TPF_regs structure and loads it in R0┘ R7. All 8 bytes for each field in the range r0┘ r7 are loaded into the corresponding registers, R0┘ R7.

When a BAL program returns to the C language program and TPF_regs is in 64-bit addressing mode, the linkage processes the structure as signed 8-byte values. The contents of the registers R0┘ R7 are not changed and are saved in the TPF_regs structure.

Because a TPF_regs structure in 31-bit addressing mode can be allocated by using the same area in memory that was used previously by a TPF_regs structure in 64-bit addressing mode, the program linkage automatically resets TPF_regs to 31-bit addressing mode. This prevents a 31-bit version of TPF_regs from being put into 64-bit addressing mode accidentally. Use the TPF_regs_Call64 function every time you want to call a BAL program using TPF_regs in 64-bit addressing mode.

The following example shows that the prototype for the TPF_regs_Call64 function defined in tpf/tpfregs.h. static inline struct TPF_regs *TPF_regs_Call64(struct TPF_regs *);The following example shows the function call coded to call BAL program ABCD by using TPF_regs in 64-bit addressing mode. ABCD(TPF_regs_Call64 (regs));The following example shows the TPF_regs structure being used in 64-bit addressing mode:

```
void ABCD(struct TPF_regs *);

int myfunction(short *s, unsigned int *ui, char *c)
{
   struct TPF_regs regs;      /* Define TPF_regs variable */
   int lc;                    /* loop counter */

   for(lc=0; lc<=100; lc++)   {
      regs.r0 = 0;           /* assign integer constant 0 to R0*/
      regs.r1 = *s;          /* assign short to R1*/
      regs.r2 = *ui;         /* assign unsigned int to R2*/
      regs.r3 = *c;          /* assign char to R3*/

      ABCD(TPF_regs_Call64 (regs));  /* Call assembler program using*/
                                     /* TPF_regs in 64-bit mode*/

      /* returns signed int, but linkage propagates sign bit, */
      /* so no cast required
      if (regs.r0 == -1) return 0;

      /* cast to signed short */
      if ((short)regs.r1 == *s) return 1;

      /* cast to unsigned int */
      if ((unsigned int)regs.r2 == *ui) return 2;

      /* '*c' is a character, so cast regs.r3 */
      if ((char)regs.r3 == *c) return 3;

     } /* end for loop */

      return -1;
  }
```

TPF_regs programming considerations

Use casting to avoid compiler warnings when comparing, saving, or retrieving pointers from the TPF_regs structure.

When storing a variable or constant into TPF_regs.rx that is less than 8 bytes, the compiler automatically takes the value and expands it to an 8-byte long data type before storing it in the TPF_regs structure. If the variable is a signed or a constant value, the compiler expands the value to a signed long. If the variable is unsigned (char, unsigned char, unsigned short, or unsigned int), the compiler expands it to an unsigned long.

When storing the contents of TPF_regs.rx into another variable, the compiler automatically truncates the value in TPF_regs.rx to fit in the size defined by the data type of the variable.

When comparing TPF_regs.rx to a variable or to a constant that is less than 8 bytes, the compiler expands the variable or constant to an 8-byte value. If the variable is signed or a constant value, the compiler expands the value to a signed long. If the variable is unsigned (char, unsigned char, unsigned short, or unsigned int), the compiler expands it to an unsigned long.

To avoid problems when comparing values returned in the TPF_regs structure, apply the appropriate cast to the field in the TPF_regs structure. The following example shows a cast on the returned character data in r0 to compare against a variable of data type char: if ((char)regs.r0 == mychar) do_something();If the pointer to TPF_regs is NULL, the linkage does not save or restore any registers when going between C language and BAL programs.

The TPF_regs_Call64 function is inlined into the code during optimization and does not require an extra function call.

Chapter 6. Error conditions and error processing

This chapter describes error conditions that ALCS and application programs might detect. It explains how application programs can take action to avoid such errors from occurring and how they can deal with them when they do.

It also describes the ALCS wait mechanism and how application programs can use it to detect single and multiple errors on I/O operations.

Error conditions

Error conditions, described next, can be grouped as follows:

- Error conditions that ALCS detects immediately
- Error conditions that ALCS detects later
- Error conditions that application programs detect.

Error conditions that ALCS detects immediately

When ALCS detects an error condition, it takes a dump of the relevant parts of storage. Depending on the severity of the error, it terminates one or more entries. These errors include:

Application loop timeouts
ALCS terminates any entry which continues for too long without losing control. (See "The ALCS wait mechanism and error processing" on page 111.) It continues to process other entries normally.

Program check in an application program
ALCS terminates the entry.

Invalid assembler macro (or C function) parameters
ALCS terminates the entry.

Corruption of an ECB control area
ALCS terminates the entry.

I/O hardware errors
ALCS automatically retries any I/O operation that fails because of an I/O hardware error. If the error persists, ALCS takes a dump, but continues processing the entry.

When a hardware error occurs during the processing of a requested I/O service, ALCS indicates this error on return from the service (see "The ALCS wait mechanism and error processing" on page 111) and sets the ECB error indicators. Application programs must deal with this condition.

Deadlock
Deadlock can (potentially) suspend the processing of two or more entries indefinitely. It results, for example, from the following sequence of events:

1. Entry 1 holds file address A.
2. Entry 2 holds file address B.
3. Entry 1 tries to hold file address B (which is currently held by entry 1).
4. Entry 2 tries to hold file address A (which is currently held by entry 2).

Deadlock can be caused by application programs using any of the following (there are also more subtle forms of deadlock than these):

- Record hold facility (two entries each trying to hold the same two file addresses)
- Resource hold facility (two entries trying to hold the same two resources)
- The TASNC (`tasnc`) service (two entries trying to assign the same two sequential files or tapes).

To avoid deadlock, try to ensure that entries only hold one file address, resource, and so on, at a time. If this is impossible, ensure that all entries request holds of file addresses, resources, or assign the sequential files or tapes, in the same order.

If deadlock occurs, ALCS terminates one of the entries which caused the deadlock.

Other errors
ALCS detects corrupted data in some system-critical records and some control blocks. In addition, it also detects storage blocks not attached to ECBs. In such cases, ALCS generally takes a diagnostic dump.

Error conditions that ALCS detects later

Some application programming errors can result in error conditions that remain undetected for a long time. They include the following:

Lost record updates
Updating a record that is not held can cause updates to that record to be lost.

Lost pool-file addresses
Pool-file addresses that are not released when they are no longer required are lost to the system. Although pool management eventually recovers these "lost" pool addresses, you should request the RELFC (`relfc`) service to release a pool-file address.

Transactions which create multiple entries
Some transactions use ALCS create services to create multiple entries that process a single transaction. Incautious use of ALCS create services can "overload" ALCS. See “Transactions that create multiple entries” on page 102.

Error conditions that application programs detect

Application programs can detect error conditions. When this happens, the application program itself must carry out (or at least initiate) the error processing. See “Dealing with errors” on page 115.

Application programs can detect the following types of error:

Input data errors
These are errors in the format or parameters of an input message.

Wait completion errors
These are errors which cause a WAITC (`waitc` function), or a macro (function) that contains an implied wait to return an error condition. See “Dealing with errors” on page 115 for details of how to test for these errors.

Logic errors
These are errors caused by incorrect entry conditions for a particular program, or by fields in a retrieved record which have been incorrectly set up by application programs.

The ALCS wait mechanism and error processing

This section describes the ALCS wait mechanism and the associated error testing. Application programs use the ALCS wait mechanism when reading records from from, and writing records to, the real-time database and sequential files.

For most DASD or sequential file **write** operations, the application program does not need to know when the write has completed. ALCS I/O services that perform write operations do not allow the application program to check that the operation has completed successfully.

An exception to this is the FILNC (`filnc`) service. See “I/O services that require an implied wait” on page 112.

When an application requests a DASD or sequential file **read** service, it must check that the read has completed successfully before it tries to use the data. There are two types of read service; those that contain an implied wait, and those that do not.

I/O services that include an implied wait

Some read services contain an implied wait. When an application calls one of these services, ALCS suspends the entry until the read is complete, or until it has detected an error.

These services are called **implied-wait** services.

For example, the FINWC (finwc) service requests ALCS to find a record and waits for I/O to complete (see Figure 79 on page 112).

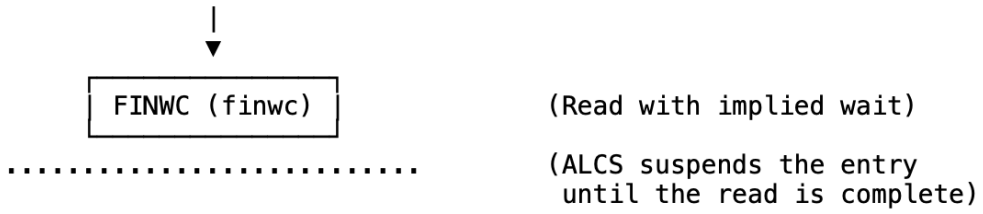


Figure 79. Read using an implied-wait service

In assembler programs, when an error occurs in the read, ALCS branches to the error label specified with the ERROR= parameter of the macro. This label must be the start of an error routine (see "Dealing with errors" on page 115).

Successful implied-wait C functions return a pointer to the record. When an error occurs, the C function returns a value of NULL. The application program must test the return value of the function and deal with errors as described in "Dealing with errors" on page 115.

You can use an implied-wait read service when the application does not need to do any processing while the read is taking place.

I/O services that require an implied wait

Other read services do not contain an implied wait. An application can request one of these services and continue processing while the read is taking place. However, it must follow it with an implied-wait service at some stage before it can access the record.

The FILNC (filnc) write service also behaves in this way. You must follow it with an implied-wait service.

In this book these read services and the write service FILNC (filnc) are referred to as **required-wait** services.

The simplest implied-wait service you can use is the WAITC (waitc) service. This causes ALCS to suspend the entry until either the read is complete or ALCS has detected an error (see Figure 80 on page 112).

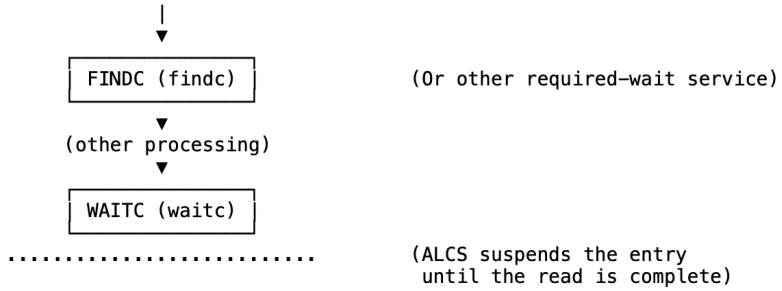


Figure 80. Wait after a required-wait read service

If no error occurs, ALCS restarts the entry at the next instruction after the WAITC (waitc) service. The storage level contains the record that has been read.

The WAITC contains a label to which ALCS branches when it detects an error. This label must be the start of an error routine which must deal with the errors as described in “Dealing with errors” on page 115.

The waitc function returns a nonzero value when it detects an error. The application program must test the return value of waitc and deal with errors as described in “Dealing with errors” on page 115.

Multiple required-wait services

An application can request several required-wait services before it requests an implied wait. It requests the implied wait by requesting a WAITC (waitc) service or by requesting another implied-wait service. Figure 81 on page 113 shows an example with a terminating FINWC (finwc) service that contains an implied wait.

Note: The three reads must all be on different data levels (for example, L4, L5, L7).

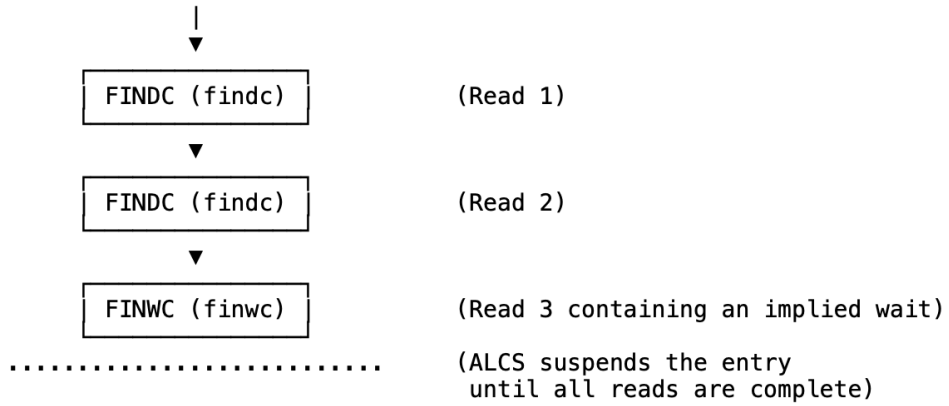


Figure 81. Multiple reads followed by an implied-wait read

To deal with this situation, ALCS uses an **I/O counter**. ALCS increments this counter by 1 every time the application requests a read service. Each time a read completes, ALCS reduces the I/O counter by 1. When the I/O counter reaches zero ALCS restarts the suspended entry.

In assembler programs, if an error has occurred in any of the reads, ALCS branches to the ERROR= label of the FINWC macro. This label must be the start of an error routine (see “Dealing with errors” on page 115).

In C language programs, if an error has occurred in any of the reads, the finwc function returns a NULL value. The application program must test the return value of finwc and deal with errors as described in “Dealing with errors” on page 115.

Loss of control

Use of any of the read services can cause the entry to lose control. However, if the application program calls a WAITC (waitc) service when the I/O counter is already zero, the entry does not lose control. So a wait **can**, but does not necessarily, cause the entry to lose control.

Summary of ALCS I/O services

All read services either require a subsequent WAITC (waitc) service or are themselves implied-wait services. The FILNC (filnc) service also requires a subsequent implied-wait service, other write services do not require an implied wait.

Each I/O service description in *ALCS Application Programming Reference - Assembler* (and function description in *ALCS Application Programming Reference - C Language*) explains if the service is a required wait, an implied wait, or neither. This information is summarized in Figure 82 on page 114.

```
No wait            Required wait            Implied wait
__________        ____________            ___________

FILEC  filec        FINDC    findc            WAITC   waitc
       filec_ext             findc_ext        FINWC   finwc
FILUC  filuc        FINHC    finhc                    finwc_ext
       filuc_ext             finhc_ext        FIWHC   fiwhc
                    FILNC    filnc                    fiwhc_ext
                             filnc_ext                find_record
                             file_record              find_record_ext
                             (with NOREL)             (with NOHOLD
(all others                  file_record_ext           or   HOLD
 not listed                  (with NOREL)              or   NOHOLD_WAIT
 here)                       find_record_ext           or   HOLD_WAIT)
                             (with NOHOLD_NOWAIT      tape_read
                              or   HOLD_NOWAIT)
                    TDTAC
                    TPRDC    tprdc
```

Figure 82. Summary of ALCS I/O services

Table 10 on page 114 summarizes how ALCS reports I/O errors with the various ALCS services.

Table 10. Reporting of I/O errors with ALCS service

Assembler service	C function	Wait service required?	Result of error (assembler)	Result of error (C language programs)
FILEC	filec filec_ext	No	Not available	Not available
	file_record file_record_ext without NOREL parameter	No		Not available
	file_record file_record_ext with NOREL parameter	Yes		Return value of waitc is nonzero (see note 2)
FILNC	filnc filnc_ext	Yes	WAITC error branch taken (see note 1)	Return value of waitc is nonzero (see note 2)
FILUC	filuc filuc_ext	No	Not available	Not available
FINDC	findc findc_ext	Yes	WAITC error branch taken (see note 1)	Return value of waitc is nonzero (see note 2)
	find_record	Implied		Return value of find_record is NULL
	find_record_ext with NOHOLD or HOLD or NOHOLD_WAIT or HOLD_WAIT parameter	Implied		Return value of find_record_ext is NULL

Table 10. Reporting of I/O errors with ALCS service (continued)				
Assembler service	**C function**	**Wait service required?**	**Result of error (assembler)**	**Result of error (C language programs)**
	`find_record_ext` with NOHOLD_NOWAIT or HOLD_NOWAIT parameter	Yes		Return value of `waitc` is nonzero (see note 2)
`FINHC`	`finhc` `finhc_ext`	Yes	`WAITC` error branch taken (see note 1)	Return value of `waitc` is nonzero (see note 2)
`FINWC`	`finwc` `finwc_ext`	Implied	`FINWC` error branch taken	Return value of `finwc` or `finwc_ext` is NULL
`FIWHC`	`fiwhc` `fiwhc_ext`	Implied	`FIWHC` error branch taken	Return value of `fiwhc` or `fiwhc_ext` is NULL
`TDTAC`		Yes	`WAITC` error branch taken (see note 1)	Not available
`TPRDC`	`tprdc`	Yes	`WAITC` error branch taken (see note 1)	Return value of `waitc` is nonzero (see note 2)
	`tape_read`	Implied		Return value of `tape_read` is NULL
All others	All others	No	Not available	Not available

Notes:

1. If this service is followed by a implied-wait macro or `WAITC`, ALCS branches to the label specified in the ERROR= parameter of the implied-wait service.
2. If this function is followed by a implied-wait function other than `waitc`, the return value of this implied-wait service is NULL.

Dealing with errors

ALCS indicates to an application when an error has occurred by branching to an error label (assembler programs) or setting a nonzero value of `waitc` function or a NULL value of an implied-wait function (C language programs).

The application has to determine what the error was and on what data level it occurred. The error indicators in the ECB and DECB allow applications to do this.

Error indicators in the ECB and DECB

The ECB contains 16 1-byte (`unsigned char`) error indicators, one for each data level. They are called EBCSD*n* (`ebcsd`*n*) where *n* is the data level (0 through hexadecimal F). This 16-byte area is also called CE1SUD. In addition, there is a 1-byte indicator CE1SUG. See Figure 83 on page 116.

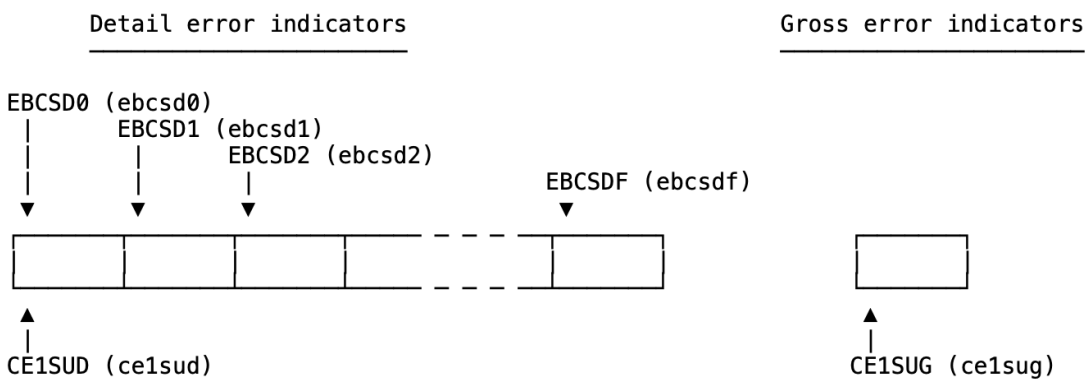


Figure 83. Error indicators in the ECB

A DECB contains a single 1-byte (`unsigned char`) error indicator, called IDECSUD (`idecsud`).

When an application requests an implied-wait service, ALCS sets one or more of these indicators, as follows:

EBCSD*n* (ebcsd*n*)

Each 1-byte (`char`) field contains bit indicators that ALCS sets when it discovers an error in processing an I/O request that specifies this particular ECB level (*n*).

The contents of EBCSD*n* (`ebcsdn`) are called the "detail error indicators".

ALCS indicates error conditions in the EBCSD*n* (`ebcsdn`) field in the ECB. The possible errors are defined by the symbols shown in Table 11 on page 116.

Table 11. Symbols for error conditions

Symbol	Condition detected by ALCS
CXSGHE	I/O hardware error
CXSGAE	Incorrect file address specified (DASD only)
CXSGIE	Record identifier mismatch (DASD only)
CXSGRE	Record code check mismatch (DASD only)
CXSGSE	Incorrect size (small) record read (sequential file only)
CXSGLE	Incorrect size (large) record read (sequential file only)
CXSGEE	End-of-file detected (sequential file only)
CXSGEE	Record not allocated (fixed file only)

CE1SUD (ce1sud)

This 16-byte array gives you an alternative way of accessing the error indicators.

In C language programs, `ebcsdn` is a `#defined` name for `ce1sud[n]`.

IDECSUD (idecsud)

This 1-byte (`unsigned char`) field contains bit indicators that ALCS sets when it discovers an error in processing an I/O request that specifies a DECB. This field is equivalent to the "detail error indicator", EBCSD*n* (`ebcsdn`), in the ECB. The possible errors are defined by the symbols shown in Table 11 on page 116.

CE1SUG (ce1sug)

ALCS sets this 1-byte (`unsigned char`) field to contain the result of a bitwise inclusive OR operation on all the 16 detail error indicators (EBCSD*n* (`ebcsd`*n*)) in the ECB and the detail error indicator (IDECSUD (`idecsud`)) in every DECB allocated to the ECB. If any of the detail error indicators have bits set this field has corresponding bits set.

The contents of CE1SUG (`ce1sug`) are called the "gross error indicators". In C language programs, the return value of the `waitc` function is the value of `ce1sug`. It is nonzero if an error has occurred.

Testing for errors

When a program has detected an error, it must test the ECB error indicators in EBCSD*n* (`ebcsd`*n*) or the DECB error indicator in IDECSUD (`idecsud`) to identify the cause.

Read two DASD records and test for errors in assembler programs

In the example shown in Figure 84 on page 117, an assembler application program uses `FINDC` to read two DASD records. They are fixed-file records with type #XMPRI. The application program reads record ordinal 5 on level 3 (D3) and record ordinal 6 on level 4 (D4).

Before issuing each `FINDC`, it uses `FACE` to calculate the file address of the record. After issuing both `FINDC`s, the application program issues `WAITC` to wait for I/O completion and test for I/O errors:

```
         LA    R00,5             LOAD ORDINAL NUMBER
         LA    R06,#XMPRI        LOAD RECORD TYPE
         LA    R07,CE1FA3        ADDRESS DATA LEVEL
         ENTRC FACE              AND CALCULATE FILE ADDRESS

         LTR   R00,R00           FILE ADDRESS OK
         BZ    ...               BRANCH IF NOT

         MVC   EBCID3(3),...     SET UP EXPECTED RECORD ID AND RCC
         FINDC D3                FIND THE RECORD ON D3

         L     R00,6             LOAD ORDINAL NUMBER
         LA    R06,#XMPRI        LOAD RECORD TYPE
         LA    R07,CE1FA4        ADDRESS DATA LEVEL
         ENTRC FACE              AND CALCULATE FILE ADDRESS

         LTR   R00,R00           FILE ADDRESS OK
         BZ    ...               BRANCH IF NOT

         MVC   EBCID4(3),...     SET UP EXPECTED RECORD ID AND RCC
         FINDC D4                FIND THE RECORD ON D4

         WAITC ERRRTN            WAIT FOR I/O ON D3 AND D4
  ⋮
ERRRTN   EQU   *
         TM    EBCSD3,CXSGIE     ID ERROR ON D3
         BO    ...               BRANCH IF YES

         TM    EBCSD4,CXSGIE     ID ERROR ON D4
         BO    ...               BRANCH IF YES
```

Figure 84. Read two DASD records and test for errors - assembler

Read two DASD records and test for errors in C language programs

In the example shown in Figure 85 on page 118, a C language program reads two DASD records and tests for errors. (The example assumes that the program has already set up the file addresses, record IDs, and optionally RCC, on data levels D0 and D1).

```
struct my_rec *rec_ptr;   /* define pointer for record structure */
findc(D0);
  :
if ( (rec_ptr = finwc(D1)) == NULL )
{
  /* error detected in implied wait */

  if (ecbptr()->ebcsd0)
  {
    /* error on level 0 */
  :
  }
  if (ecbptr()->ebcsd1)
  {
    /* error on level 1 */
  :
  }
}
else
{
  /* records read on level 0 and 1 are without error */
}
```

Figure 85. Read two DASD records and test for errors - C language

Multiple errors

An application might contain several implied-wait service requests interspersed with required-wait service requests. See Figure 86 on page 118.

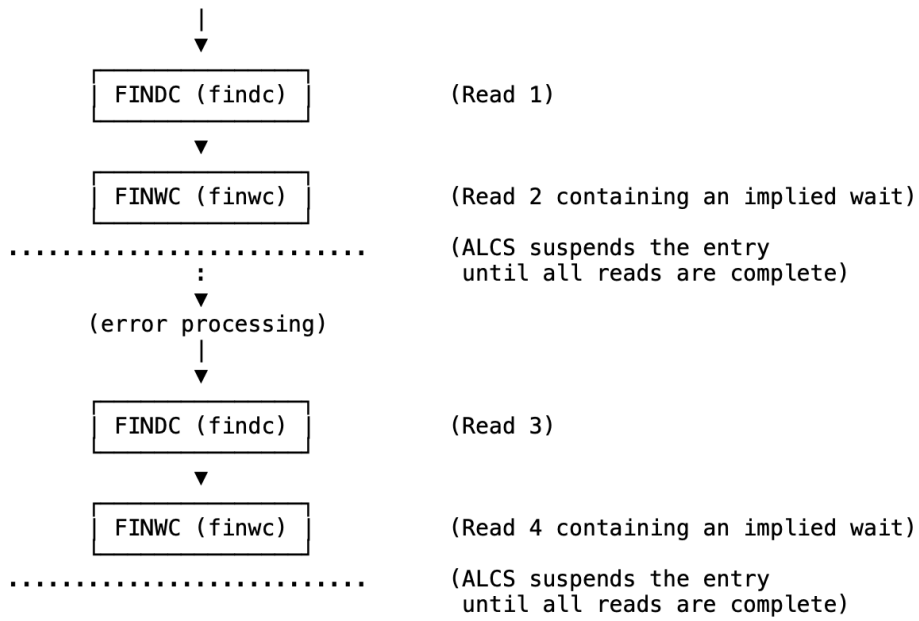


Figure 86. Multiple implied-wait services

If the first implied-wait service returns an error value, ALCS's setting of the error indicators depends on whether or not the second wait service detects a new error.

1. If the second implied-wait service does not detect a new error, it does not return an error value. ALCS **does not** clear the old error indicators. (In the example, Figure 86 on page 118, errors would refer to Read 1 or Read 2.)

2. If the new implied-wait service detects a new error, it does return an error value. ALCS clears any error indicators from the first implied-wait service. The error indicator settings all refer to the second wait service. (In the example, Figure 86 on page 118, errors would refer to Read 3 or Read 4.)

In the C language program example shown in Figure 87 on page 119, if the first `finwc` returns an error value, the error routine checks the type of error by testing `ebcsd1`. If the second `finwc` also returns an error, a second error routine checks the type of error by testing `ebcsd1`.

```
struct my_rec *rec_ptr;   /* define pointer for record structure */
if ( (rec_ptr = finwc(D1)) == NULL )
{
  /* first error routine  */
  if (ecbptr()->ebcsd0 & CXSGAE)
  {
    /* invalid file address */
  .
  }
  .
  if ( (rec_ptr = finwc(D1)) == NULL )
  {
    /* second error routine  */
    if (ecbptr()->ebcsd0 & CXSGAE)
    {
      /* invalid file address */
  .
    }
  }
  else
  {
    /* no error on second finwc()
    /* ecbptr()->ebcsd0 still contains error code for first finwc() */
  .
  }
}
```

Figure 87. Processing multiple I/O errors - C language

Sequential file errors

In processing sequential files, some of the conditions that ALCS detects might not be errors. Examples are: incorrect size (small) block, and end-of-file. The application must take appropriate action, as shown in the following examples.

You can test the return condition using the symbols shown in Table 11 on page 116. For example, to test for end-of-file, in a C language program, you could use the code shown in Figure 88 on page 119.

```
topnc("VPH",INPUT,NOBUFF);        /* open VPH tape for input  */
tprdc("VPH",D4,L2);               /* read a record from VPH   */
if (waitc())
{
/* error occurred */
  if (ecbptr()->ebcsd4 & CXSGEE)
  {
    puts("end-of-file on VPH tape");
    exit(0);
  }
}
/* process the record */
```

Figure 88. Test for sequential file errors - C language

Testing for several conditions

You can test for one of several conditions at the same time, for example:

Testing for several conditions in assembler programs

The assembler code example shown Figure 89 on page 120 reads a record from the general sequential file HOT into a size L2 storage block. The example tests for and accepts records that are smaller than the L2 block.

```
         TOPNC NAME=HOT,STATUS=I   OPEN THE FILE FOR INPUT
  ⋮
         TPRDC NAME=HOT,LEVEL=DD,BLOCK=L2  READ A BLOCK
         WAITC ERRRTN              AND WAIT FOR I/O

SHORT    EQU   *
         L     R02,CE1CRD          LOAD RECORD ADDRESS
         LH    R03,CE1CCD          LOAD RECORD LENGTH
  ⋮
EOF      EQU   *
         TCLSC HOT                 CLOSE THE FILE
  ⋮
ERRRTN   EQU   *
         TM    EBCSDD,CXSGSE       SHORT RECORD
         BO    SHORT               BRANCH IF YES - ACCEPT IT

         TM    EBCSDD,CXSGEE       END-OF-FILE
         BO    EOF                 BRANCH IF YES - CLOSE THE FILE
  ⋮
```

Figure 89. Test for several sequential file errors - assembler program

Testing for several conditions in C language programs

A similar C language code example is shown in Figure 90 on page 120.

```
/* define macro for ECB level 4 error indicator */
#define EBCSD4 (ecbptr()->ebcsd4)

topnc("VPH",INPUT,NOBUFF);        /* open VPH tape for input  */
tprdc("VPH",D4,L2);               /* read a record from VPH   */
if (waitc())
{
  switch (EBCSD4)
  {
    case 0     :
      /* no error */
      break;                     /* continue processing record */
    case CXSGHE:
      puts("hardware error on reading VPH tape");
      exit(0);
    case CXSGEE:
      puts("end-of-file on VPH tape");
      exit(0);
    case CXSGLE:
      puts("record too large");
      exit(0);
    case CXSGSE:
      puts("record too small");
      break;                     /* continue processing record */
    default:
      printf("unexpected error %#02X on reading VPH\n",EBCSD4);
      exit(0);
  }
}
/* process the record */
```

Figure 90. Test for several sequential file errors - C language

Recovery from wait-completion errors

The method of recovery depends on the type of error:

I/O hardware errors
Normally, an application program must terminate with a suitable message when an I/O hardware error occurs. However, when the program is writing a pool record for the first time, the error routine can itself attempt the recovery (see “Error recovery” on page 122).

End-of-file when reading a sequential file
This might not indicate an error condition. It might just indicate that the program has reached the end of the data. However, some sequential files might have a specific end-of-data record (for example, a record containing '9999...'). If the program receives an end-of-file indication with such a file before it has read the end-of-data record, then this is an error that the application must deal with.

Short records
When ALCS reads a short-length record from a sequential file, it puts the data into the storage block, as normal. It also stores the record length in the associated storage level (see “ECB storage levels” on page 79). The application can then process the short record that is available.

Errors from find and hold services
When ALCS returns an error value from a FINHC (`finhc`) or FIWHC (`fiwhc`) service, the file address might or might not be held. (This depends on what point in the processing the error occurred.) The application program must determine whether or not the file address is held, because if it is held the application program must unhold it.

The program can determine if the file address is held by testing whether the ECB or DECB storage level specified in the macro or C function call is occupied. It does this by calling the `levta` macro (`levtest`) service specifying the corresponding storage level or DECB address. If the storage level in the ECB of DECB is occupied, then the error occurred after the file address was held and the application program must unhold it.

Application program logic errors

Application programs can detect two types of logic errors:

- Incorrect entry conditions
- Database corruption.

Application programs should deal with them as follows:

Checking for logic errors

It is not practical for an application program to check every entry condition and every field in a referenced record. However, you might like to consider the following points:

- Where you can perform a check using a few extra statements, include them in the program.
- Before converting data from one type or format to another, check the data.
- Before performing calculations, check the values used in the calculation.
- Check values used as counters in loops.
- Check values which serve as indexes to tables against an appropriate maximum.
- Functions that many programs call should check all entry conditions. The entry conditions should be as simple as possible.

Recovery from logic errors

When an application program detects an application program logic error, it should do the following:

- Identify all the errors that have occurred.

- Call an error processing program, or request the ENTDC (`entrc`) service to transfer control to an error processing program. The called program constructs an appropriate error response message, or attempts an error recovery procedure, or both (see "Error recovery" on page 122).

  The error processing program sets a code in the ECB, for example, in EBER01 (`eber01`). This code indicates the nature of the error or errors.
- When the error-testing program returns to the calling program (using BACKC or `return`) the calling program tests the error switches and takes appropriate action (for example, sends a message to the originating terminal).

Error recovery

An application program can recover from an error situation by doing one or more of the following:

- Requesting operator intervention
- Retrying the process
- Using a duplicate copy of the data
- Falling back to an earlier version of the data
- Purging corrupted data that is of low importance
- Attempting to reconstruct the data.

Each of these has advantages and disadvantages as described below.

Request operator intervention
The safest technique for error recovery is generally to inform the operator of the error, using a dump or a message to RO CRAS, or both. The operator can then perform the necessary error recovery procedures. To minimize the risk of operator error, supply as much relevant information as possible, and subsequently check any corrective action taken by the operator.

Retry the process
When ALCS detects an I/O hardware error, it retries the I/O channel programs extensively before returning to the calling program and indicating an error in the implied-wait service. For this reason, application programs should not reissue an I/O macro (C function) call when ALCS indicates a hardware error.

However, there is one occasion when application programs can do this. If ALCS indicates a hardware error when the program is writing a record to a newly dispensed pool-file address, the application program can issue a GETFC (`getfc`) service to obtain a second pool file address. The application program can then write the record to this second address.

Note: The application program should not release the unused pool file address. If the unused pool-file address is released, ALCS will dispense it again, and the hardware error will recur.

Use duplicated information
If an application program identifies invalid data in a record field, it might be able to get the valid data from other fields in the record, or from other records. (This assumes some duplication of data in records.)

Note: If your installation uses duplicated databases, ALCS maintains exact duplicates of all DASD records in the duplicate databases. Application programs cannot directly access these duplicate records. The duplicates are maintained only for protection against hardware errors. ALCS handling of a duplicate database is transparent to application programs.

Fallback to a previous version
Critical data should be backed up regularly. If an application program finds that the current version of the data on DASD is in error, it might be possible for it to use the last saved version instead. However, this method of error recovery (known as **fallback**) is not generally used in application programs.

Purge corrupt data
If an application program discovers errors in unimportant data, it might be able to delete that data and all references to it. This process is known as **purging**. Purging should be used with caution, since it is

rarely possible to determine the consequences of deleting a data item. If you decide to purge data, ensure that details of purged data are preserved for subsequent inspection.

Attempt to reconstruct the data
An application program might be able to compute a "most probable" value for a field containing an incorrect value. However, it is difficult to predict the consequences if the data item should be reconstructed incorrectly. Use this method of error recovery with caution.

Designing error recovery routines

When designing error recovery routines for an application, consider the following points:

Response time
Input messages to ALCS applications typically require a fast response. Extensive error recovery can take a long time and delay the response. If an application program detects a need for error recovery while processing an input message, it can create a new entry to perform the error recovery. The original entry can then construct and transmit a suitable error message without delay.

Compounding the error
Some error recovery techniques used by application programs can introduce new errors, which might be more severe than the original error.

Concealing the cause of the error
Automatic activation of error recovery procedures can conceal the cause and even the occurrence of an error. Because of this, you must ensure that information about errors and their recovery is kept for inspection. You can do this using dumps, or messages to RO CRAS, or both.

Chapter 7. Application program management

This chapter describes:

- Application programming languages available in ALCS
- How to choose suitable application programming languages
- Callable services for high-level languages
- Using callable services with existing applications
- Getting storage in high-level languages.

Application programming languages

ALCS lets you write application programs in several languages. These languages are grouped here in the following categories:

- Assembler
- SabreTalk
- C language
- Other high-level languages.

The following sections describe characteristics of the languages in these categories, under the following headings:

Existing applications

There are large numbers of existing application programs that you might want to use on your ALCS system. If you are migrating to ALCS Version 2 from another TPF-family platform you can continue using your existing applications with few or no program changes.

Performance

The choice of programming language can affect the performance characteristics of the application. “Choosing languages for application programming” on page 130 discusses this topic in more detail.

Portability

The choice of programming language can affect the portability of the application. (This is not the only consideration; applications can contain both expected and unexpected dependencies on the platform for which they were developed.)

Other programming interfaces

ALCS supports these programming interfaces:

SQL
For accessing relational data bases.

CPI-C
For synchronous (conversational) communication with other applications.

ALCS also supports the related SNA programming interface, advanced program-to-program communication (APPC).

MQI
For asynchronous (queued) communication with other applications.

OLA
For communication with applications in WebSphere Application Server for z/OS.

TCP/IP
TCP/IP Sockets calls can be used to communicate with other applications on the same MVS machine or on a remote system.

Which of these services is available depends on the programming language that you use.

Monitor services and TPF Database Facility (TPFDF)

ALCS provides a wide range of specialized **monitor services** for assembler and SabreTalk language programs, and a more restricted range for C language programs. These services are described in *ALCS Application Programming Reference - Assembler* and *ALCS Application Programming Reference - C Language* respectively.

TPFDF provides an access method for application programs on TPF-family platforms. TPFDF provides services for assembler and C language programs. See the “Bibliography” on page 270 for more information about TPFDF.

Programs in other languages cannot access these specialized services directly. Instead, you must develop **callable services** to interface between these programs and the ALCS environment, including your existing application (if any).

By developing callable services, you can make your high-level language programs independent of the unique characteristics of the ALCS environment. This greatly simplifies porting the applications to or from other platforms. It also reduces the need for ALCS-specific programming skills in high-level language application programmers.

Callable services are described in more detail in “Providing callable services for high-level language programs” on page 132.

ALCS application program interfaces

Figure 91 on page 126 summarizes the various interfaces that application programs can make with other services.

Language restrictions

ALCS imposes some restrictions on the language facilities that you can use in ALCS application programs. These restrictions are different for the different languages, but always include the following:

- ALCS application programs must use ALCS services, TPFDF services, or other interfaces, when they perform any input or output operations on data.
- More generally, ALCS application programs must not directly invoke services provided by MVS or other MVS subsystems. For example, assembler language programs must not issue macros that generate SVC or PC instructions, or that generate branch entries to MVS.

These restrictions also apply to any programs that your application calls. This means (for example) that you must not call a utility program such as DFSORT which directly invokes MVS services.

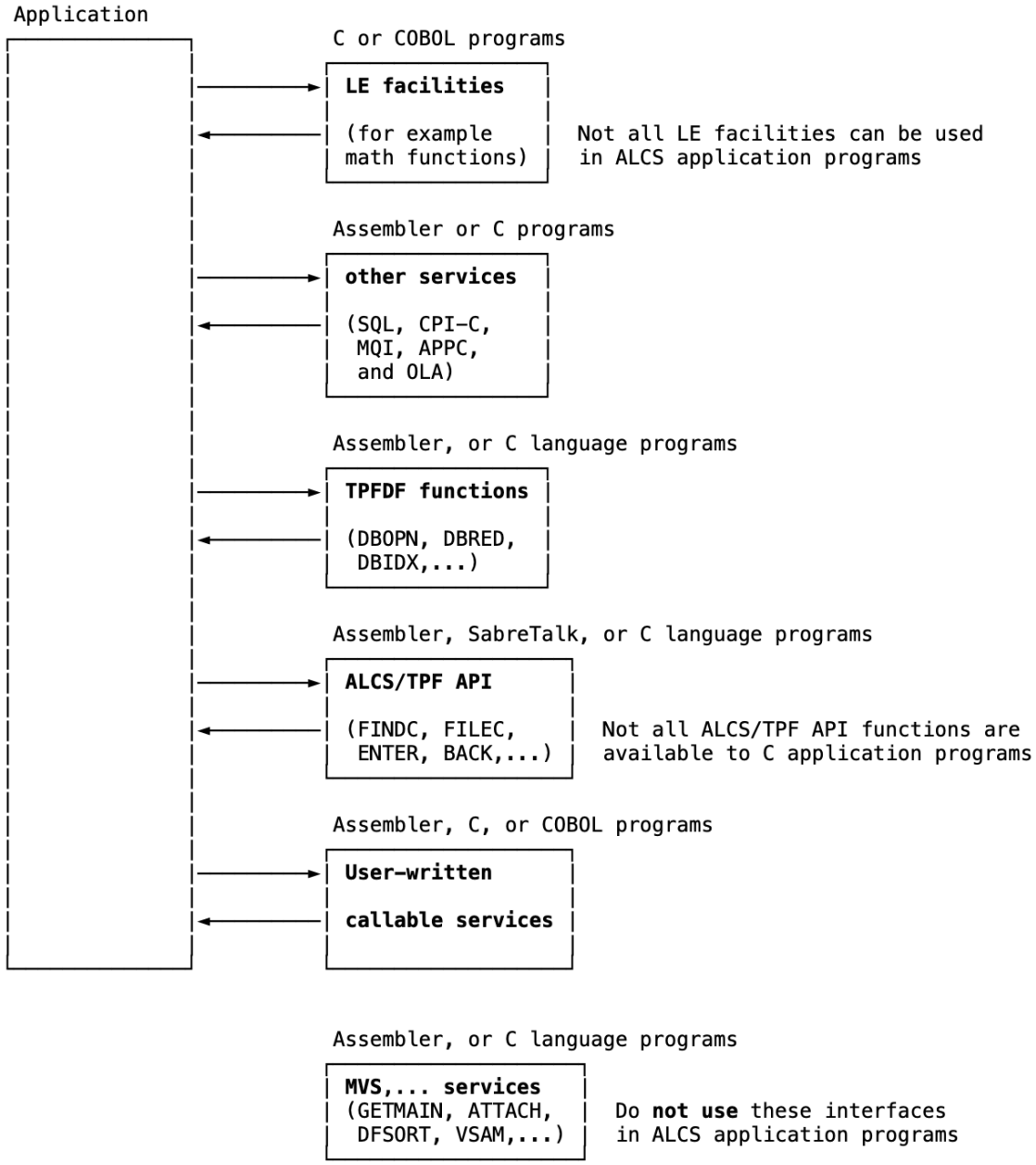


Figure 91. Summary of ALCS application program interfaces

Assembler

Existing applications

Most existing TPF-family applications are written in assembler language. You can run these programs under ALCS with few or no program changes.

In general, assembler language applications from other platforms do not work under ALCS. But you may be able to reuse assembler language routines in ALCS applications, provided that they comply with ALCS restrictions.

Performance

Provided that equally efficient algorithms are used, assembler language gives the best possible performance (shortest path length).

Portability

Assembler language programs are easily portable between TPF-family platforms, but difficult to port to or from other platforms.

Note: The High Level Assembler provides some facilities that are not available with Assembler H.

Other services

Assembler application programs can use SQL, CPI-C, APPC, MQI and OLA calls.

Monitor services and TPFDF

All ALCS monitor services and TPFDF services are available to assembler application programs.

Language restrictions

There are a number of restrictions that apply to ALCS assembler language application programming, for example:

- Programs must start with the ALCS `BEGIN` macroinstruction and end with the ALCS `FINIS` macroinstruction.
- Program and entry point (called **transfer vector** in ALCS) names must be 4 characters long (see Chapter 8, “Assembling, compiling, and link-editing application programs,” on page 145).
- Programs must not modify general registers 8 through 13 (RAP through RLD).
- Only the floating-point registers 0, 2, 4, 6 can be used in assembler programs.
- Programs cannot use privileged or semiprivileged instructions, or the SVC or PC instructions. They cannot use macros that generate these instructions.
- Programs must run in primary address space control (ASC) mode.
- Programs must not exceed 32KB.
- Programs cannot use the BAKR instruction.

For a full list of other restrictions, refer to *ALCS Application Programming Reference - Assembler*.

SabreTalk

SabreTalk is a programming language specially designed for writing application programs for the TPF-family platforms.

Existing applications

Some existing TPF-family applications are written in SabreTalk. You can run these programs under ALCS with few or no program changes.

There are no SabreTalk applications for other platforms.

Performance

Typically SabreTalk applications have path lengths that are longer than assembler applications, but shorter than other high-level languages.

Portability

SabreTalk programs are easily portable between TPF-family platforms, but not portable to other platforms.

Other services

These other services are not available to SabreTalk programs.

Monitor services and TPFDF

Most ALCS monitor services are available to SabreTalk programs. SabreTalk programs cannot access TPFDF facilities.

Language restrictions

SabreTalk is a special purpose TPF-family language. ALCS does not impose any restrictions on SabreTalk.

C language

Existing applications

Some existing TPF-family applications are written in C. You can run these programs under ALCS with few or no program changes.

There is a large number of existing C language applications for other platforms. But you may find it difficult to port these applications to ALCS if they contain dependencies on the platform for which they were developed.

You may be able to reuse C language routines in ALCS applications, provided that they comply with ALCS restrictions.

Performance

The performance of C language programs is similar to other high-level language programs.

Portability

With careful design, you can develop C language applications that are portable.

Applications that use the TPF-family API functions (`entrc`, `findc`, and so on) are portable only to and from other TPF-family platforms.

Applications that avoid using TPF-family API functions, and that comply with ALCS C language programming restrictions, can be portable to and from other (as well as TPF-family) platforms.

Other services

C application programs can use SQL, CPI-C, APPC, MQI and OLA calls.

Monitor services and TPFDF

Most ALCS monitor services and TPFDF services are available to C language programs. You must avoid using these services if want your programs to be portable to other (not TPF-family) platforms.

Language restrictions

There are a few restrictions that apply to ALCS C language application programming. The restrictions are similar to the restrictions imposed by other transaction processing platforms (such as CICS). The main restrictions are:

- You must not use C I/O functions (except for `stdin` and `stdout`).
- You must not have a `main`() function; ALCS supplies this.
- The FLOAT(NOAFP) compiler option must be used to limit the compiler to generating code using only floating-point registers 0, 2, 4, 6.

The C API is available, with restrictions mentioned in *ALCS Application Programming Reference - C Language*. A TPF-C compatible API is also available, again with some differences where the underlying TPF and ALCS functions differ.

Other high-level languages

Existing applications

There are a large number of existing applications for other platforms written in languages such as COBOL. But you may find it difficult to port these applications to ALCS if they contain dependencies on the platform for which they were developed.

Note: Programs that comply with ANSII COBOL or PL/I standards are highly portable (they are unlikely to contain significant platform dependencies).

You may be able to reuse routines in ALCS applications, provided that they comply with ALCS restrictions.

Performance

See “Choosing languages for application programming” on page 130.

Portability

Currently, you can only port applications written in assembler, SabreTalk, and C to other TPF-family platforms.

With careful design, you can develop programs in other high-level languages that are portable to and from other platforms. These programs must comply with ALCS language restrictions.

Other services

High-level language application programs can use SQL calls, provided that the DB2 precompiler supports the language. They can use CPI-C and APPC provided that APPC/MVS supports the langua And they can use MQI calls provided that WebSphere MQ for z/OS supports the language.

Monitor services and TPFDF

ALCS monitor services and TPFDF services are not available to high-level language (other than C) programs. You can develop your own callable services to make selected ALCS monitor services or TPFDF services available to your high-level language programs. See “Providing callable services for high-level language programs” on page 132.

Language restrictions

There are a few restrictions that apply to ALCS high-level language application programming. The restrictions are similar to the restrictions imposed by other transaction processing platforms (such as Customer Information Control System (CICS)). The main restriction is that you must not use I/O functions.

Do not use the following COBOL Language statements or reserved words:

```
ACCEPT
CLOSE
DELETE
DISPLAY (except to display diagnostic information during conversational trace)
FD
FILE
FILE-CONTROL
FILE-SECTION
INPUT-OUTPUT
I-O-CONTROL
MERGE
OPEN
READ
RERUN
REWRITE
SD
SORT
START
STOP literal
USE declaratives (you can use USE FOR DEBUGGING)
WRITE
```

Choosing languages for application programming

This section discusses some factors that can affect which programming language or languages you use to develop ALCS application programs.

Your decision may also depend on factors such as software license costs, national or enterprise standards, and so on.

Existing or purchased applications

Many ALCS installations use ALCS primarily to run an existing application. Typically, these applications consist of thousands of assembler language programs (though they can include some SabreTalk and C language programs).

For these applications, and for ALCS itself, you must have an assembler (Assembler H or High Level Assembler). And you will almost certainly need some staff with assembler language programming and debugging skills.

You may also need compilers and appropriate skills for the SabreTalk and C language programs. You may prefer to implement small enhancements, customization, and so on by updating existing programs without rewriting them in some other language.

Performance

Assembler language usually produces programs with the shortest possible path lengths. A commonly accepted guideline is that high-level language programs generate on the order of ten times as many processor instructions as assembler language programs with equivalent function.

On entry to, and return from, a high-level language program there are processes that construct and discard the high-level language environment. The performance cost of these processes can be significant for very short programs.

Note that:

- The path length of an application program usually does not make much difference to the end-user response time. The time required for I/O operations is much greater than the execution time of processor instructions. However, a longer average transaction path length requires a more powerful processor for a given transaction rate.
- The power of a high-level language can encourage programmers to use more efficient (but more complex) algorithms than they would use in assembler language. A more efficient algorithm can help offset the apparent overheads of the high-level language.

- Even on transaction processing systems, usually only a few programs need the highest possible performance. The path length of infrequently used transactions makes little difference to the average path length for all transactions.

Productivity

Using high-level languages instead of assembler language can significantly improve application programmer productivity. It is often much quicker and cheaper to write a program in a high-level language - especially if the program uses complex algorithms. Also, high-level languages can be easier to debug and fix.

You may be able to use an application generator (sometimes called a "fourth generation" language). If the application generator produces COBOL or C source code (or other language supported by ALCS) as output, you may be able to compile this to create ALCS application programs. But you must check that the source code complies with ALCS programming restrictions.

Skills availability and skills sharing

It is usually easier to find programmers with high-level language skills - especially COBOL and C - than with assembler skills.

If you intend to use ALCS monitor services in assembler or C language programs you need to understand these services.

If your installation employs application programmers who work on other platforms (such as CICS, MVS batch, and so on) it may be advantageous to standardize on a single language for the bulk of application development across all platforms. If you do this, you may be able to share application design, programming, and debugging skills, allowing the same staff to work on different platforms.

This is also true for some other programming interfaces. If your installation uses (or plans to use) SQL, CPI-C, MQI, or OLA then staff with the appropriate skills can use them across several different platforms, including ALCS.

Marketability of applications

If you plan to develop new applications, or enhancements to existing applications, you may want to sell the code you develop. (A number of users of the TPF-family platforms sell their applications.)

If you want to sell your application, or at least allow for the possibility, be aware that different TPF-family platforms support different languages (all support assembler language), and impose different programming restrictions.

ALCS Application Programming Reference - Assembler and *ALCS Application Programming Reference - C Language* contain detailed information about differences between the restrictions imposed by ALCS and TPF.

Any application that you sell may need to coexist with other applications already installed on your customers' systems. To allow this, you need to be careful how your application interacts with data and other programs. This is discussed in more detail in “Providing callable services for high-level language programs” on page 132.

Portability of applications

You may want ALCS applications that you develop to be portable to other platforms such as CICS, OS/2, OS/400, or AIX. To allow this, you should:

- Use a programming language that is supported by both ALCS and the other platform. Both C and COBOL are supported by a wide variety of platforms.
- Avoid using facilities that are available only under ALCS. In particular, do not directly access data stored on the ALCS data base. Instead, use a callable service. (See “Providing callable services for high-level language programs” on page 132.)

Function

Your choice of programming language can be affected by the functions you need in the application. In particular:

- You must write in assembler or C to use the ALCS monitor services.
- You cannot use SabreTalk if you need to use the other facilities (SQL, CPI-C, MQI and OLA).
- You must use a high-level language if you need to use the mathematical functions, array manipulation functions, and so on, that these languages provide.

Providing callable services for high-level language programs

If you are developing a new application, it will probably need to access (and possibly update) some information on the ALCS database.

As an example, consider an airline application. A typical airline application system maintains information such as:

- Flight schedules. Most airline seat reservation systems store some schedule information in the application global area. But different systems store the information in different formats and use different serialization conventions. Some systems do not save schedule information in the global area.
- Passenger data. Most airline seat reservation systems store passenger data in data base records called passenger name records (PNRs). But the format of these records varies between systems.
- End-user data. Most airline application systems store end-user data in data base records called agent assembly area records (AAAs). But the format of these records varies from system to system.

If you are developing a new application that will be part of such a system, it will probably need to access (and possibly update) some or all of this information.

If you write your new application in assembler, SabreTalk, or C then you could access the information directly. For example, your application could use ALCS global services to access flight schedule information, and use ALCS find/file services to read and write passenger name records (PNRs) or agent assembly area (AAA) records. But if you do this, your application program logic will contain dependencies on:

- TPF-family monitor services. Your application will not be easily portable to platforms that do not provide these services.
- The formats and access methods for the data. Your application will not be easily portable to systems, including TPF-family systems, that store the information in different places (for example, in data base records instead of in the global area) or that use different formats (record layouts, field lengths, and so on).
- The information that is stored. Different systems may store different data. For example, your system may store information about in-flight movies. If your application assumes that this information is available, it may not work on a system that does not store in-flight movie information.

You can avoid these dependencies by developing **callable services** that interface between your new application and the existing applications and data base. Callable services are (usually small) programs that you write in C or assembler language. Your application programs invoke these programs using the CALL interface. The callable service routines interface directly with the data base and existing application programs using ALCS monitor services. Figure 92 on page 133 shows this schematically.

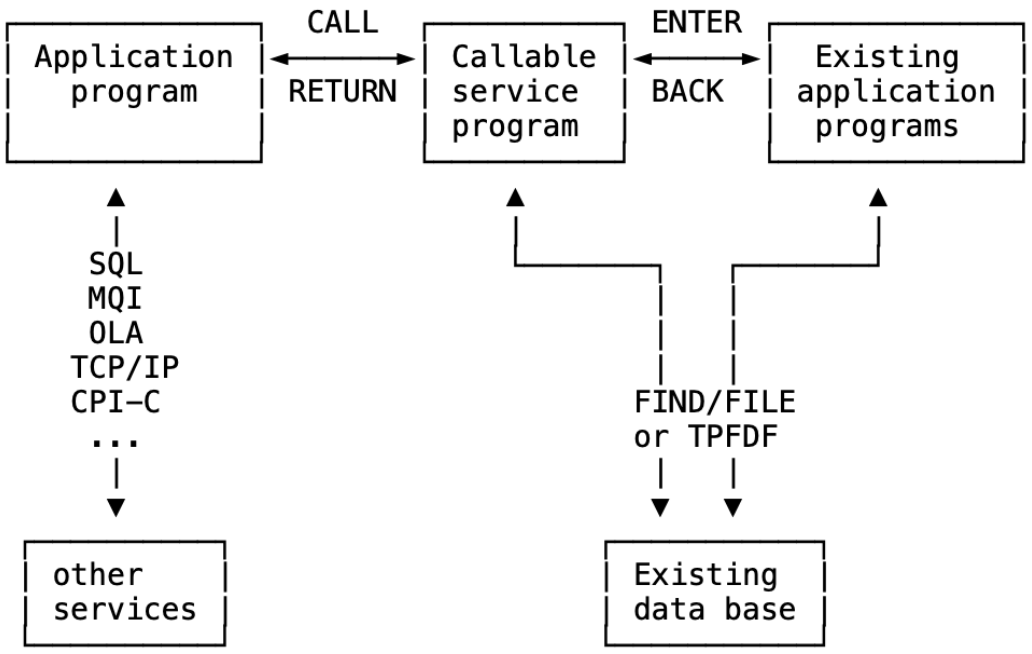


Figure 92. Callable services

The purpose of the callable service program in Figure 92 on page 133 is to isolate the application program from application and system functions that may be different on other systems.

Figure 92 on page 133 shows just one application program using the callable service, but you should design your callable service routines to be general-purpose so that many different application programs can use them.

Developing callable services

This section recommends some techniques to help you design and implement callable services.

Deciding what services to provide

Before you code your high-level language application, first decide what callable services you intend to implement. You should provide services that:

- Are easy to use in the calling application
- Can be used by several applications, not just the one you are developing now
- Improve the portability of the calling application.

To help you achieve these objectives, consider the following tests when you are designing a callable service:

- Consider if you could implement the service on a different system.

  For example, if your system stores end-user information in AAA records, you might implement a "retrieve AAA" callable service. But this would make it difficult to port your application to a system that does not save end-user information in AAA records (or a system that has a different format for AAA records).

  It is better to implement an "extract end-user information" callable service. On your system, the service routine can read an AAA record and return the required information to the caller. Provided that the information exists somewhere, you could implement the same service on a different system. The *service* would be the same, even though the service *routine* would be different.

- Consider if you could implement the service on a different platform.

  For example, you might implement a service to format an IBM 3270 display in a particular way and display data in specified fields. This might be difficult to implement, or inappropriate, or both on a platform such as OS/2 which provides a graphical user interface.

  It would be better to implement a service that accepts unformatted data and selects the appropriate formatting depending on the destination. That is, the service routine contains the device (or platform) dependent logic, not the calling routine.
- Consider if the end user can understand the service.

  For example, you might implement a service that sets the terminal hold (AAA hold) bit for a particular terminal. An end-user might find it easier to understand a service that prevents input from the terminal. Although this is a small difference (the implementation of the service would be identical in this case), *thinking* of the service in this way might make it easier for an application programmer to understand the function. It might also make it easier to understand how to implement the service on a different platform.

Deciding the name for a service

You must define an 8-character name for each callable service. Most languages allow you to define a longer, mnemonic name. You might, for example, define a longer name within a macro. See Chapter 8, "Assembling, compiling, and link-editing application programs," on page 145 for more details.

Serialization considerations for callable services

You can develop callable services to perform a wide variety of different functions. Services that do not access shared data (the database or the application global area) usually have no serialization requirements.

If a service does access shared data then either the calling programs or the callable service routine may need to serialize these accesses. It is usually better to include the serialization logic in callable service routines, because:

- You can directly invoke ALCS serialization services only from assembler, C, and SabreTalk programs. For other languages, you *must* include the serialization logic in the callable service routines.
- TPF-family serialization services (for example, record hold) are different from those that other platforms provide. It is easier to port your application (the calling programs) to other platforms if the serialization logic is in the callable service routines.
- Different installations can use different serialization for the same data - even if the applications run on the same platform. It is even possible that your installation may decide to change the serialization for some data. You can insulate your application (the calling programs) from these differences by including the serialization logic in the callable service routines.

The serialization that you need for a callable service depends on how the service accesses shared data. The following sections discuss serialization for:

- Services that read shared data. They return the information to the calling program. They are called **extract** services.
- Services that update shared data in a simple way. They store information that the calling program provides. They are called **simple update** services.
- Services that update shared data only if the shared data contains the expected before-update information. The calling program provides the data to store; it may also provide the expected before-update information. They are called **conditional update** services.
- Services that allow the calling program to extract and update information in separate calls. They are called **complex** services.

For more general information about serialization, see "Serialization" on page 41 and *ALCS Concepts and Facilities*.

Serialization for extract services

An extract service routine typically:

1. Reads one or more records
2. Extracts information from the records
3. Returns the information to the calling program.

If the information is entirely contained within a single record, then the service routine probably does not require any serialization.

But if the information is spread across several records, the service routine may need to use some form of serialization to ensure that it extracts consistent information. You need to review how other entries update the records.

Before you develop an extract service, consider whether calling programs must call the service routine more than once for the same entry. If possible, try to avoid this by having a single call return all the information that calling programs require. If you cannot do this, then you may need more complex serialization to ensure that the service returns consistent information, see "Serialization for complex services" on page 136.

Serialization for simple update services

A simple update service routine typically:

1. Reads one or more records
2. Updates information in the records
3. Returns to the calling program.

The service routine must serialize access to the records that it updates. Probably you will use record hold for this.

Before you develop a simple update service, consider whether calling programs can safely update the data unconditionally. You cannot (for example) use a simple update service if the calling program works like this:

1. Call a service to extract information from a record.
2. Use the returned information to compute the new (replacement) information.
3. Call a simple update service to store the new information.

The sequence is unsafe because another entry can update the information between steps "1" on page 135 and "3" on page 135. In that case, step "2" on page 135 incorrectly uses out-of-date information to compute the replacement information.

(You may be able to avoid this problem by including step "2" on page 135 in the service routine. For example a calling program can pass an increment to a service routine; the service routine updates a field by adding the increment.)

If your calling programs request updates that depend on the existing information in the database, consider implementing a conditional update service, see "Serialization for conditional update services" on page 135. If you cannot do this you may need more complex serialization, see "Serialization for complex services" on page 136.

Serialization for conditional update services

A conditional update service routine typically:

1. Reads one or more records
2. Checks the information already in the records

   If the information is the calling program expects, then:
3. Updates information in the records

4. Returns to the calling program.

The simplest case is where the condition is fixed. For example, you might develop a service routine to decrement the number of available seats on an aircraft if and only if the number is not already zero. In this case, the calling routine does not pass information about the condition - the condition is always "the number of available seats is not already zero".

In more complex cases, the calling program can provide test information to the service routine. For example, a service routine might perform the function:

> Reserve *n* hotel rooms if there are at least *m* rooms unoccupied.

In this case, the calling program provides both the number of rooms required (*n*) and the minimum number of available rooms (*m*).

Serialization for complex services

A complex update service typically requires the calling program to make a sequence of calls - to different service routines or to the same service routine but with different parameters. The entire sequence must be serialized. In the types of service already described, the service routine (or part of it) must be serialized. But for complex services, the *calling program* must be serialized.

To serialize the calling program, you can provide service routine calls that start and end serialization for a particular object. For example, you might have a complex service that provides a selection of calls to extract and update information for a particular airline flight. In this case, you could provide a call to start serialization for the flight and another to end serialization for the flight.

Return codes and reason codes

Callable service routines should set a **return code** to indicate whether or not the service completed correctly. The return code is a returned parameter (see “Parameters” on page 137).

Some callable service routines can detect many different error conditions. You can use a different return code value for each possible error, but it is likely to simplify the calling programs if the service routine sets the return code to indicate that an error occurred. It can use another returned parameter, called the **reason code** to provide more information about the error.

IBM has established a standard for return code values. You may want to follow this to provide a consistent interface to all callable services. The standard is:

Return code
Meaning

0
Normal completion - it worked. You should set this return code value for normal completion even if you do not expect the calling routine to check the return code. If you set return code 0, you should also set the reason code to 0.

4
Minor error (**Attention**) - it worked, but possibly not the way the caller expected.

If there are several possible reasons for return code 4, you can use a different reason code value for each. If there is only one possible reason, you should set the reason code to 0.

8
Error - it did not work.

If there are several possible reasons for return code 8, you can use a different reason code value for each. If there is only one possible reason, you should set the reason code to 0.

For example, a hotel booking system might implement a callable service to return the name of the guest staying in a specified room number. The service routine might set return codes as follows:

Return code
Meaning

0
The service returns the guest name.

4
There is no guest staying in the room.

8
The service did not work. The reason code indicates why, as follows:

Reason code
Meaning

0
There is no such room in the hotel.

1
The routine could not access the guest name list.

and so on.

Parameters

You should use parameters to transfer data between a calling application and a callable service. Typically, a callable service has two types of parameter, **passed parameters** and **returned parameters**. The calling routine puts data into passed parameters. The callable service routine puts data into the returned parameters.

The hotel booking system callable service, used as an example in "Return codes and reason codes" on page 136, might include the following parameters:

Room number
This is a passed parameter - the calling routine provides the room number.

Guest name
This is a returned parameter - the service routine returns the guest name (when the return code is 0).

Return code
This is a returned parameter - the service routine returns the return code.

Reason code
This is a returned parameter - the service routine returns the reason code (when the return code is 8).

To make your application as portable as possible, you need to design carefully the parameters for each callable service. In particular, you need to avoid parameters that make the calling routine depend on how you implement the service routine.

You might find it helpful to consider the following:

- You can develop a callable service that provides several functions. To do this you can include a passed parameter that specifies which function the caller requires.

  Even if your initial version of the callable service only provides a single function, you can include a passed parameter to allow you to add more functions later.
- It is sensible to allow for additional or different parameters. In the hotel booking system example, you might want to generalize the callable service to allow the caller to specify a hotel name, a date, and so on. Similarly, you might extend the service to return additional information about the guest, such as expected departure date, and so on.
- For parameters that contain character data, it is wise to use variable length character strings in preference to fixed length.

  For example, the airline industry used a 2-character code for each airline. After many years the standard changed to a 3-character code. This type of change is much easier to implement if program interfaces use variable length strings.

Using callable services with existing application programs

This section contains an example, showing how you can use an existing program from a high-level language by providing a callable service to access it. The callable service can be called from a high-level language program (or from an assembler program). See Figure 93 on page 138.

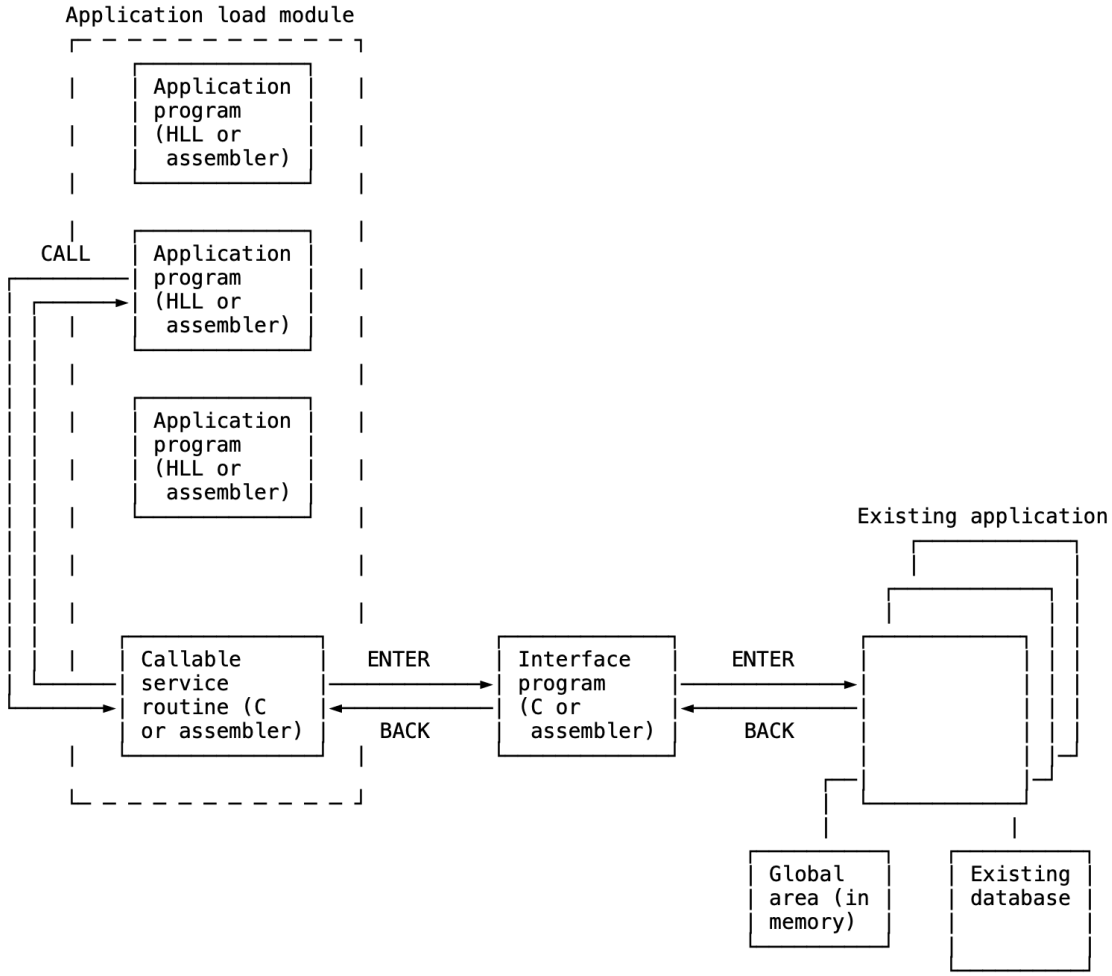


Figure 93. Using a callable service with an existing application program

The interface program shown in Figure 93 on page 138 sets up the entry conditions (for example register contents) for the existing program.

You need not have a separate program to set up these conditions, you could incorporate them in the callable service. However having a separate program has some advantages:

- The interface program need not be in the same load module as the callable service.
- The interface program could be changed when necessary, for example to access a different application program, without changing the callable service in any way.

Example of a callable service for an existing application

This example shows a callable service, GETPNT, which gets information from a passenger name record (PNR). The callable service requires a function written in assembler, and a new interface program, GPNT, which sets up the calling conditions for an existing IPARS program PRC1.

Callable service written in assembler (GETPNT)

The callable service shown in outline in Figure 94 on page 139 is written in assembler.

ALCS callable services written in assembler must use the ICELOG and ICPLOG macros, described in *ALCS Application Programming Reference - Assembler*.

```
         BEGIN NAME=GETPNT
         ICPLOG
  ⋮
setup call to get PNR, flight number, and passenger name
  ⋮
         ENTRC GPNT
  ⋮
setup return conditions
  ⋮
         ICELOG
  ⋮
         FINIS ,
         END   ,
```

Figure 94. Example callable service GETPNT - assembler

GPNT interface program

This assembler program provides an interface (sets up entry conditions) to the PRC1 program which gets PNR information. See Figure 95 on page 139.

```
         BEGIN NAME=GPNT
  ⋮
setup entry conditions for PRC1 to get PNR
  ⋮
         ENTRC PRC1 get the PNR information
  ⋮
move PNR information to record area passed from HLL
  ⋮
         BACKC ,
  ⋮
         FINIS ,
         END   ,
```

Figure 95. GPNT interface program - assembler

Calling GETPNT from a C language program

You use the GETPNT callable service by calling it in a C language program as shown in Figure 96 on page 140.

```
#pragma linkage(getpnt,OS)
  ⋮
struct pnr_information pnr_inf;
  ⋮
/* prototype for GETPNT function */
void getpnt(char*, char*, struct pnr_information *, int);
  ⋮
/*------------------------------------------------*/
/*   example GETPNT call in a C language program   */
/*------------------------------------------------*/

getpnt(name, flight, &pnr_inf, retcode);

if (retcode == 0)
{
 ...process PNR information
}
```

Figure 96. Using the callable service GETPNT from a C language program

Calling GETPNT from a COBOL language program

You use the GETPNT callable service by calling it in a COBOL program as shown in Figure 97 on page 140.

```
*--------------------------------------
* EXAMPLE GETPNT CALL IN A COBOL PROGRAM
*--------------------------------------
  ⋮
     CALL 'GETPNT' USING NAME, FLIGHT, PNR_INF, RET_CODE.
     IF RET_CODE EQ ZERO .... process PNR ....
  ⋮
```

Figure 97. Using the callable service from a COBOL program

Example callable service written in C language

Figure 98 on page 141 shows a callable service, QFLTTYPE, written in C language. This example shows how an existing IPARS program can be called from a high-level language. It returns the flight type for a supplied carrier code and flight number.

```
/*-------------------------------------------------------------*/
/*  QFLTTYPE( char *carrier, int flight_number, int * retcode); */
/*                                                             */
/*  this function calls the IPARS program NGT1 to get the      */
/*  flight type of the specified flight.                       */
/*  i.e. type 0, 1, or 2                                       */
/*                                                             */
/*  If the flight does not exist a value of -1 is returned.    */
/*  This value is returned in the location specified by        */
/*  parameter 3 as well as the in the return code to allow     */
/*  this function to be called from COBOL and PL/I             */
/*-------------------------------------------------------------*/

int QFLTTYPE (char  *carrier,
              int    flight_number,
              int   *retcode      )
{
struct TPF_regs regs = {0,0,0,0,0,0,0,0};

/* work area for passing parameters to NGT1 */
char work[16];

/* definition to extract the flight type returned by NGT1 */
#define FLTYPE ((work[8]&0x30)>>4)

  memcpy(work,carrier,2);
  sprintf(work+2,"%04d",flight_number);
  memcpy(work+6,"\00\00\06",3);
  regs.r2 = (int)work;
  entrc("NGT1",regs);
  if (ecbptr()->eber01 != 0) *retcode = -1;
  else *retcode = FLTYPE;
  return (*retcode);
}
```

Figure 98. Callable service (QFLTTYPE) written in C language

Calling QFLTTYPE from a C language program

Figure 99 on page 141 shows how a C language program could use the callable service QFLTTYPE.

```
/*-------------------------------------------------*/
/* call QFLTTYPE() to get the flight type of BA0009 */
/*-------------------------------------------------*/
  carrier = "BA";
  f_num = 9;
  QFLTTYPE(carrier, f_num, &f_type);

  printf("flight %2.2s%04d ", carrier, f_num);

  if (f_type < 0)
    printf("not valid\n");
  else
    printf("is type %d\n", f_type);
```

Figure 99. Using the callable service QFLTTYPE from a C language program

Calling QFLTTYPE from a COBOL program

Figure 100 on page 142 shows how a COBOL program could use the callable service QFLTTYPE.

```
     77   CARRIER    PIC X(2).
     77   F-NUM      PIC S9(9) COMP VALUE ZERO.
     77   F-TYPE     PIC S9(9) COMP VALUE ZERO.
  ⋮
*-------------------------------------------------------------*
*    CALL QFLTTYPE  TO GET THE FLIGHT TYPE OF BA0009          *
*-------------------------------------------------------------*
     MOVE 'BA' TO CARRIER.
     MOVE 9    TO F-NUM.
     CALL 'QFLTTYPE' USING CARRIER, F-NUM, F-TYPE.

     IF F-TYPE IS LESS THAN ZERO
       DISPLAY 'flight ' CARRIER F-NUM 'is not valid'
     ELSE
       DISPLAY 'flight ' CARRIER F-NUM 'is type ' F-TYPE.
```

Figure 100. Using the callable service QFLTTYPE from a COBOL program

Calling other functions and programs

The following information explains how to call other functions and programs in your applications.

Calling a C program (from assembler)

There can be times when you want to call a segment written in C from a segment written in assembler. Use the CALLC macro if the C shared object (CSO) being called passes paramenters. See the *ALCS Application Programming Reference - Assembler* for information on the CALLC macro.

Note: The preceding information does not apply to C++ functions except for those that use extern "C" linkage.

When you call a CSO from a BAL program using the `entxc` macro, the function being used as the entry point cannot take any parameters.

Calling an assembler program (from C)

A special protocol is required when C language programs call basic assembler language (BAL) programs because the conventions used to pass parameters are different. This protocol is used to pass parameters to the BAL program and receive the return values. BAL programs run in 31-bit addressing mode and use registers R0-R7 to receive parameters and return values.

TPF_regs structure

The prototype statement that declares a BAL program call from a C language program must contain one parameter, a pointer to the TPF_regs structure. The return data type must be void.

For example, the following shows the prototype statement when BAL program ABCD is called and instructs the complier to check the data of the parameter and not to expect a return value:

```
void ABCD(struct TPF_regs *) ;
```

The TPF_regs structure must be included in the prototype statement even if the BAL program that is called does not receive any parameters (in registers) or return any values (in registers). For this, the calling C language program must pass NULL instead of a pointer to a TPF_regs structure or errors can result.

Note: Because BAL prorams do not recognize how values are returned to a C function, the results of declaring a return data type other than void cannot be predicted.

The following example shows how the TPF_regs structure is defined in tpf/tpfregs.h. Registers r0-r7 are 4 bytes each and correspond to registers R0-R7.

Calling an assembler program (from C)

```
struct TPF_regs {
    long r0;
    long r1;
    long r2;
    long r3;
    long r4;
    long r5;
    long r6;
    long r7;
  } ;
```

TPF_regs in 31-bit addressing mode

When a C language program calls a BAL program, the linkage takes the data from fields r0-r7 in the TPF_regs structure and loads it into R0-R7.

When a BAL program returns to the C language program and TPF_regs is in 31-bit addressing mode, the linkage processes R0-R7 as signed 4-byte values and propagates the sign bit (31) through the upper 4 bytes of each register. After the sign bits are propagated, the contents of R0-R7 are saved in the TPF_regs structure for reference by the C language program.

For following example shows the function call coded to call BAL program ABCD with TPF_regs

```
ABCD(regs);
```

The following example shows how TPF_regs is used:

```
void ABCD(struct TPF_regs *) ;

int myfunction(short *s, unsigned int *ui, char *c)
{
   struct TPF_regs regs;  /* Define TPF_regs variable */

   regs.r0 = 0 ;          /* assign integer constance 0 to R0 */
   regs.r1 = *s;          /* assign short to R1 */
   regs.r2 = *ui ;        /* assign short to R2 */
   regs.r3 = *c ;         /* assign char to R3 */

   ABCD (regs) ;         /* Call assembler program using TPF_regs */
                          /* in 31-bit mode */

   /* returns signed int, but linkage propagates sign bit, */
   /* so no cast required                                  */
   if (regs.r0 == -1) return 0 ;

   /* cast to signed short */
   if ((short)regs.r1 == *s) return 1;

   /* cast to unsigned int */
   if ((unsigned int)regs.r2 == *ui) return 2 ;

   /* '*c' is a character so case regs.r3  */
   if ((char)regs.r3 == *c) return 3 ;

   return -1 ;
  }
```

TPF_regs programming considerations

- Use casting to avoid compiler warnings when comparing, saving, or retrieving pointers from the TPF_regs structure.

- When storing a variable or constant into TPF_regs.rx that is less than 4 bytes the compiler automatically takes the value and expands it into a 4 byte long data type before storing it in the TPF_regs structure. If the variable is a signed or a constant value, the compiler expands the values to a signed long. If the variable is unsigned (char, unsigned char, unsigned short, or unsigned int) the compiler expands it to an unsigned long.
- When storing the contents of TPF_regs.rx into another variable, the compiler automatically truncates the value in TPF_regs.rx to fit in the size defined by the data type of the variable.
- When comparing TPF_regs.rx to a variable or to a constant that is less than 4 bytes, the compiler expands the variable or constant to a 4 type value. If the variable is signed or a constant value, the compiler expands the value of a signed long. If the variable is unsigned (char, unsigned char, unsigned short, or unsigned int), the compiler expands it to an unsigned long.
- To avoid problems when comparing values returned in the TPF_regs structure, apply the appropriate cast to the field in the TPF_regs structure. The following example shows a cast on the returned character data in r0 to compare against a variable of data type char:

  ```
  if ((char)regs.r0 == mychar) do_something();
  ```

- If the pointer to TPF_regs is NULL, the linkage does not save or restore any registers when going between C language and BAL programs.

Chapter 8. Assembling, compiling, and link-editing application programs

This chapter describes:

- Application source modules and load modules
- Static and dynamic program linkage
- How to name programs and transfer vectors
- How to assemble or compile application programs, build program headers, and prelink high-level language programs
- How to link-edit application programs
- Loading application programs for testing and production use.

Application source modules

You write your ALCS application programs in whatever language you choose (see “Application programming languages” on page 124).

In addition to the source modules for your application programs, you will also need source modules for the assembler language macrodefinitions, C header files, COBOL copy books, PL/I include files, and so on.

ALCS supplies definitions for macros that you can use in assembler language programs, and header files that you can use in C language programs. You can also develop your own assembler language macros and C header files. For a description of the supplied assembler macros and guidelines on how to write your own macrodefinitions, see *ALCS Application Programming Reference - Assembler*. For a description of the supplied C header files and guidelines on how to write your own, see *ALCS Application Programming Reference - C Language*.

You can use whatever editors, library management systems, and so on that you prefer, but you will probably want to keep your source modules in an MVS partitioned data set (PDS) or partitioned data set - extended (PDSE). For example, you might choose to use the ISPF EDIT program to write and update your source modules.

Program size

ALCS imposes some restrictions on the maximum size of an application program. In particular, ALCS assembler language programs must not exceed 32KB of object code. IBM recommends that you do not write application programs to the maximum size permitted. Some reasons for this are:

- Some TPF-family products impose stricter limits on program size. In particular, TPF restricts application programs to a maximum of 4KB.
- Assembler language programs that exceed 4KB require more than one base register.
- Small programs are usually easier to write, understand, and maintain than large programs.
- It can be difficult to apply small changes or fixes to a program that is already at or near the maximum size.

IBM recommends that you restrict ALCS assembler language application programs to approximately 3500 bytes of object code. Depending on the total size of the program constants, this corresponds approximately to 900 executable instructions.

There is no maximum size limitation for high-level language programs. However, IBM recommends that you avoid writing large monolithic programs. Ideally you should aim to structure your code into functions or subroutines which are small enough to fit onto a single page of the program listing. If a function is common to several programs or is in itself a logically separate routine IBM recommends you to compile

© Copyright IBM Corp. 2003, 2019

it as a separate module. Programs and functions or subroutines that have been compiled separately are combined when building the load modules.

TPF Compatibility:

Another reason for dividing programs into small functions is that if you are writing programs that must be compatible with TPF support, the maximum size of each compiled program or function is 4K bytes. This restriction does not apply to ISO-C support in TPF.

Converting assembler DSECTs to C language structures

You may have (or plan to write) C language applications that refer to data for which you already have assembler language DSECTs. You can, of course, write C language structures for this data. An alternative is to use a utility program that automatically converts assembler DSECTs into corresponding C structures.

If you have a large number of DSECTs to convert, a utility can substantially reduce the programming effort, and also reduce the chance of accidentally creating a C structure that does not map the existing data exactly.

You can use the DSECT conversion utility provided with the IBM z/OS XL C/C++ Compiler. ALCS includes an ISPF panel to run the DSECT utility. If you prefer to build your own JCL for the process, you can use the panel to create a sample job.

The application global area DSECTs are a special case. ALCS includes a special utility (DXCBGTAG) that process assembler language global area DSECTs into a form that allows C language programs to access the global area. DXCBGTAG is described in *ALCS Installation and Customization*

Library control, naming conventions, and versions

A typical ALCS installation has a large number of application source modules (programs, macros, header files, and so on). For example, airline passenger services applications comprise tens of thousands of application program source modules.

If your installation does more than a trivial amount of application development, enhancement, customization, or maintenance, you will probably have different versions of some source modules. For a given module, you might have:

- A version that is in use on your production system
- A version containing a fix that is being tested
- A version containing enhancements that is still being developed.

Libraries

You will probably find it easier to keep track of your source modules if you use separate libraries. As a minimum:

- Keep different types of source module in different libraries. For example, use separate libraries for:

  Assembler programs
  Assembler macrodefinitions
  C language programs
  C header files.
- Keep your own source modules in separate libraries from IBM-supplied source modules. Also, if you modify programs that you have bought, keep your own modified versions of the programs in separate libraries from the original unmodified versions.
- Keep production versions of source modules in separate libraries from development and test versions.

Naming conventions

You will probably need a convention that helps you to avoid name clashes - that is, accidentally having two different source modules with the same name. In particular, you should be careful to avoid using

names that might clash with source modules that you might buy from application vendors or source modules that IBM provides. You should always avoid names starting 'DXC'.

You might also find it helpful to use a naming convention that associates source modules with their function.

Naming conventions - programs and transfer vectors

You must give each new program and transfer vector a unique 4-character name. Each name must start with an alphabetic character, but names beginning with A, B, and C are reserved for IBM programs as follows:

A
Installation-wide ECB-controlled exit programs

B
Data programs (such as Recoup descriptor programs)

C
ALCS ECB-controlled monitor programs.

The following names are also reserved:

```
FACE
FACS
GOA<i>n</i>
RLCH
TIA1
UGU1
XHP1
```

TPFDF compatibility:

To avoid conflict with TPFDF, do not use program names, or transfer vector names, beginning with UF.

Naming conventions - assembler macros

For a full list of restrictions, refer to *ALCS Application Programming Reference - Assembler*. These include the following:

- Do not use names starting with the characters DXC.
- Do not use names of the form *ccncc*, where *c* is an alphabetic character and *n* is numeric (for example, EB0EB and CM1CM).
- Do not use names of the form *ccc*EQ, where *c* is an alphabetic character (for example, CPSEQ and SYSEQ).
- Do not use names starting with the characters ZU or DB.

Naming conventions - high-level language components

For a full list of restrictions, refer to *ALCS Application Programming Reference - C Language*. These include the following:

- Do not use names starting with `dxc` or `tpf` for header files or programs. These file names are used by ALCS.
- Do not define preprocessor or C language variables starting with the characters: `DXC, dxc,` TPF, or `tpf`. Your names might conflict with names used by ALCS.
- ALCS uses file names starting with the characters `c$` for header files that have been created from assembler DSECT macrodefinitions. Do not use such names for other files.

ALCS cross-reference facility - DXCXREF

You can use the ALCS cross-reference facility program, DXCXREF, to check if a particular name (or any symbol) is already used in your existing source module libraries. You can run DXCXREF from the ISPF panels (Operations).

Version numbers

Although ALCS does not require it, many installations associate a 2-character **version number** with each application program source module (note that although it is called a version "number", it is not necessarily numeric). Different versions of the program have different version numbers. Less commonly, some installations also use version numbers with other types of source module, such as assembler macrodefinitions.

Typically, installations that use version numbers include the version number as the last two characters of the source and object module name. For example, for version 21 of program FAAA the source and object module name is 'FAAA21'.

If you want the version number available at execution time, you can include it in the VERSION parameter of the BEGIN macroinstruction of assembler programs, or in the VERSION parameter of the BEGIN statement in the entry points definition file for high-level language programs. If you do this, this version number is displayed (for example) in the response to the ALCS ZDPRG command.

Application load modules

Before you can use an application program under ALCS, you must assemble or compile the program source module to produce an object module. For each source module, there is one object module. Although it is not a requirement, IBM recommends that you use the same name for the source and object modules.

You must then link-edit the object module to produce a load module, called an **application load module** in ALCS. You can link-edit a single object module to produce an application load module. But it is likely that you will want to include several object modules in each application load module.

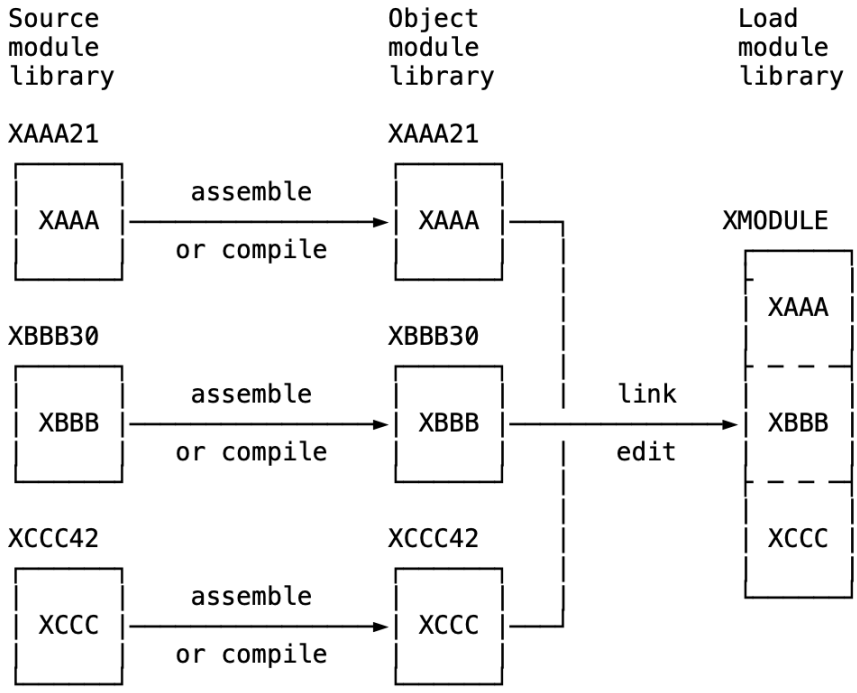


Figure 101. Relationship between source, object, and load modules

Figure 101 on page 148 illustrates this process. In the figure there are three application programs, XAAA, XBBB, and XCCC. The source and object module names include the version number, XAAA is version 21, XBBB is version 30, and XCCC is version 42. All three programs are included in the application load module called XMODULE.

Load module names

You must decide on a name for each application program load module. The name you choose does not have to be the name of any of the object modules that the load module contains. You use this name to tell ALCS which load module you want loaded - ALCS looks at the **contents** of the load module to discover which programs (object modules) it contains.

To avoid name clashes with components of ALCS itself, do not use load module names that start with 'DXC'.

How ALCS uses application load modules

A typical ALCS installation has thousands of application programs. ALCS must load these programs into memory before you can use them. You do not provide ALCS with a list of programs to load; instead, you provide a list of load modules.

There are three ways to specify the load modules that you want ALCS to load:

- You (or an ALCS system programmer) must provide a list of application load modules that ALCS loads at startup time. Usually, these load modules contain a complete "base" set of application programs.
- You (or an ALCS operator) can use the ALCS ZPCTL LOAD command to load additional application load modules while ALCS is executing. This allows you to add or modify application programs and have your end users start to use them without bringing down ALCS.
- You can use the ALCS ZPCTL LOAD command with the TEST option to load additional application load modules while ALCS is executing. If you use the TEST option, then only the messages that you enter use the new or modified programs. This allows you to test your new or modified application programs with minimum disruption to other ALCS end users (their messages continue to use the existing versions of programs).

ALCS Installation and Customization describes the list of application load modules that ALCS loads at startup time. *ALCS Operation and Maintenance* describes the ZPCTL command.

ALCS maintains a list of all the application programs in memory. When it loads an application load module, ALCS checks the contents of the load module to discover what programs it contains, and updates the list as follows:

- When it finds a program that is not already in the list, it **adds** the new program.
- When it finds a program that is already in the list, it **replaces** the existing program with the new version.

ALCS updates its list of application programs in such a way that:

- Any entry already executing at the time ALCS loads the application load module continues to use the existing versions of the programs it contains.
- All the programs in a load module appear to be loaded simultaneously. That is, any entry either uses all the new and changed programs in the load module or it uses none of them.

This allows you to change the interfaces between programs and start to use the new programs (with the new interfaces) without shutting down ALCS or disrupting entries that are already executing. Suppose, for example that you change two programs XAAA and XBBB so that the new version of XAAA works correctly with the new version of XBBB, but does not work with the old version of XBBB. You should include the new versions of both XAAA and XBBB in the same application load module - this ensures that any entry which uses the new version of XAAA also uses the new version of XBBB.

As well as allowing you to load application load modules with ZPCTL LOAD, ALCS also allows you to unload them with ZPCTL UNLOAD. You might want to do this if your end users report errors in the programs in the load module.

As with ZPCTL LOAD, ALCS processes ZPCTL UNLOAD in such a way that:

- Any entry already executing at the time ALCS unloads the application load module continues to use the versions of the programs it contains. (ALCS keeps the load module in memory until there are no entries left that still require the programs it contains.)
- All the programs in a load module appear to be unloaded simultaneously. That is, any entry either uses all programs in the load module, or it uses none of them.

Choosing what programs to include in a load module

You will probably have two (at least) types of application load module:

- Those that ALCS loads during startup. This set of application load modules contains the complete base application that you use on your production system.

  These are called the **base application load modules**
- Those that you load dynamically with ZPCTL LOAD (and possibly unload with ZPCTL UNLOAD). These application load modules contain fixes and enhancements to the base application, and possibly complete new functions.

  These are called **update application load modules**.

When you are deciding which application programs to include in which load modules, you need to consider that, for both types of load module:

- Programs that link to each other using the ALCS dynamic program linkage (ENTER/BACK) mechanism can be in the same application load module, or they can be in different load modules. (The ALCS ENTER/BACK mechanism is only available in assembler, SabreTalk, and C language application programs.)
- Programs that link to each other using the static program linkage (CALL/RETURN) mechanism must be in the same application load module. In addition:
  - If an application load module contains any programs that use these programming interfaces (SQL, CPI-C, APPC, MQI, OLA, and TCP/IP) it must also include the special ALCS program CDSN.
  - If an application load module contains any programs that use a callable service it must also contain the callable service routine.
  - High-level language routines that call each other as external procedures must be in the same application load module.

  Note that if (for example) two load modules contain programs that call the same callable service routine you must include a copy of the callable service routine in both load modules.

  “Static and dynamic program linkage” on page 156 provides more detail.

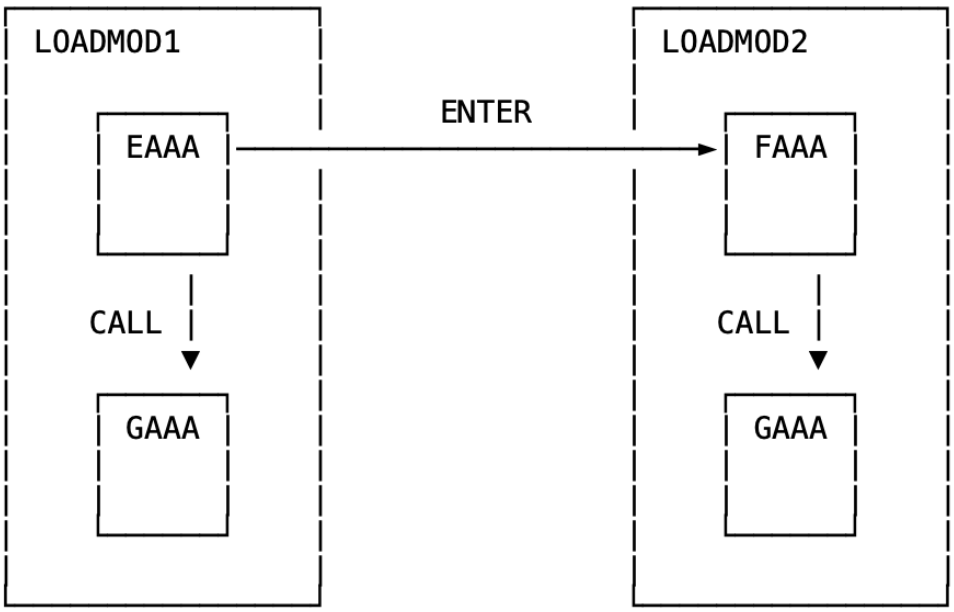


Figure 102. Load modules - static and dynamic program linkage

Figure 102 on page 151 shows two application load modules, LOADMOD1 and LOADMOD2. LOADMOD1 contains program EAAA, which ENTERs program FAAA and CALLs program GAAA. LOADMOD2 contains program FAAA which also CALLs GAAA.

Programs EAAA and FAAA can be in different load modules because they link using the ALCS ENTER/BACK mechanism. LOADMOD1 must contain a copy of program GAAA because EAAA CALLs it. LOADMOD2 must also contain a copy of program GAAA because FAAA CALLs it. You could, of course, include EAAA and FAAA in the same load module, in which case they could both use the same copy of GAAA.

Guidelines for base application load modules

You should aim to keep the total number of base application load modules down to about 10 or 20. Typically this means that each load module will contain hundreds of application programs.

It is not particularly important which programs are in which base application load module because ALCS loads all these load modules before it starts to process input messages. But remember:

- If any load module contains programs that CALL other programs, you must include copies of the CALL programs in the load module. If you do not do this, the link-editor program will report unresolved external references.
- Other than CALLed programs, it is best to avoid having the same program in more than one base application load module. ALCS only uses one copy (the one it loads last), and other copies simply waste memory.
- You will probably find it helpful to have a simple rule to decide which programs you include in which load modules. For example, you might choose to group application programs by name, linking all programs that start with D into one load module, all that start with E into another, and so on. (Remember that you should not use names starting with A, B, or C for your application programs - these names are reserved for ALCS.)

Guidelines for update application load modules

When you link-edit an update application load module, you should include all the programs affected by the update. This allows you to apply the complete update with a single ZPCTL LOAD command. And if it turns out that your update does not work, you can remove it with a single ZPCTL UNLOAD command.

For example, if you develop a fix that requires you to change the programs BAAA, CAAA, and DAAA, and to add a new program EAAA, then you should link-edit an update application load module that contains BAAA, CAAA, DAAA, and EAAA. It does not matter if programs BAAA, CAAA, and DAAA are in the same base application load module or in different ones.

If you later develop an unrelated fix that affects different application programs, you should link-edit another application load module to contain this second fix. This allows you to apply or remove the two unrelated fixes independently.

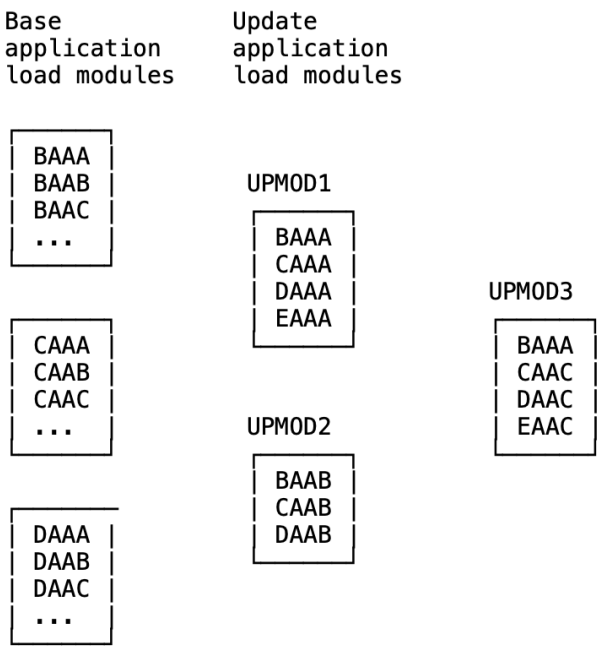


Figure 103. Relationship between base and update application load modules

Figure 103 on page 152 shows a possible scheme where programs are included in the base application load modules according to the initial letter of the program name. The figure shows update load modules, UPMOD1, UPMOD2, and UPMOD3, that contain fixes. Each of these update load modules contains all the new and changed programs associated with the fix.

UPMOD1 and UPMOD2 are independent of each other, but both UPMOD1 and UPMOD3 contain program BAAA. This means you must load UPMOD1 before you load UPMOD3. If you load UPMOD3 first, then you load a new version of BAAA but continue to use the old versions of CAAA, DAAA, and EAAA (if any). BAAA may not work correctly with these old versions. Similarly, if you need to unload UPMOD1, you must first unload UPMOD3

Note: A dynamic linkage from BAAB to BAAA will branch from the BAAB in the base load module to the BAAA in the update load module. But a static linkage from BAAB to BAAA will ignore the new BAAA in the update load module.

Merging updates into base application load modules

When you restart ALCS after a planned or unplanned shutdown, ALCS automatically reloads all the base application load modules. But the ALCS operator must use the ZPCTL LOAD command to reload any update application load modules.

To avoid this manual intervention, you must either:

- Add the names of the update application load modules to the base application load module list,

  or:

- Relink-edit the base application load modules to include the new and updated application programs.

Your installation may choose to add names to the base application load module list as soon as the corresponding fixes have been tested - this minimizes the need for operator intervention at ALCS restart. Later, at some convenient time, your installation can rebuild (that is, relink-edit) the base application load modules and remove the update load modules that are no longer required from the list.

Special considerations for high-level language programs

Assembler and SabreTalk application programs have a simple structure. Each ECB-controlled program is a single source module. The source and object module names are (usually) the same as the name of the ECB-controlled program (except that the name of the source and object modules can include the version number). Figure 104 on page 153 shows this schematically.

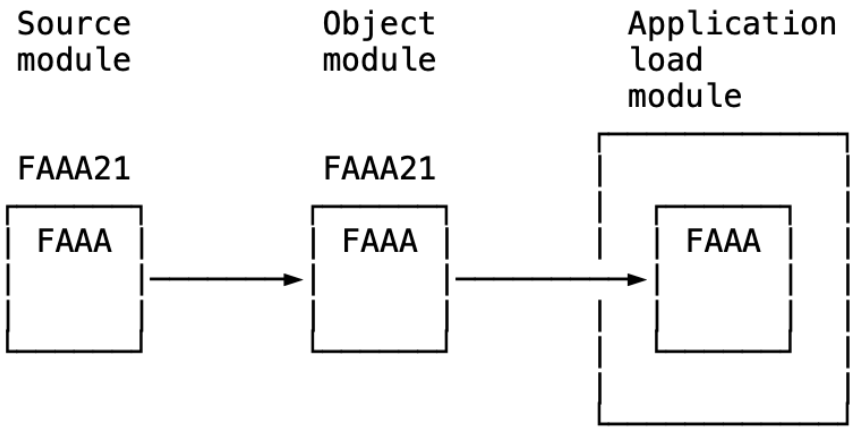


Figure 104. Program structure - assembler and SabreTalk

For assembler and SabreTalk application programs, you decide which ECB-controlled programs to include in an application load module at the time you link-edit the load module.

High-level language application programs can have a more complex structure. You can develop several source modules, compile them (into the same number of object modules), and then combine them into a single ECB-controlled program. Later sections of this chapter explain how you combine high-level language object modules into ECB-controlled programs. Figure 105 on page 154 shows this schematically.

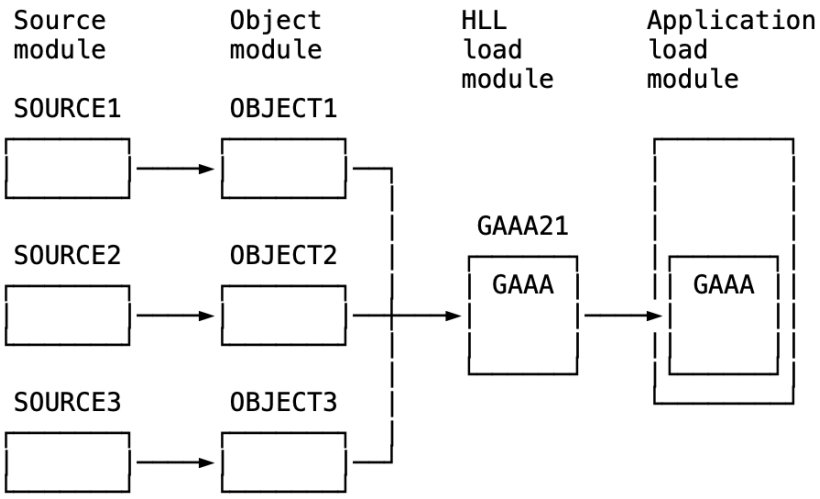


Figure 105. Program structure - single high-level language program

Note that the process of building the high-level language ECB-controlled program GAAA21 results in an HLL load module. You can use this HLL load module as an ALCS application load module. But you can also (as shown in Figure 105 on page 154) build an ALCS application load module that contains assembler and SabreTalk application programs and the HLL application program. To do this, you link-edit the assembler and SabreTalk *object* modules with the HLL *load* module to produce a single ALCS application load module.

You can include any number of assembler and SabreTalk object modules in an ALCS application load module, but you can only include *one* HLL load module in each ALCS application load module. If you want to include more than one high-level language ECB-controlled program in an ALCS application load module, you must build an HLL load module that contains the required programs. Figure 106 on page 155 shows this schematically.

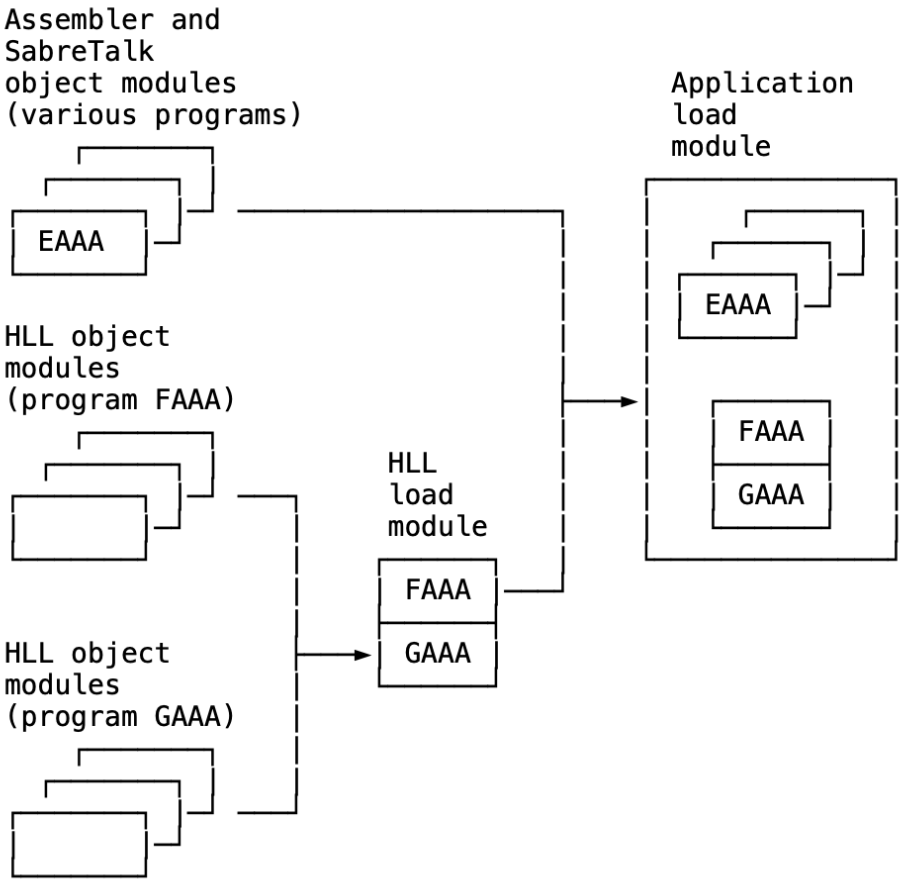


Figure 106. Program structure - multiple high-level language programs

Special considerations for SQL programs

If you have an application that includes SQL statements, you must build a DB2 application plan for the application. When the application executes, the application plan identifies the SQL statements that your application is authorized to use. If you change an application that includes SQL statements, you must build a new plan.

The process of building an application plan is described in "DB2 for z/OS precompile" on page 161 and "How to build application plans (DB2 BIND)" on page 169. During this process, you create a database request module (DBRM) for each source module. When you build an application load module, you combine the DBRMs for the programs in the load module to create the application plan.

When an application executes its first SQL statement, this starts an SQL thread. ALCS and DB2 associate the application with the plan. This association continues until the application ends the thread, with an explicit or implied `COMMIT` or `ROLLBACK`.

An application that uses SQL can include several programs that call each other using static or dynamic program linkage. The thread continues across these linkages. You will probably want to keep all these programs in the same load module because they all use the same application plan.

Note that when you change one of these programs and build an update load module, you must also build a new application plan. You need to include *all* the programs that use the plan in the update load module, even if you have not changed them all.

When you decide which programs to include in a load module you may have more than one application that uses SQL. You will probably want to keep different applications in different load modules to help you keep the corresponding application plans separate.

By default, ALCS assumes that the application plan for an application load module has the same plan name as the load module name. But your installation may have an installation-wide exit that associates a plan with the application in a different way. This allows you to have different plans for different applications in the same load module.

Static and dynamic program linkage

ALCS supports two kinds of program linkage: **dynamic program linkage** and **static program linkage**.

Dynamic program linkage

In ALCS and TPF, dynamic program linkage is usually called the ENTER/BACK mechanism. The calling program uses an ALCS ENTER service such as ENTRC (`entrc`). The called program returns using the ALCS BACK service BACKC (explicit or implied `return`).

ALCS dynamic program linkage uses ALCS monitor services to transfer control between programs. This allows ALCS to decide at execution time which version of the target program to use (ALCS can have several versions of the same program in memory at one time). Also ALCS ENTER/BACK monitor services can transfer control between programs in different application load modules.

Figure 107 on page 156 shows this schematically.

Using ALCS dynamic program linkage has several consequences, including:

- You may have a program that is called by many other programs. If the calling programs use dynamic program linkage, they can all use the same copy of the program even if they are in several different application load modules.
- You may need to load a new version of a program; for example, to apply a fix. To do this, you load (ZPCTL LOAD command) an application load module that contains the new version of the program.

  If callers of the program use dynamic linkage you do not need to include them in the load module. Similarly, you do not need to include anything that the program calls using dynamic program linkage.

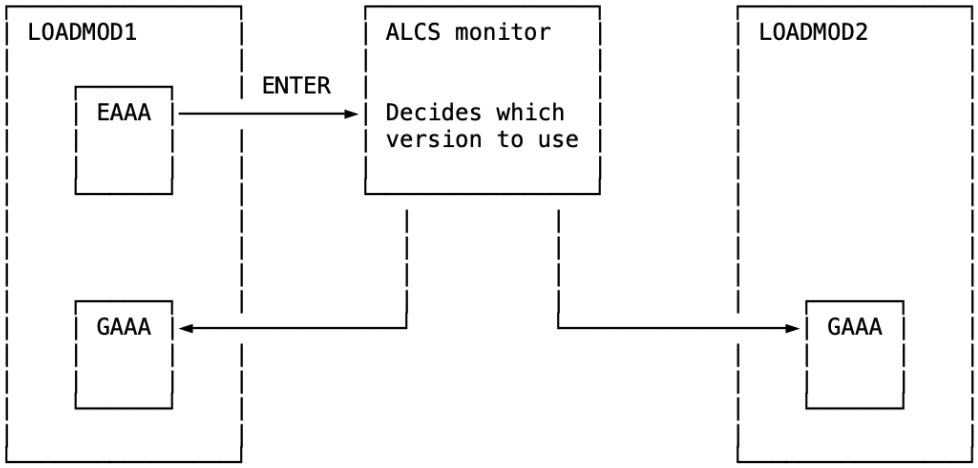


Figure 107. ALCS dynamic program linkage - schematic

Static program linkage

In ALCS, static program linkage is usually called the CALL/RETURN mechanism. The calling program uses the MVS CALL macroinstruction (assembler language), or a conventional procedure or function call statement (high-level language). The called program returns using the MVS RETURN macroinstruction (assembler language), or an implicit or explicit return statement (high-level language).

Static program linkage does not use any ALCS monitor services. Static linkage is not "aware" that there can be many versions of a program. It always transfers control to the version in the same load module.

Figure 108 on page 157 shows this schematically.

Using ALCS static program linkage has several consequences, including:

- You may have a program that is called by many other programs. If the calling programs use static program linkage, then either:
  - All the calling programs must be in the same load module, together with the called program,

    or:
  - Every load module that contains a calling program must also contain a copy of the called program.
- You may need to load a new version of a program; for example, to apply a fix. To do this, you load (ZPCTL LOAD command) an application load module that contains the new version of the program.

  If callers of the program use static linkage you must include them in the load module. Similarly, you must include anything that the program calls using static program linkage.

Figure 108. ALCS static program linkage - schematic

Linkage independence in high-level language applications

In high-level languages, you can write application programs in such a way that it does not matter what type of linkage (static or dynamic) is used at execution time.

For C language programs, you can develop header files that use the C macro facility to "convert" a conventional function call into an ALCS dynamic program linkage, for example an `entrc`. For other high-level language programs, you cannot use dynamic program linkage directly. Instead, you must write a callable service routine (in C or assembler). Your high-level language program uses static linkage (CALL) to the callable service routine. The callable service routine can then use dynamic program linkage (ENTER).

A **called** high-level language program can use a conventional return statement (or implied return) to return to a calling program for both types of linkage. The programmer does not need to know the type of linkage that its caller uses. If the calling program uses the ALCS ENTER service ALCS intercepts the return and performs the BACK service as required.

Deciding which type of linkage to use

Static linkage is faster (uses fewer instructions) than dynamic linkage.

Dynamic linkage is more flexible than static linkage. It's easier to apply fixes, enhancements, and so on when programs use dynamic program linkage.

You cannot directly invoke ALCS dynamic linkage services from languages other than assembler, C, and SabreTalk. Instead, you must invoke dynamic linkage services (if you need them) from within a callable service routine (which is statically linked to the calling programs).

Generally, static linkage is best suited for:

- Small code fragments (such as callable service routines) which are unlikely to need changes. In this case it doesn't matter much that you might need many copies (one for each load module).

  Or:

- Highly specialized routines that are only ever likely to have a small number of closely related callers. In this case, you can probably put all the callers in one load module without undue problems.

TPF compatibility:

TARGET (TPF) **only** supports dynamic linkage. ISO-C support in TPF supports both dynamic and static linkage.

Program names, transfer vectors, and high-level language entry points

An ALCS dynamic linkage program call (ENTER service) uses a 4-character name to identify the called program. Any application program that you can link to with an ENTER service must have a program header. ALCS uses information in the program header to associate the 4-character name with the program (remember that the load module name does not identify the programs that the load module contains).

ALCS allows dynamic linkage to multiple entry-points in the same program. Each of these entry points is called a **transfer vector**, and has a 4-character name that you specify to the ENTER service. Programs with transfer vectors have a program header extension that associates the 4-character transfer vector names with the corresponding entry points.

Figure 109 on page 159 shows two programs. EAA1 has a single entry point. To enter this program, you might code the following in an assembler program:

```
          ENTRC EAA1              ENTER PROGRAM EAA1
```

Program FAA1 has multiple entry points called FAA1, FAA2, and so on. To enter this program at entry point FAA2, you might code the following in an assembler program:

```
          ENTRC FAA2              ENTER PROGRAM FAA2
```

(Note that the calling program does not need to "know" that FAA2 is a transfer vector of program FAA1.)

Figure 109 on page 159 shows program EAA1 with no header extension because EAA1 has only a single entry point. It shows program FAA1 with a header and a header extension. The header extension contains information about where entry points FAA1, FAA2, and so on are within the FAA1 program.

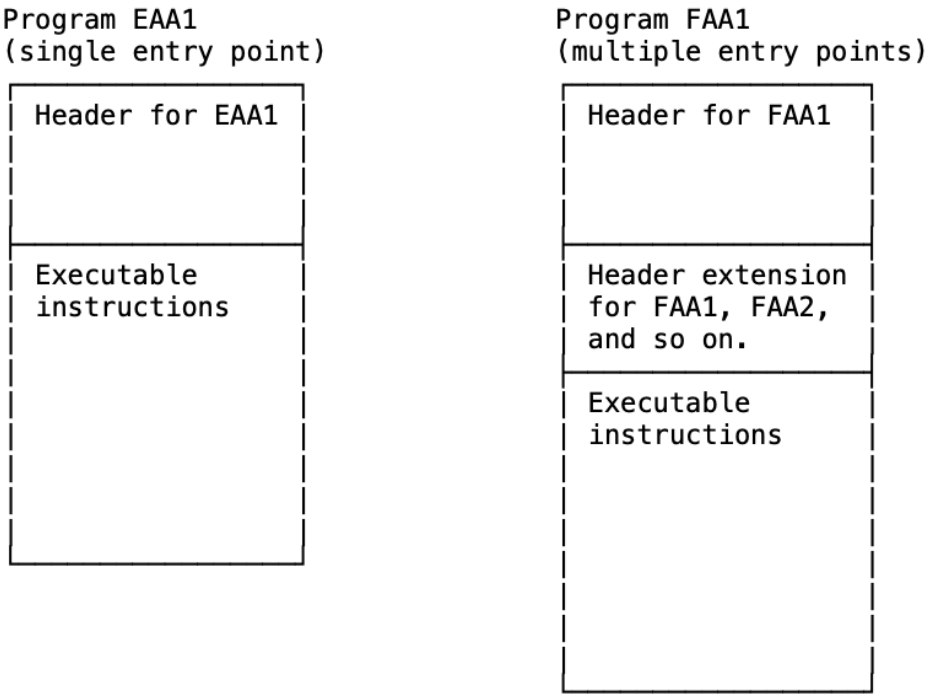


Figure 109. ALCS application program headers

Creating the program header - assembler and SabreTalk

For assembler programs, the BEGIN macroinstruction generates the program header, and the TRANV macroinstruction generates an entry in the program header extension (you code one TRANV macroinstruction for each entry point).

Note: You **do not** use the assembler language ENTRY instruction to define entry points for ALCS dynamic program linkage. But you do use the ENTRY instruction to define entry points (if any) for static program linkage.

For SabreTalk programs, the SabreTalk compiler generates an assembler language source module that includes a BEGIN macroinstruction that generates the program header.

Creating the program header - high-level languages

For high-level language programs, you do not define the 4-character names for ALCS dynamic program linkage within the program source code - there are no high-level language equivalents of the ALCS BEGIN and TRANV macroinstructions.

Instead, you use the ordinary syntax of the high-level language to define conventional entry points. You can use these conventional entry points for static program linkage. But if you want to use them for ALCS dynamic program linkage, you must use the ALCS offline utility program DXCBCLPP to create the program header and program header extension.

To do this, you prepare an **entry points definition file**. This file contains input statements for program DXCBCLPP. The input statements provide 4-character names for the high-level language program entry points that you want to use with ALCS dynamic program linkage.

Program, DXCBCLPP produces an assembler source module for the program header. You must then assemble this source code and prelink the resulting object module with the program object module (which the compiler produces).

Figure 110 on page 160 shows this process schematically. The actual program preparation process is more complex than this. For example, Figure 110 on page 160 does not show some additional output that program DXCBCLPP produces for use by the prelink process.

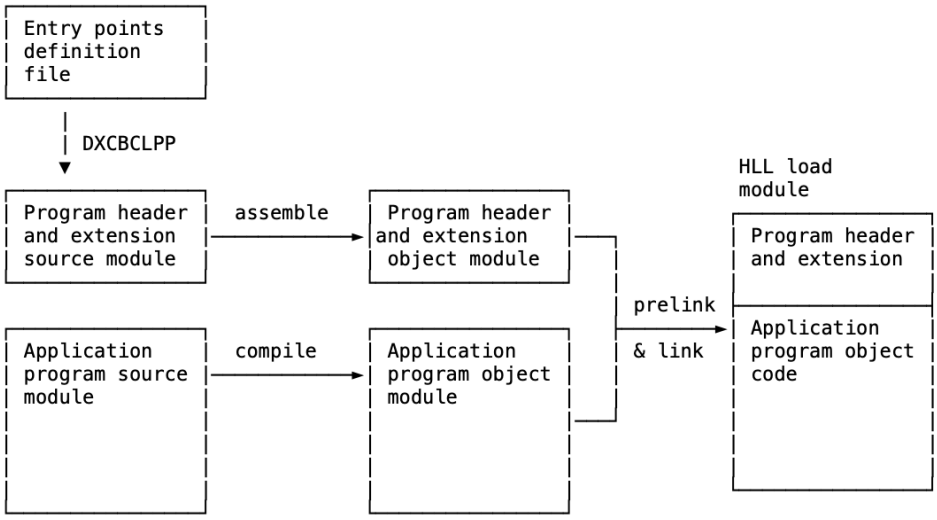


Figure 110. Preparing the ALCS application program header for a high-level language application program - schematic

Long function names in C language programs

The C language allows function names longer than the eight-character external names that the MVS linkage editor supports.

To resolve this, the z/OS Language Environment® Prelinker (and the MVS linkage editor) or the binder allow you to map long names into short names for object modules produced by the compiler. You can also use the compiler directive `#pragma map`. When a C language program calls an assembler program using static program linkage you can either use the short name of the program (8 characters or less) or map a suitable long name, using the `#pragma map` directive. For example, to call program PROGRAM3 by the name `calculate_top`, you could code the following:

```
#pragma map(calculate_top, "PROGRAM3")
```

You must repeat the `#pragma map` in each source module in which you want it to be effective (or put it in a header file which you `#include` in each source module).

Overview of program preparation

About this task

This section presents an overview of the process that you use to produce an ALCS application load module, starting from source modules. This process is slightly different for the different languages. It comprises some or all of the following steps:

Procedure

1. SabreTalk compile (SabreTalk programs).

   The SabreTalk compiler generates an assembler source module.
2. DB2 for z/OS precompile (programs containing SQL).

If your application program contains SQL statements, you must process it with the DB2 for z/OS precompiler. The precompiler replaces the SQL statements with assembler language or high-level language instructions.

3. Assemble (assembler or SabreTalk programs) or compile (other languages).

   If your application program contains SQL statements, then you actually assemble or compile the output from step "2" on page 160.

4. Build program header (high-level language programs).

   For languages other than assembler and SabreTalk, you must use program DXCBCLPP to build program headers, as follows:

   a) Prepare the entry points definition file, and run the ALCS program DXCBCLPP.

   b) Assemble the output from step "4.a" on page 161.

   Note: This step identifies *all* the high-level language programs that you want to include in the application load module that you are building.

5. Prelink (high-level language programs).

   Run the prelinker using the prelinker control statements produced by "4.a" on page 161. This creates an object module.

6. Link-edit HLL load module (high-level language programs).

   Link edit the object module produced by step "5" on page 161 with the program headers from step "4.b" on page 161. This produces an HLL load module.

   The above steps produce object modules for assembler and SabreTalk application programs (output from step "3" on page 161) and HLL load modules (output from step "6" on page 161). The remaining steps build an ALCS application load module. The application load module can include one HLL load module, or one or more assembler or SabreTalk object modules, or both.

7. Link-edit ALCS application load module.

   Link-edit one or more object modules (if required) from step "3" on page 161 with the HLL load module (if required) from step "6" on page 161. This creates an ALCS application load module.

   If you do not want to include any assembler or SabreTalk programs, you can use an HLL load module as an ALCS application load module without performing this link-edit step.

8. DB2 for z/OS bind (load modules containing SQL).

   If your application load module includes programs that contain SQL statements, you must use the DB2 for z/OS `BIND` command to build the DB2 for z/OS application plan for the load module.

How to assemble or compile application programs

For all application programs, you must assemble or compile the source modules to produce object modules.

SabreTalk compile

Refer to your SabreTalk compiler documentation for information about how to compile SabreTalk programs. The output from this compilation is an assembler source module that you process the same way as any other assembler language ALCS application program.

DB2 for z/OS precompile

If your application program includes any SQL statements you must precompile the program, using the DB2 for z/OS precompiler.

The DB2 precompiler produces an output source module that is the same as the input source module, except that SQL statements are replaced by instructions or statements in the appropriate language. For

example, in assembler language programs, the DB2 precompiler replaces SQL statements with assembler DSECTs, BALR instructions, and so on.

The DB2 precompiler also produces a data base request module (DBRM). The DBRM is input to the bind process, described in "How to build application plans (DB2 BIND)" on page 169.

The DB2 precompile process is described in *DB2 for z/OS Application Programming and SQL Guide*.

Using EXEC SQL INCLUDE

You run the DB2 precompiler **before** you assemble or compile your program. Therefore you cannot use assembler COPY, C `#include` directives, and so on, to copy (include) SQL statements in your program.

Instead, you can use the EXEC SQL INCLUDE statement.

Assemble

You can use the IBM Assembler H or the IBM High Level Assembler to assemble ALCS application programs, including assembler source modules produced by the SabreTalk compiler.

Note that if your program contains SQL statements, you assemble the source module produced as output by the DB2 precompiler.

You should direct the output object module to a suitable library. This object module is input to the link-edit process, described in "How to link-edit application load modules" on page 169.

For information about assembling programs, refer to the books provided with the assembler that you are using.

The assembler must have access (usually the SYSLIB DDNAME) to the libraries that contain definitions for the macros that your application uses. Typically these include:

- Macrodefinitions provided with ALCS. This is the library whose dataset name has the final qualifier DXCMAC1.
- Your own macrodefinitions for application program use.
- Macrodefinitions provided by vendors (if any). If you use the IBM TPFDF product, this includes the TPFDF macrodefinitions.
- Macrodefinitions provided with MVS and associated products. This includes the library usually called SYS1.MACLIB and the macro library provided with WebSphere MQ for z/OS.

IBM recommends that you use the following assembler options:

ALIGN
: This option is required for all the assemblies of ALCS applications. NOALIGN can result in malfunction of the program.

DECK or OBJECT
: Produces the output object module.

LIST
: Produces an assembly listing. Specify NOLIST if you do not require a listing.

NOLINK
: Each application load module can contain several application programs. Hence option LINK is not suitable for assembling application programs.

RENT
: Verifies that the program is reentrant.

RLD
: Produce a relocation dictionary listing. Use the relocation dictionary to check that the application program does not contain any relocatable address constants.

  TPF compatibility:

  Relocatable address constants are not allowed in TPF applications.

SYSPARM(DXCDFO)
The ALCS BEGIN macro defaults the addressing mode and serialization parameters to AMODE=24,SHR=ALL,XCL=ALL. Application programs that originate from TPF systems do not specify these parameters, because the TPF BEGIN macro does not support them. If these programs can run in 31-bit addressing mode, and do not require serialization, specify the assembler option SYSPARM(DXCDFO) as an alternative to updating the BEGIN macroinstruction. This option changes the ALCS BEGIN macro defaults to:

```
AMODE=31,SHR=NONE,XCL=NONE
```

SYSPARM(DXCD31)
This option changes the BEGIN default parameters to:

```
AMODE=31,SHR=ALL,XCL=ALL
```

USING(MAP)
This option is only valid for the High Level Assembler. It produces a USING map at the end of the assembly. Specify USING(NOMAP) when you do not require a USING map.

XREF(SHORT)
XREF produces a cross-reference listing. Although a large number of symbols are defined in ALCS application programs, particularly in the BEGIN macro expansion, only a few of these symbols are referenced in any one program. Use the XREF(SHORT) option to save paper. Specify NOXREF when a cross-reference listing is not required.

Compile

You use the appropriate IBM compiler to compile ALCS application programs.

Note that if your program contains SQL statements, you compile the source module produced as output by the DB2 precompiler.

You should direct the output object module to a suitable library. This object module is input to the prelink-edit process, described in "How to link-edit application load modules" on page 169.

For information about compiling programs, refer to the books provided with the compiler that you are using.

C compiler options

The compiler must have access (usually the SYSLIB DDNAME) to the libraries that contain header files that your application uses. Typically these include:

- Header files provided with ALCS. This is the library whose dataset name has the final qualifier DXCHDR1.
- Your own header files.
- Header files provided by vendors (if any). If you use the IBM TPFDF product, this includes the the TPFDF header files.
- Standard header files provided with WebSphere MQ for z/OS.
- Standard header files provided with the compiler.

IBM recommends that you use the following C compiler options: (When you use the C compilation option of the ISPF panels, these parameters are automatically incorporated into the JCL.)

FLOAT(NOAFP)
Required. The FLOAT(NOAFP) compiler option must be used to limit the compiler to generating code using only floating-point registers 0, 2, 4, 6.

LONGNAME
Use this option when you are using external function names longer than 8 characters in your C language program. For more information on LONGNAME, see the *z/OS XL C/C++ User's Guide*.

OPTIMIZE(0) or OPTIMIZE(1) or OPTIMIZE(2)
Optional. Sets the optimization level of the compiler.

RENT
Generates reentrant object module.

Do **not** use any of the following options:

```
ALIAS, EXECOPS, TARGET(TPF)
```

COBOL compiler options

The compiler must have access (usually the SYSLIB DDNAME) to the libraries that contain copy files that your application uses. Typically these include:

- Your own copy files
- Copy files provided with WebSphere MQ for z/OS.

IBM recommends that you use the following COBOL compiler options: (When you use the COBOL compilation option of the ISPF panels, these parameters are automatically incorporated into the JCL.)

DECK or OBJECT
Produce the output object module. (The ALCS ISPF panels use the option OBJECT)

LIB
Allows COPY, BASIS, and REPLACE statements.

NODYNAM
Do not load CALLed programs dynamically at runtime.

RENT
Generates reentrant object module.

Do **not** use any of the following options:

```
DATA(24), DYNAM, OUTDD
```

PL/I compiler options

IBM recommends that you use the following PL/I compiler options:

OPTIMIZE(0) or OPTIMIZE(1) or OPTIMIZE(2)
Optional. Sets the optimization level of the compiler.

REENTRANT
Generates reentrant object module.

How to build program headers and prelink high-level language programs

You cannot directly link-edit object modules from the IBM compilers into ALCS application program load modules. Instead, you must build an ALCS program header and use the z/OS Language Environment Prelinker to combine:

The program header
The program object code from the compiler
Some program object code provided with ALCS

into a single object module for input to the link-edit process, described in "How to link-edit application load modules" on page 169.

Figure 111 on page 165 shows this process schematically. Note that this figure shows more detail than Figure 110 on page 160. In particular:

- In addition to the source module for the program header and extension, the DXCBCLPP program also produces prelink-edit control statements that are input to the prelink-editor.
- The process can produce a prelinked object module that contains object code from more than one application program source module.
- The process can produce a prelinked object module that contains more than one program header, each with its extension.
- In addition to the program header and extension, and the application object code, the prelinked object module also contains some object code provided with ALCS.

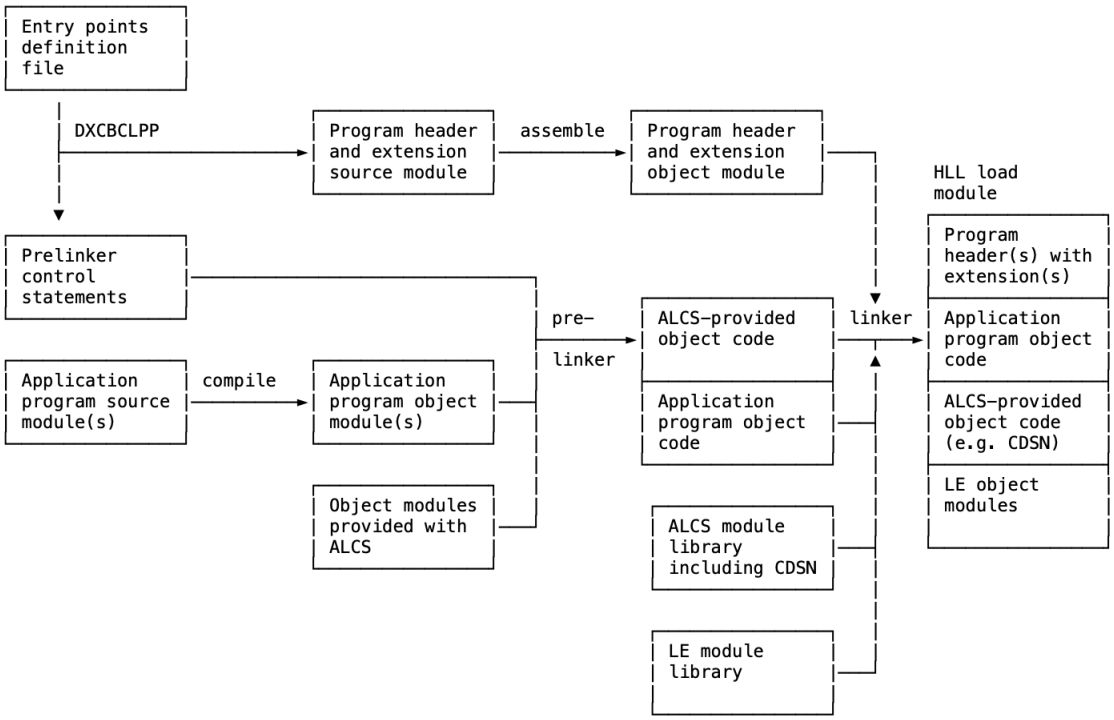


Figure 111. Preparing the program header and prelinking a high-level language program

The entry points definition file identifies the high-level language object modules that you want to include in an HLL load module. It also associates 4-character program and transfer vector names with these object modules. The transfer vector names allow you to call entry points in the high-level language program (or programs) using ALCS dynamic program linkage.

An HLL load module can include one or more ECB-controlled programs. Each of these ECB-controlled programs can comprise one or more high-level language object modules.

If you do not want to include any assembler or SabreTalk programs, you can use an HLL load module as an ALCS application load module. Alternatively, you can link-edit the HLL load module with assembler or SabreTalk program object modules to create an ALCS application load module.

You cannot link-edit more than one HLL load module into an ALCS application load module. Instead, you must build a new HLL load module that contains all the programs that you want in the application load module. For example, you might have one HLL load module that contains the ECB-controlled program HLL1, and another HLL load module that contains the ECB-controlled program HLL2. If you now want to include HLL1 and HLL2 in a single application load module, you must first create a new HLL load module that contains both programs.

Program header build - entry points definition file

The entry points definition file contains information that program DXCBCLPP uses to build one or more program headers, each with a program header extension.

ALCS requires a program header so that it can recognize the start of a program within an application load module.

The program header extension associates 4-character names with entry points in the high-level language program. You use these 4-character names with the ALCS dynamic program linkage services (ENTER).

Note that for high-level language programs, you **always** have a program header extension - even if there is only one entry point that you can call using ALCS dynamic program linkage services.

An entry points definition file is a physical sequential MVS data set (or it can be a member of a PDS or PDS-E - you may want to keep your entry points definition files in a source statement library). It comprises 80-byte records in fixed or fixed-blocked record format.

Entry points definition file - syntax

Each record of an entry points definition file contains a control statement, which can be a comment or a BEGIN or TRANV statement.

Each statement comprises the following fields:

Operation
Identifies the statement type. You must include one or more space (blank) characters before the operation field.

Operands
Follows the operation field. At least one space (blank) character must separate the operation field from the operand field.

Comments
Follows (optionally) the operand field. At least one space (blank) character must separate the comments field (if any) from the operands field.

The comments field is not shown in the syntax diagrams which follow.

You can enter all characters in upper or lower case.

Entry points definition file - comment statement

You can include comment statements anywhere within the entry points definition file. The asterisk must be in column 1.

The syntax is:

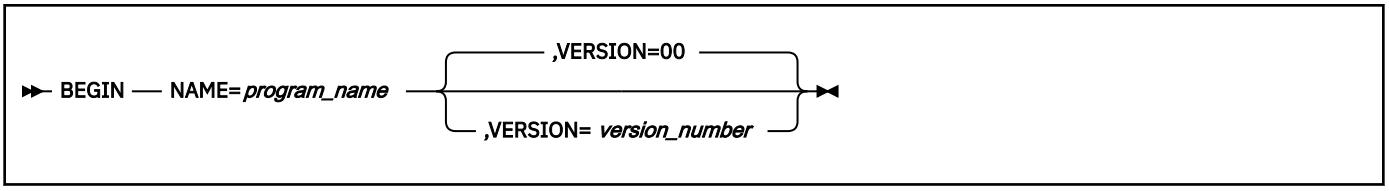
Where:

comment_text
Any comment text.

Entry points definition file - BEGIN statement

Include one BEGIN statement for each ECB-controlled program that you want to include in the HLL load module. The first non-comment statement in the entry points definition file must be a BEGIN. You can include more than one BEGIN statement in an entry points definition file - you must follow each BEGIN statement with at least one TRANV statement.

The syntax is:



Where:

program_name
4-character name for the program header. This must be the same as *program_name* on the first or only TRANV statement that follows this BEGIN statement.

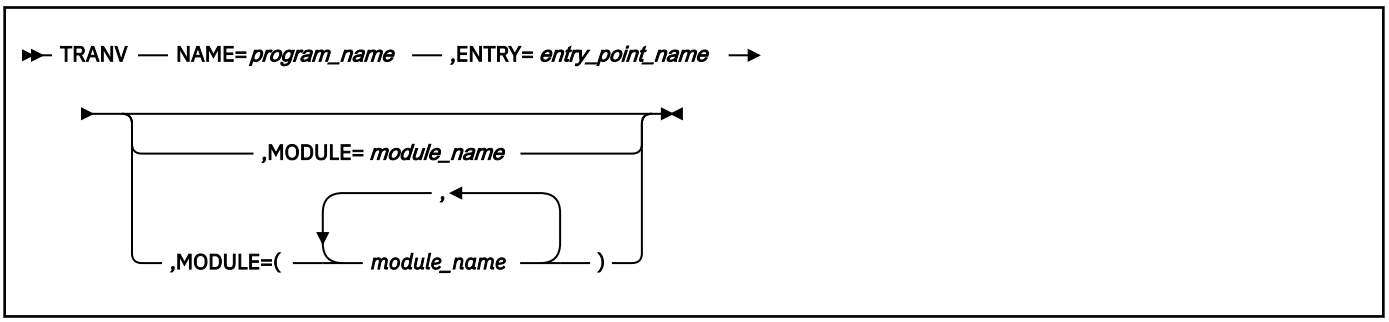
version_number
2-character alphanumeric version number for the program header.

Entry points definition file - TRANV statement

Include one or more TRANV statements following each BEGIN statement in the entry points definition file. Each TRANV statement performs the following functions:

- It identifies an input object module (from the compile) to be included in the output object module (from the prelink-edit).
- It identifies an entry point within the input object module that you plan to call using ALCS dynamic program linkage (ENTER).
- It specifies the 4-character name that you plan to use in ALCS dynamic program linkage calls (ENTER) to that entry point.

The syntax is:



Where:

program_name
4-character name for the entry point for ALCS dynamic program linkage (ENTER).

entry_point_name
Entry point name, up to 8 characters. This is the routine that you want to be executed when ALCS executes an ENTER for the *program_name* above.

module_name
Name of the input object code module (from the compile) that contains the entry point, up to 8 characters. If you omit this operand, DXCBCLPP assumes that the module name is the same as the entry point name (*entry_point_name*).

You can specify a list of module names, separated by commas (,) and enclosed in parentheses. You can spread these over several lines provided the last character on a line is a comma (see Figure 112 on page 168).

Entry points definition file - example

ALCS includes an ISPF panel to build an application load module. If you prefer to build your own JCL for the process you can use the panel to create a sample job.

Note for ISPF panels:

If you use ISPF panels to generate your high level language application load module, it is necessary to provide a list of the entry point definition files which should be provided as input to the DXCBCLPP program. In the example below, the list would contain only one entry point definition file name; however, it may contain multiple entry point definition file names, each on a separate line.

Figure 112 on page 168 shows an example entry points definition file. The example creates two program headers, one for SZOA version 00, and one for SZOB version 00.

The program header and program header extension for SZOA specify three entry points that you can use with ALCS dynamic program linkage (ENTER). All three entry points are in the input object module SZ0A000. To ENTER module SZOA000 at entry point SZ0A00 you specify the 4-character name SZOA, and so on.

The program header and program header extension for SZOB also specify three entry points that you can use with ALCS dynamic program linkage. But in this case, the three entry points are in three different input object modules - SZ0B000, SZXB000, and SZYB000. To ENTER module SZOB000 at entry point SZ0B00 you specify the 4-character name SZOB, and so on.

```
BEGIN NAME=SZOA,VERSION=00
TRANV NAME=SZOA,ENTRY=SZOA00,MODULE=SZ0A000
TRANV NAME=SZXA,ENTRY=SZXA00,MODULE=SZ0A000
TRANV NAME=SZYA,ENTRY=SZYA00,MODULE=SZ0A000
BEGIN NAME=SZOB,VERSION=00
TRANV NAME=SZOB,ENTRY=SZOB00,MODULE=SZ0B000
TRANV NAME=SZXB,ENTRY=SZXB00,MODULE=(SZXB000,SZOB001)
TRANV NAME=SZYB,ENTRY=SZYB00,MODULE=(SZYB000,
          SZYB001,SZYB002,SZYB003)
```

Figure 112. Example entry points definition file

Program header build - DXCBCLPP program

Program DXCBCLPP processes the entry points definition file and produces an assembler language source module and a data set containing control statements for the z/OS Language Environment Prelinker.

You must provide the following JCL DD statements:

INFILE
The data set that contains the entry points definition file.

OUTHDR
The data set where program DXCBCLPP places the output assembler source module for the program headers and extensions.

OUTLNK
The data set where program DXCBCLPP places the output prelink-edit control statements.

After you run program DXCBCLPP, you must assemble the output assembler source module. You can use the IBM Assembler H or the IBM High Level Assembler.

For information about assembling programs, refer to the books provided with the assembler that you are using.

You should direct the output object module to a suitable library. This object module is input to the link-edit process, described in “How to link-edit application load modules” on page 169.

Building the HLL load module

After you run program DXCBCLPP and assemble the source module for the program headers and extensions, you must use the IBM prelink-editor and the link-editor (or binder) to build an HLL load module. Figure 111 on page 165 shows this process.

ALCS includes an ISPF panel to build an HLL load module. If you prefer to build your own JCL for the process, you can use the panel to create a sample job.

If you do not want to include any assembler or SabreTalk programs, you can use the HLL load module as an ALCS application load module. Alternatively, you can link-edit the HLL load module with assembler or SabreTalk program object modules to create an ALCS application load module. See "How to link-edit application load modules" on page 169.

How to link-edit application load modules

To build an ALCS application load module, you use the link-editor or binder program to combine one or more assembler or SabreTalk object modules and (optionally) an HLL load module. See "Choosing what programs to include in a load module" on page 150, "Special considerations for high-level language programs" on page 153, and "Special considerations for SQL programs" on page 155.

ALCS includes an ISPF panel to build an application load module. If you prefer to build your own JCL for the process, you can use the panel to create a sample job.

How to build application plans (DB2 BIND)

When an application load module contains programs that include SQL statements you must build a DB2 application plan before you run the programs. See *DB2 for z/OS Application Programming and SQL Guide*.

For each source module that contains SQL statements, you create a database request module (DBRM) using the DB2 precompiler. See "DB2 for z/OS precompile" on page 161. Then you must bind the DBRM using the DB2 BIND subcommand. You can bind a DBRM directly to a DB2 application plan, or you can bind it to a DB2 package which is included in a plan. A package contains just one DBRM. You can group packages together into a collection. A plan can contain DBRMs bound directly, a list of packages or collections of packages, or any combination.

Use the DB2 BIND PACKAGE subcommand to bind a DBRM to a package. Use the DB2 BIND PLAN subcommand to bind a DBRM to a plan or to include a list of packages or collections of packages in a plan. See *DB2 for z/OS Command Reference*.

When you change a source module that contains SQL statements, you create a new DBRM using the DB2 precompiler and rebind the DBRM.

If you have multiple versions of the source module, you can use the VERSION option of the DB2 precompiler to identify the program with a specific version of a package.

If you intend to load a new version of a program in a test load module, you can bind the program to a test version of its package and build a new DB2 application plan which includes this package together with the collection of packages for all other programs in the application suite.

By default, ALCS assumes that the application plan for an application load module has the same plan name as the load module name. But your installation may have an installation-wide exit that associates a plan with the application in a different way. This allows you to have different plans for different applications in the same load module.

Chapter 9. Program testing and problem determination

This chapter describes the tools you can use to test and debug your application programs.

Test database facility

New and changed application programs can contain errors capable of damaging the real-time database. You must test them in a way that avoids damage to the existing database, but still provides realistic testing conditions.

One method would be to copy all or part of the existing database to a test database. However, the volume of test data used for testing different programs could exceed the hardware available. This is especially true because ALCS allows any number of test systems to be in operation at one time, so that application programmers can test their different programs simultaneously and independently.

A better method is to use the ALCS **test database facility**.

How the test database facility works

The ALCS test database facility gives a program read-only access to a database. When the program updates a record, ALCS writes it to a separate data set (the **test data set**) instead of updating the database from which it was read. When the program next reads the same record, ALCS gets it from the test data set. Thus the application seems to have normal read-write access to the database, which however remains unchanged.

An example is shown in Figure 113 on page 170. At the start of testing the application had read-access to records A through to Z on the live database. If the test dataset facility is being used, ALCS services any write requests from the application by writing these records into the test dataset. In the example, the application has written records B, D, F, and H. When the application requests ALCS to read of any of these records (B, D, F, or H), ALCS reads them from the test dataset. The existing database cannot be damaged by the application.

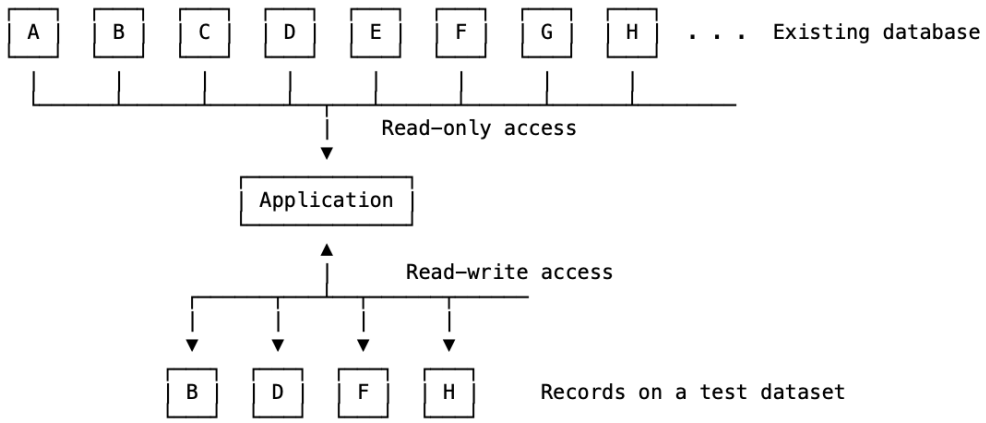


Figure 113. The test database facility

The ALCS test database facility makes it safe to test programs with a production database. However, most users prefer to use this facility with a stable test database that meets all the testing requirements of the programmers. This results in minimum impact on production work, and does not create excessive demands on DASD space.

Benefits

Using a test database allows you to:

Recover from database damage

During a test, an application program can write bad data that makes the database unusable. If this happens, it is not necessary to restore the test database. Deleting all the records in the test data set has the same effect. An easy way of doing this is to delete the test data set and allocate a new (empty) data set.

If a series of tests takes several days or weeks, it is possible to back up the test data set at convenient times. If a particular test writes bad data, it is easy to recover by restoring a backup of the test data set.

Share the test database with other applications

Several programmers may need a test database. The test database facility allows them all to share the same copy of the test database. If each programmer (or group of programmers) has a separate test data set, they can run tests simultaneously without interfering with one another. Also, programmers who are running a series of tests do not put other tests at risk by changing or damaging their data.

You can create a new test data set by copying an existing test data set. In this way, you can run tests against a database that already includes updates from previous testing. For example, one programmer might initialize a number of records, then a second programmer might copy the test data set. Both programmers can then run tests that use these records.

Analyze database updates

You can use standard MVS utility programs to display the contents of a test data set. This allows you to identify which records your testing has updated and to check the contents of the updated records.

Conversational trace

Conversational trace is intended to help you debug application programs. It lets you see the effect of each single program instruction as the entry processes a message that you input at a terminal.

Conversational trace is enabled (or disabled) by the operator entering a command from the Prime CRAS. When conversational trace is enabled you can start the trace by entering a ZTRAC command from any CRAS terminal. Depending on the ZTRAC command parameters, you can trace

- the next input message from your own terminal (this is the usual case), or
- the next input message from another terminal which you specify in the ZTRAC command (this case is called "remote terminal trace"), or
- an entry that is created by the system (this case is called "asynchronous trace").

Conversational trace monitors any program used by the entry as it processes the message. Unless you specify otherwise, conversational trace comes to a temporary halt whenever it encounters a monitor-request macro in the program. (The monitor-request macro can be one coded in an assembler program or can be a monitor-request macro resulting from a C function call.)

ALCS displays details of the monitor-request macro on the terminal before it processes it. ALCS can optionally include extra information in this display. The extra information can include ECB contents, register contents and so on.

You can now take various actions at the terminal by using trace subcommands. For example, you can:

- Request ALCS to supply additional information (for example, from storage areas)
- Set up particular values in storage or registers
- Branch to a different instruction
- Start processing one instructions at a time (step-by-step processing)

- Exit the entry.

For a full list of options, see *ALCS Operation and Maintenance*.

Notes:

1. If your application uses a special 3270 screen format, for example if it uses the ALCS 3270 screen mapping support, you should use remote terminal trace or diagnostic trace (see *ALCS Operation and Maintenance*).
2. Conversational trace can be used to trace ALCS macros issued by C language applications through the C application programming interface routines, but there are restrictions if you use step-by step processing.

Workstation trace and remote debugger

The remote debugger is a source level debugger for C/C++ application programs. You use both Prime CRAS and your local workstation to control the debugger facility.

You enable or disable workstation trace using the ZTRAC command from Prime CRAS. For more information on the ZTRAC command see *Installation and Operations*.

- After you enable workstation trace, the debugger starts wth the next input message from your local terminal that runs a C/C++ program compiled with the `DEBUG(FORMAT(ISO),HOOK(NOBLOCK,NOPATH))` option.
- The compile process creates a dataset with the same name as the C/C++ source suffixed with ".DBG".
- Your ALCS system must have RACF authority to read these datasets.
- The debugger stops when the entry exits and input from the terminal remains inhibited until the debugger stops.

Supply the `DEBUG()` option when compiling C/C++ programs to be debugged. Each such program contains a small header generated using a BAL program that is generated by DXCBCLPP. By specifying `SYSPARM(CDEBUG)` when you assemble the BAL program, and then relinking your C/C++ program, you can turn debugging off for specific programs without having to recompile them.

For example, assume a series of 3 C/C++ programs where the first calls the second and the second calls the third. You can debug the second of the 3 programs without debugging the others due to the use of the `SYSPARM(CDEBUG)` using workstation trace control.

This sample USREID code shows how the debug tool is recognized.

```
*=================================================================*
*        END OF USER MACRO DEFINITIONS                            *
*=================================================================*
         PRINT NOGEN
USREID   CSECT ,                  RESTORE CSECT
         EJECT ,
*=================================================================*
*        USER OBJECT COMMENTS SHOULD BE INSERTED BELOW            *
*=================================================================*
         SPACE 1
*=================================================================*
*        END OF USER OBJECT COMMENTS                              *
*=================================================================*
         EJECT ,
UEID1000 DC    0H'0'
         DXCSAVE SAVE,PUSH,       SAVE CALLERS REGISTERS             -
               ID=USER,           AND CHAIN THE USER SAVE AREA       -
               SIZE=512,          LENGTH DEFINED IN SAVE AREA        -
               WORKREG=R02
         SPACE 1
         DROP  R15                DROP PREVIOUS BASE REGISTER
         USING USREID,R10
         LR    R10,R15            USE R10 AS BASE REGISTER
         SR    R15,R15            SET GOOD RETURN CODE
         ST    R01,PARMSAVE       SAVE INPUT PARAMETER LIST
         SPACE 1
*=================================================================*
```

```
*        START OF USER CODE                                            *
*======================================================================*
         SPACE 1
         L     R02,PARM1
         LTR   R02,R02             IS THIS A DEBUG WS DISPLAY CALL?
         BNZ   UEID1900            NO EXIT WITH OK CODE
WSDEBUG1 EQU   *
         L     R09,PARM2          LOAD ECB ADDRESS
         L     R08,CE1CRP     POINT TO THE LAST CALLED APPLICATION PGM
         SLL   R08,1          SWITCH OFF FIRST BIT
         SRL   R08,1
         TM    7(R08),X'10'        IS IT A C MODULE ?
         BZ    UEID1900            NO -  EXIT
         TM    7(R08),X'04'        IS IT A DEBUG MODULE ?
         BNZ   UEID1900            YES- SHOW THAT WS IS ACTIVE
         LHI   R15,4               NO. TELL C THAT WS IS INACTIVE
         SPACE 1
*======================================================================*
*        END OF USER CODE                                              *
*======================================================================*
         SPACE 1
UEID1900 DC    0H'0'
         DXCSAVE POP,RESTORE,      RESTORE CALLERS REGISTERS           -
               EXCEPT=((R15))      (EXCEPT RETURN INFORMATION)
         BR    R14                 RETURN TO ALCS MONITOR
*
```

When debugging a sequence of C/C++ programs which invoke each other using ALCS dynamic linkage, entrc(), you can activate or deactivate debugging for some programs in the sequence without having to recompile the programs with/without the DEBUG() option. This can be accomplished using an HLASM SYSPARM value and a user exit.

- When you assemble the output from the DXCBCLPP program, specify "SYSPARM(CDEBUG)" as one of the PARM= parameters. This will turn on a bit switch in the ALCS program header.
- Use user exit USRWSTR to interrogate the bit switch and specify if Debug Tool should be activated for a particular program. Refer to the section "The ALCS installation-wide monitor exits" in the *ALCS Installation and Customization* manual for details on coding a USRWSTR user exit.

Appendix A. Sample application: assembler

This example application demonstrates the use of the ALCS assembler language application programming interface (API).

It consists of one file of assembler language source code, named APG1.

Purpose of the application

The application lets the user display information about the IPARS agents that are using ("sined in at") a particular display terminal connected to ALCS. The application retrieves this information from the IPARS agent assembly area (AAA) record allocated for that display terminal.

Application entry conditions

The application program APG1 has two transfer vectors which expect different entry conditions as follows:

Transfer vector APG1

ECB field EBROUT contains the CRI of the originating terminal. The example application retrieves information from the AAA record for the originating terminal, and sends the response to the same terminal.

Transfer vector APG2

ECB field EBROUT contains the CRI of the originating terminal. A message block in CM1CM format is attached to the entry on level D0. The application retrieves information from the AAA record for the terminal whose CRI is in field CM1TXI in the attached message block, and sends the response to the originating terminal.

Application return conditions

General register R01 is used by this program and is not restored. All ECB work areas and data and storage levels are restored before returning from the application.

After sending the response message, this application returns to the calling program by BACKC. If there is no calling program, then the application exits normally.

Application normal responses

If the AAA record for the specified terminal is not initialized, the application gives the following response message:

```
hh.mm.ss Information for Agent CRN-crn

AAA record not initialized
```

Otherwise the normal response is:

```
hh.mm.ss Information for terminal CRN-crn

Agent  Sined-in  Active  Initials  Duty code
-----------------------------------------
A      sss       aaa     ii        dd
B      sss       aaa     ii        dd
C      sss       aaa     ii        dd
D      sss       aaa     ii        dd
E      sss       aaa     ii        dd
```

Where:

sss
yes if the IPARS agent is sined in, otherwise no.

aaa
yes if the IPARS agent area is currently active, otherwise no.

ii
Two initials of the agent currently using the terminal.

sss
Two-character duty code of the agent currently using the terminal.

An example of the normal response message follows:

```
19.25.12 Information for terminal CRN-PYESARE2

Agent  Sined-in  Active  Initials  Duty code
-----------------------------------------
A      no
B      yes       yes     XX        MS
C      no
D      yes       no      AB        SU
E      no
```

Application error responses

This application can issue one of the following system errors. If it does, then the entry terminates without returning to the calling program (if there is one).

A00010 INVALID OR MISSING CRI

Explanation: Self-explanatory.

A00020 ERROR RETURN FROM FACE

Explanation: The ALCS file address compute (FACE) monitor service could not determine a valid file address using the AAA fixed-file record type #WAARI and the terminal ordinal number.

A00030 FINWC ERROR FOR AAA FIXED FILE

Explanation: An error occurred when the application issued a FINWC monitor-request macro to find the AAA fixed-file record for the terminal.

A00040 FINWC ERROR FOR AAA POOL FILE

Explanation: An error occurred when the application issued a FINWC monitor-request macro to find one of the suspended AAA pool-file records for the terminal.

A00050 INVALID CRI IN EBROUT

Explanation: Self-explanatory.

Installing the application

Install the application program APG1 into a suitable assembler source library. Assemble it using the IPARS for ALCS macro library as well as the ALCS application macro library. Link-edit it into an application load module for loading on to ALCS.

Running the application

After loading on to ALCS, you can use the application in one of two ways:

1. Use the ALCS ZDRIV command to start transfer vector APG2, for example:

```
ZDRIV APG2 0200EC
```

2. Call APG1 or APG2 from another program, after establishing correct entry conditions, as follows:

   APG1
   EBROUT must contain a valid CRI.

   APG2
   There must be a message block attached on level D0 which contains a valid CRI in field CM1TXI.

Source file (APG1 ASSEMBLE)

```
         BEGIN NAME=APG1,VERSION=00,AMODE=31,TYPE='EXAMPLE',           -
               XCL=NONE,SHR=NONE
         SPACE 1
         TRANV NAME=APG1,ENTRY=APG1
         TRANV NAME=APG2,ENTRY=APG2
         SPACE 1
*=====================================================================*
*        ALCS - DISPLAY INFORMATION FROM AGENT'S AAA RECORD           *
*=====================================================================*
*                                                                     *
*        ON ENTRY AT APG1                                             *
*        ----------------                                             *
*                                                                     *
*        EBROUT CONTAINS CRI OF ORIGINATING TERMINAL.                 *
*        THE RESPONSE MESSAGE IS SENT TO THIS TERMINAL.               *
*                                                                     *
*        THE PROGRAM RETRIEVES INFORMATION FOR THE CRI IN EBROUT.     *
*                                                                     *
*        ON ENTRY AT APG1                                             *
*        ----------------                                             *
*                                                                     *
*        EBROUT CONTAINS CRI OF ORIGINATING TERMINAL.                 *
*        THE RESPONSE MESSAGE IS SENT TO THIS TERMINAL.               *
*                                                                     *
*        INPUT MESSAGE ON LEVEL D0 IN CM1CM FORMAT.                   *
*        THE PROGRAM RETRIEVES INFORMATION FOR THE CRI IN             *
*        FIELD CM1TXI.                                                *
*---------------------------------------------------------------------*
         EJECT ,
         CO0IC REG=R14             COMIC RETURNED DATA AREA
         CM1CM REG=R01             INPUT/OUTPUT MESSAGE BLOCK
         WA0AA REG=R02             AAA RECORD
         EJECT ,
******** ALASC WORK BLOCK USER DATA DESCRIPTION
         SPACE 1
         ALASC ACTION=EQUATES      GENERATE EQUATES
         SPACE 1
ALASBLOK DSECT ,                   DESCRIBE AUTOMATIC WORK BLOCK
         DC    F'0'                REGISTER R07 SAVED BY ALASC
ALASR00  DC    16F'0'              REGISTER SAVE AREA
         SPACE 1
ALASORN  DC    F'0'                ORDINAL NUMBER
```

```
ALASCRN  DC    CL8' '              CRN
ALASCRI  DC    XL3'0'              CRI
         SPACE 1
ALASSIO  DC    X'0'                SINED IN AGENT INDICATORS
ALASIOT  DC    X'0'                ACTIVE AGENT INDICATORS
         SPACE 1
ALASBITA EQU   X'80'               AGENT A SINED IN OR ACTIVE
ALASBITB EQU   X'40'               AGENT B SINED IN OR ACTIVE
ALASBITC EQU   X'20'               AGENT C SINED IN OR ACTIVE
ALASBITD EQU   X'10'               AGENT D SINED IN OR ACTIVE
ALASBITE EQU   X'08'               AGENT E SINED IN OR ACTIVE
         SPACE 1
ALASCUR  DC    C' '                CURRENT ACTIVE AGENT
*                                  X = AAA NOT INITIALIZED
ALASFAS  DC    5F'0'               FILE ADDRESSES OF SUSPENDED AAAS
         SPACE 1


******** DATA FOR SINED IN AGENTS
         SPACE 1
ALASINF  DC    0F'0'               FIRST DATA AREA
ALASSIN  DC    CL2' '              INITIALS
ALASSIS  DC    CL2' '              DUTY CODE
ALASIND1 DC    X'0'                0 = AGENT NOT SINED IN
*                                  1 = AGENT SINED IN
*                                  2 = ERROR ON FINWC
ALASIND2 DC    X'0'                0 = AGENT NOT ACTIVE
*                                  1 = AGENT ACTIVE
ALASINFL EQU   *-ALASINF           LENGTH OF ONE INFORMATION AREA
         DC    4XL(ALASINFL)'0'    DEFINE FOUR MORE DATA AREAS
         SPACE 1
ALASWORK DC    XL64'0'             WORK AREA
         SPACE 1
         RSECT ,                   RESTORE PROGRAM CSECT NAME
         EJECT ,
*--------------------------------------------------------------------*
*        ENTRY POINT                                                 *
*--------------------------------------------------------------------*
         SPACE 1
APG1     DC    0H'0'
         LA    R01,EBROUT          LOAD ADDRESS OF CRI TO BE QUERIED
         B     APG1000             BRANCH TO COMMON PROCESSING
         SPACE 1
*--------------------------------------------------------------------*
*        ENTRY POINT                                                 *
*--------------------------------------------------------------------*
         SPACE 1
APG2     DC    0H'0'
         LEVTA LEVEL=D0,NOTUSED=APG1E010 BRANCH IF NO BLOCK ATTACHED
         SPACE 1
         L     R01,CE1CR0          LOAD INPUT MESSAGE BASE
         CLC   CM1CCT,=Y(10)       IS THERE A CRI
         BNE   APG1E010            NO - TAKE ERROR BRANCH
         SPACE 1
         TRT   CM1TXI(6),TRTAB1    IS CRI VALID
         BNZ   APG1E010            NO - TAKE ERROR BRANCH
         SPACE 1
         TR    CM1TXI(6),TRTAB2    TRANSLATE CRI TO HEX
         PACK  EBW000(4),CM1TXI(7) CONVERT CRI TO BINARY
         LA    R01,EBW000          LOAD ADDRESS OF CRI TO BE QUERIED
         EJECT ,
*--------------------------------------------------------------------*
*        COMMON PROCESSING                                           *
*--------------------------------------------------------------------*
         SPACE 1
APG1000  DC    0H'0'
         ALASC SIZE=L2             GET AN AUTOMATIC STORAGE BLOCK
         SPACE 1
         USING ALASBLOK,R07        APPLICATION WORK AREA BASE
         SPACE 1
         STM   R00,R15,ALASR00     SAVE CALLER'S REGISTERS
         L     R14,0(,R07)         LOAD CALLER'S R07
```

```
         ST    R14,ALASR00+7*4     SAVE CALLER'S R07
         SPACE 1


******** GET CALLER'S COMMUNICATION TABLE INFORMATION
         SPACE 1
         MVC   ALASCRI,0(R01)      SAVE RESOURCE CRI
         COMIC CRI=ALASCRI,                                            X
               DATA=SYS,                                               X
               AREA=(0,ICELEN)
         BC    B'0100',APG1E010    BRANCH IF CRI IS INVALID
         SPACE 1
         MVC   ALASCRN,ICECRN      SAVE RESOURCE CRN
         MVC   ALASORN,ICEORN      SAVE RESOURCE ORDINAL NUMBER
         SPACE 1
******** CALCULATE AAA FILE ADDRESS
         SPACE 1
         DETAC LEVEL=DF            DETACH CONTENTS OF LEVEL DF
         SPACE 1
         L     R00,ALASORN         LOAD ORDINAL NUMBER
         LA    R06,#WAARI          LOAD RECORD TYPE NUMBER
         LA    R07,CE1FAF          LOAD DATA LEVEL ADDRESS FOR RESULT
         ENTRC FACE                COMPUTE RECORD FILE ADDRESS
         SPACE 1
         L     R07,CE1AUT          LOAD ALASC BLOCK ADDRESS
         LA    R07,8(,R07)         LOAD APPLICATION WORK AREA BASE
         LTR   R00,R00             IS FILE ADDRESS VALID
         BZ    APG1E020            YES - TAKE ERROR BRANCH
         SPACE 1
******** FIND THE AGENT'S AAA RECORD
         SPACE 1
         XC    CE1FAF(4),CE1FAF    CLEAR ID/RCC FOR FIND
         FINWC DF,APG1E030         FIND AAA RECORD
         EJECT ,
******** RETRIEVE INFORMATION FROM AAA RECORD
         SPACE 1
         L     R02,CE1CRF          LOAD AAA RECORD BASE
         MVI   ALASCUR,C'X'        FLAG RECORD NOT INITIALIZED
         CLC   WA0BID,APG1ID       IS RECORD INITIALIZED
         BNE   APG1090             NO - BR TO BUILD RESPONSE MESSAGE
         SPACE 1
         MVI   ALASCUR,C' '        FLAG NO ACTIVE AGENT
         MVC   ALASFAS(5*4),WA0AGA SAVE SUSPENDED AAA FILE ADDRESSES
         MVC   ALASSIO,WA0SIO      SAVE SINED IN AGENT INDICATOR
         MVC   ALASIOT,WA0IOT      SAVE ACTIVE AGENT INDICATOR
         SPACE 1
******** CHECK SINED IN AGENTS AND IDENTIFY ACTIVE AGENT
         SPACE 1
         LA    R03,ALASINF         POINT TO DATA AREA FOR AGENT A
         L     R04,=A(ALASBITA)    LOAD BIT MASK TO TEST FOR AGENT A
         LA    R05,APG1LIST        POINT TO LIST OF AGENT AREAS
         LA    R00,5               LOAD NUMBER OF AGENT AREA LETTERS
         SPACE 1
APG1010  DC    0H'0'
         TM    ALASSIO,0           (EXECUTED INSTRUCTION)
         EX    R04,*-4             IS AGENT SINED IN
         BZ    APG1015             NO - BRANCH
         SPACE 1
         MVI   ALASIND1-ALASINF(R03),1 INDICATE AGENT IS SINED IN
         SPACE 1
APG1015  DC    0H'0'
         TM    ALASIOT,0           (EXECUTED INSTRUCTION)
         EX    R04,*-4             IS AGENT ACTIVE
         BZ    APG1020             NO - BRANCH
         SPACE 1


         MVC   ALASCUR,0(R05)      SAVE ACTIVE AGENT AREA LETTER
         MVI   ALASIND2-ALASINF(R03),1 INDICATE AGENT IS ACTIVE
         MVC   ALASSIN-ALASINF(2,R03),WA0SIN MOVE IN AGENT INITIALS
         MVC   ALASSIS-ALASINF(2,R03),WA0SIS MOVE IN AGENT DUTY CODE
         SPACE 1
```

```
APG1020  DC    0H'0'
         LA    R03,ALASINFL(,R03)  POINT TO DATA AREA FOR NEXT AGENT
         SRL   R04,1               ADJUST INDICATOR FOR NEXT AGENT
         LA    R05,1(,R05)         POINT TO NEXT AGENT AREA LETTER
         BCT   R00,APG1010         CHECK ALL INDICATORS
         EJECT ,
******** RETRIEVE INFORMATION FROM ANY SUSPENDED AAA AREAS
         SPACE 1
         RELCC LEVEL=DF            RELEASE STORAGE BLOCK
         SPACE 1
         LA    R03,ALASINF         POINT TO DATA AREA FOR AGENT A
         LA    R04,ALASFAS         POINT TO SUSPENDED AAA FILE ADDRS
         LA    R05,5               LOAD NUMBER OF AGENT AREAS
         SPACE 1
APG1030  DC    0H'0'
         ICM   R00,B'1111',0(R04)  IS THERE ANY FILE ADDRESS
         BZ    APG1050             NO - BRANCH
         SPACE 1
******** FIND THE POOL RECORD
         SPACE 1
         ST    R00,CE1FMF          MOVE IN FILE ADDRESS FOR FIND
         MVC   CE1FAF(4),APG1ID    MOVE IN ID/CTL FOR FIND
         FINWC DF,APG1040          FIND AAA RECORD
         SPACE 1
******** RETRIEVE INFORMATION FROM POOL RECORD
         SPACE 1
         L     R02,CE1CRF          LOAD POOL RECORD BASE
         SPACE 1
         MVC   ALASSIN-ALASINF(2,R03),WA0SIN MOVE IN AGENT INITIALS
         MVC   ALASSIS-ALASINF(2,R03),WA0SIS MOVE IN AGENT DUTY CODE
         SPACE 1
         RELCC LEVEL=DF            RELEASE STORAGE BLOCK
         B     APG1050
         SPACE 1
APG1040  DC    0H'0'
         MVI   ALASIND1-ALASINF(R03),2 INDICATE ERROR ON FINWC
         RELCC LEVEL=DF,TYPE=COND  RELEASE STORAGE BLOCK
         SPACE 1
APG1050  DC    0H'0'
         LA    R03,ALASINFL(,R03)  POINT TO DATA AREA FOR NEXT AGENT
         LA    R04,4(,R04)         POINT TO NEXT FILE ADDRESS
         BCT   R05,APG1030         CHECK ALL FILE ADDRESSES
         EJECT ,
*-------------------------------------------------------------------*
*        BUILD RESPONSE MESSAGE                                     *
*-------------------------------------------------------------------*
         SPACE 1
APG1090  DC    0H'0'
         CRUSA S0=F                RELEASE ANY STORAGE BLOCK
         GETCC LEVEL=DF,SIZE=L3    GET A NEW STORAGE BLOCK
         LR    R01,R14             LOAD OUTPUT MESSAGE BASE
         SPACE 1
         CLI   ALASCUR,C'X'        IS AAA RECORD INITIALIZED
         BNE   APG1100             YES - BRANCH
         SPACE 1


         MVC   CM1CCT,=Y(APG1MUL+5)  MOVE IN MESSAGE CCT
         MVC   CM1TXT(APG1MUL),APG1MU MOVE IN MESSAGE TEXT
         MVC   CM1TXT+(APG1MUN-APG1MU)(8),ALASCRN MOVE IN CRN
         SPACE 1
         B     APG1900             BRANCH TO COMPLETE MESSAGE
         SPACE 1
APG1100  DC    0H'0'
         LA    R04,APG1MI          LOAD MOVE-FROM ADDRESS
         LA    R05,APG1MIL         LOAD MOVE-FROM LENGTH
         LA    R02,CM1TXT          LOAD MOVE-TO ADDRESS
         LR    R03,R05             LOAD MOVE-TO LENGTH
         MVCL  R02,R04             MOVE IN MESSAGE TEXT
         SPACE 1
         MVC   CM1CCT,=Y(APG1MIL+5)  MOVE IN MESSAGE CCT
         MVC   CM1TXT+(APG1MIN-APG1MI)(8),ALASCRN MOVE IN CRN
```

```
         SPACE 1
         MVI   CM1TXT+(APG1MIA-APG1MI),C'A'
         MVI   CM1TXT+(APG1MIB-APG1MI),C'B'
         MVI   CM1TXT+(APG1MIC-APG1MI),C'C'
         MVI   CM1TXT+(APG1MID-APG1MI),C'D'
         MVI   CM1TXT+(APG1MIE-APG1MI),C'E'
         EJECT ,
         LA    R03,ALASINF         POINT TO DATA AREA FOR AGENT A
         LA    R04,CM1TXT+(APG1MIA-APG1MI)  POINT TO OUTPUT AREA
         LA    R00,5               LOAD NUMBER OF AGENT AREAS
         SPACE 1
APG1110  DC    0H'0'
         MVC   APG1MISI-APG1MIA(3,R04),APG1MYES
         CLI   ALASIND1-ALASINF(R03),1  IS AGENT SINED IN
         BE    APG1120                  YES - BRANCH
         SPACE 1
         MVC   APG1MISI-APG1MIA(3,R04),APG1MNO
         CLI   ALASIND1-ALASINF(R03),2  ERROR RETURN FROM FINWC
         BNE   APG1140                  NO - BRANCH
         SPACE 1
         MVC   1(APG1MIAL-1,R04),APG1MERR MOVE IN FINWC ERROR MESSAGE
         B     APG1140                  BRANCH FOR NEXT AGENT
         SPACE 1
APG1120  DC    0H'0'
         MVC   APG1MIAC-APG1MIA(3,R04),APG1MNO
         CLI   ALASIND2-ALASINF(R03),1  IS AGENT ACTIVE
         BNE   APG1130                  NO - BRANCH
         SPACE 1
         MVC   APG1MIAC-APG1MIA(3,R04),APG1MYES
         SPACE 1
APG1130  DC    0H'0'
         MVC   APG1MIIN-APG1MIA(2,R04),ALASSIN-ALASINF(R03) INITIALS
         MVC   APG1MIDU-APG1MIA(2,R04),ALASSIS-ALASINF(R03) DUTY CODE
         SPACE 1
APG1140  DC    0H'0'
         LA    R03,ALASINFL(,R03)  POINT TO DATA AREA FOR NEXT AGENT
         LA    R04,APG1MIAL+1(,R04)  POINT TO NEXT OUTPUT AREA
         BCT   R00,APG1110         GET DATA FOR ALL AGENT DATA AREAS
         EJECT ,


******** COMPLETE MESSAGE AND SEND IT
         SPACE 1
APG1900  DC    0H'0'
         TIMEC DISPLAY,            MOVE IN TIME STAMP                X
               TIME,                                                 X
               AREA=CM1TXT
         SPACE 1
         COMIC CRI=EBROUT,                                           X
               DATA=SYS,                                             X
               AREA=(0,ICELEN)
         BC    B'0100',APG1E050    BRANCH IF CRI IS INVALID
         SPACE 1
         MVC   CM1CRI,ICECRI+1     MOVE IN DESTINATION ADDRESS
         SPACE 1
         TM    ICESTPP,L'ICESTPP   IS IT A PRINTER
         BO    APG1910             YES - BRANCH
         SPACE 1
         MVC   CM1CMW(2),=AL1(#WEW,#CAR) CONTROL CHARS FOR DISPLAY
         SENDC DF,A                SEND RESPONSE MESSAGE
         SPACE 1
         B     APG1920
         SPACE 1
APG1910  DC    0H'0'
         MVC   CM1CMW(2),=AL1(#NOP,#CAR) CONTROL CHARS FOR PRINTER
         SENDC DF,L                SEND RESPONSE MESSAGE
         SPACE 1
******** RETURN TO CALLING PROGRAM, OR EXIT
         SPACE 1
APG1920  DC    0H'0'
         ATTAC LEVEL=DF            ATTACH CONTENTS OF LEVEL DF
         SPACE 1
```

```
         LM    R00,R14,ALASR00     RESTORE CALLER'S REGISTERS
         SPACE 1
         BACKC ,                   RETURN TO CALLING PROGRAM, OR EXIT
         EJECT ,
*------------------------------------------------------------------*
*        ERROR ROUTINES                                            *
*------------------------------------------------------------------*
         SPACE 1
******** INVALID CRI
         SPACE 1
APG1E010 DC    0H'0'
         LA    R00,APG1M01
         SERRC E,A00010
         SPACE 1
******** ERROR RETURN FROM FACE
         SPACE 1
APG1E020 DC    0H'0'
         LA    R00,APG1M02
         SERRC E,A00020
         SPACE 1
******** ERROR RETURN FROM FINWC
         SPACE 1
APG1E030 DC    0H'0'
         LA    R00,APG1M03
         SERRC E,A00030
         SPACE 1
******** ERROR RETURN FROM FINWC
         SPACE 1
APG1E040 DC    0H'0'
         LA    R00,APG1M04
         SERRC E,A00040
         SPACE 1


******** INVALID CRI IN EBROUT
         SPACE 1
APG1E050 DC    0H'0'
         LA    R00,APG1M05
         SERRC E,A00050
         EJECT ,
*------------------------------------------------------------------*
*        PROGRAM CONSTANTS                                         *
*------------------------------------------------------------------*
         SPACE 1
******** TABLE TO VALIDATE CHARACTER HEX DATA
         SPACE 1
TRTAB1   DC    256X'04'
         ORG   TRTAB1+C'0'
         DC    10X'00'
         ORG   TRTAB1+C'A'
         DC    6X'00'
         ORG   ,
         SPACE 1
******** TABLE TO TRANSLATE FROM CHARACTER HEX TO HEX
         SPACE 1
TRTAB2   EQU   *-C'A'
         DC    XL(256-C'A')'00'
         ORG   TRTAB2+C'A'
         DC    X'0A0B0C0D0E0F'
         ORG   TRTAB2+C'0'
         DC    X'00010203040506070809'
         ORG   ,
         SPACE 1
APG1ID   DC    C'AA',X'0000'       AAA RECORD ID/CTL
APG1LIST DC    C'ABCDE'            AGENT AREA LETTERS
         SPACE 1
******** RESPONSE MESSAGE -- AAA RECORD NOT INITIALIZED
         SPACE 1
APG1MU   DC    C'hh.mm.ss '
         DC    C'Information for agent CRN-'
APG1MUN  DC    CL8' '
         DC    AL1(#CAR),C' ',AL1(#CAR)
```

```
         DC    C'AAA record not initialized'
         DC    AL1(#CAR,#SOM,#EOM)
APG1MUL  EQU   *-APG1MU
         EJECT ,
******** RESPONSE MESSAGE
         SPACE 1
APG1MI   DC    C'hh.mm.ss '
         DC    C'Information for Agent CRN-'
APG1MIN  DC    CL8' '
         DC    AL1(#CAR),C' ',AL1(#CAR)
         DC    C'Area  Sined-in  Active  Initials  Duty code'
         DC    AL1(#CAR)
         DC    C'----------------------------------------'
         DC    AL1(#CAR)
APG1MIA  DC    CL1' ',CL5' '
APG1MISI DC    CL3' ',CL7' '
APG1MIAC DC    CL3' ',CL5' '
APG1MIIN DC    CL2' ',CL8' '
APG1MIDU DC    CL2' ',CL7' '
APG1MIAL EQU   *-APG1MIA


         DC    AL1(#CAR)
APG1MIB  DC    CL(APG1MIAL)' ',AL1(#CAR)
APG1MIC  DC    CL(APG1MIAL)' ',AL1(#CAR)
APG1MID  DC    CL(APG1MIAL)' ',AL1(#CAR)
APG1MIE  DC    CL(APG1MIAL)' ',AL1(#CAR)
         DC    AL1(#SOM,#EOM)
APG1MIL  EQU   *-APG1MI
         SPACE 1
APG1MERR DC    C' *** Error retrieving AAA pool record *** '
APG1MYES DC    C'yes'
APG1MNO  DC    C'no '
         EJECT ,
******** SYSTEM ERROR MESSAGES
         SPACE 1
APG1M01  DC    AL1(L'APG1M01T)
APG1M01T DC    C'INVALID OR MISSING CRI'
         SPACE 1
APG1M02  DC    AL1(L'APG1M02T)
APG1M02T DC    C'ERROR RETURN FROM FACE'
         SPACE 1
APG1M03  DC    AL1(L'APG1M03T)
APG1M03T DC    C'FINWC ERROR FOR AAA FIXED FILE'
         SPACE 1
APG1M04  DC    AL1(L'APG1M04T)
APG1M04T DC    C'FINWC ERROR FOR AAA POOL FILE'
         SPACE 1
APG1M05  DC    AL1(L'APG1M05T)
APG1M05T DC    C'INVALID CRI IN EBROUT'
         EJECT ,
*-------------------------------------------------------------------*
*        LITERAL POOL                                               *
*-------------------------------------------------------------------*
         SPACE 1
         LTORG ,
         EJECT ,
         FINIS ,
         SPACE 1
         END   ,
```

Appendix B. Sample application: C language

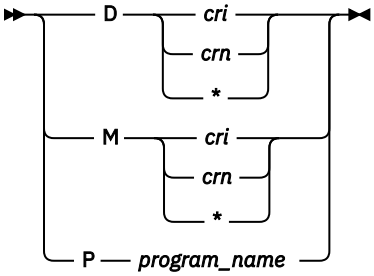
This sample application demonstrates use of the C applications programming interface (API). It consists of 9 files of source code:

DXCBINQ (Entry points definition file)
DXCBINQH (C source code)
DXCBINQ0 (C source code)
DXCBINQ1 (C source code)
DXCBINQ2 (C source code)
DXCBINQ3 (assembler)
DXCBINQD (C source code)
DXCBINQM (C source code)
DXCBINQP (C source code)

Two of these files are written in assembler, the other seven are written in C. Each file is described individually later on.

Purpose of the application

The application allows a user to inquire about any printer or display terminal, or about the storage address of an application program. The user does this by means of a command D, M, or P with the following syntax. (Syntax diagrams are explained in the *ALCS Operation and Maintenance*.)



Where:

D
Display information about a specified terminal.

M
Monitor a specified terminal.

P
Display the storage address of an application program.

cri
The communication resource identifier (CRI) of the terminal (6 hexadecimal digits).

crn
The communication resource name (CRN) of the terminal (1 to 8 alphanumeric characters).

***
This indicates the user's own terminal.

program_name
The 4-character name of an application program.

© Copyright IBM Corp. 2003, 2019

Sample input

The following are all valid inquires:

DAT14
Display information about terminal with a CRI of 'AT14'

DPYCS109M
Display information about terminal with a CRN of 'PYCS109M'

D*
Display information about this terminal.

MPRC
Monitor the terminal that has a CRI of 'PRC'

M*
Monitor this terminal.

PSAMP
Display the address of program SAMP.

Example output is shown in the next section.

Sample output

The following screens show examples of the use of D, M, and P commands issued from a terminal routed to the application TST1. See “Running the application” on page 198. (The first screen shows the result of keying an invalid input message (X).)

```
 X
 DXCBINQ0 started: ecb=02F794A0
 valid options are:

 Dtermid  - display a terminal
 Mtermid  - monitor a terminal
 Pprog    - display the address of a program
            termid is CRI or CRN
 DXCBINQ0 ended


D
DXCBINQ0 started: ecb=02E494A0
terminal ID not specified

 display terminal status

  parameters : cri
    or         crn

  the CRN, CRI and logged on status of the terminal are displayed
DXCBINQ0 ended


M
DXCBINQ0 started: ecb=02EC94A0
terminal ID not specified

 monitor terminal status

  parameters : cri
    or         crn

  the CRN, CRI and logged on status of the terminal are displayed
DXCBINQ0 ended


P
DXCBINQ0 started: ecb=02EE14A0
no program name specified

Usage: display the address of an ALCS program

  parameters : prog

  where prog is the 4 character program name
DXCBINQ0 ended


DAT14
DXCBINQ0 started: ecb=02F194A0
CRN=PYCS109O CRI=02008A Status=active
DXCBINQ0 ended


DPYCS109M
DXCBINQ0 started: ecb=02F494A0
CRN=PYCS109M CRI=020088 Status=active
DXCBINQ0 ended


MPRC
DXCBINQ0 started: ecb=02E114A0
started monitor: CRN=PYCS109M CRI=020088 Status=active
DXCBINQ0 ended


M*
DXCBINQ0 started: ecb=02ED94A0
started monitor: CRN=PYCS109O CRI=02008A Status=active
DXCBINQ0 ended
```

If the status of the current terminal changes, messages such as the following appear:

```
18.30.30 SNDU  UNSOLICITED MESSAGE FROM CRN-STVPROC
               CREATION DATE 19/09/89 TIME 18.30.30
SAMP: TERMINAL PYCS109M/020088 NOW INACTIVE

18.34.31 SNDU  UNSOLICITED MESSAGE FROM CRN-STVPROC
               CREATION DATE 19/09/89 TIME 18.34.31
SAMP: TERMINAL PYCS109M/020088 NOW ACTIVE


PSAM0
DXCBINQ0 started: ecb=02F894A0
program SAM0 is at address 7F268018
DXCBINQ0 ended
```

If the requested program is not found, TST1 responds as follows:

```
PXXXX
DXCBINQ0 started: ecb=02E294A0
program 'XXXX' not found
DXCBINQ0 ended
```

How the program source files are related

Some of the program source files call functions defined in the other program source files. For example, the routine in DXCBINQ0 calls functions defined in DXCBINQD, or DXCBINQM, or DXCBINQP, or none, depending on the contents of the input message.

The relationship between the program source files is shown in Figure 114 on page 186. Program source files that contain functions that are called from another file are shown below the file that calls them.

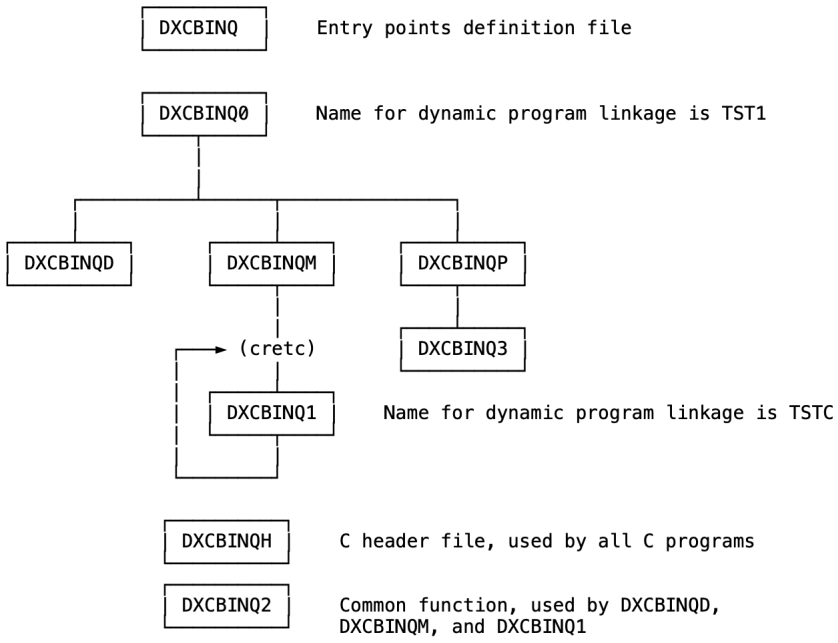


Figure 114. Relationship between files in the sample application

See the Chapter 7, “Application program management,” on page 124 for details of how to assemble and link-edit the programs. The individual files are described next.

DXCBINQ

This is the entry points definition file. You process this using the offline program DXCBCLPP, as described in “Program header build - DXCBCLPP program” on page 168. You create a load module as described in “Building the HLL load module” on page 168. In this example the load module has the same name as the entry points definition file, DXCBINQ.

The purpose of this file is to define to ALCS two 4-character program names (TST1 and TSTC) and to associate each of these names with a C language function (DXCBINQ0 and DXCBINQC respectively).

```
*=============================================================*
*       DXCBINQ -- SAMPLE C APPLICATION FOR ALCS V2            *
*=============================================================*
* Purpose:                                                    *
*         Sample application to demonstrate ALCS V2 C support *
*                                                             *
*         Entry points definition file used as input to the   *
*         link-edit procedure to produce the DXCBINQ load     *
*         module                                              *
*                                                             *
*=============================================================*
 BEGIN NAME=TST1
*sample command processor routine entry point
 TRANV NAME=TST1,ENTRY=DXCBINQ0,
       MODULE=(DXCBINQ0,DXCBINQ2,DXCBINQ3,DXCBINQD,
               DXCBINQM,DXCBINQP)
*entry point for program activated by CRETC
 TRANV NAME=TSTC,ENTRY=DXCBINQC,MODULE=DXCBINQ1
```

DXCBINQH

This file is a header file which is `#included` in each of the C programs in this sample application. Amongst other things it does the following:

- `#includes` all the files required to use the ALCS (and other) functions used in the application.
- `#defines` the function `getcri` and maps this to the function defined in the file DXCBINQ2.
- `#defines` the `showcri` and `showcrn` macros which allow easy use of the `comic` function.
- `#defines` a `valid_program_name` macro, used in DXCBINQP to test the program name.
- `#defines` a `rout_umsg` macro, used in DXCBINQ1 to send an unsolicited message to a terminal. (The CRI of the terminal is supplied by the calling function.)
- `#defines` various other general purpose symbols and a `monitor_data` structure.

```
/*
*=====================================================================*
*       DXCBINQH -- SAMPLE C APPLICATION FOR ALCS V2                  *
*=====================================================================*
* Purpose:                                                            *
*         Sample application to demonstrate ALCS V2 C support         *
*                                                                     *
*         Header file containing definitions used in the sample       *
*         C programs                                                  *
*                                                                     *
*=====================================================================*
*/


#pragma strings (readonly)
#include <tpfeq.h>
#include <tpfapi.h>
#include <tpfio.h>
#include <string.h>
#include <ctype.h>
```

```
#include <c$co0ic.h>

/*  prototype definitions for sample application functions */

void dxcbinqd(char *);
void dxcbinqm(char *);
void dxcbinqp(char *);
void dxcbinqt(void);

/*  definitions internal functions */

int getcri(char *);               /* sample C function         */
#pragma map(getcri,"DXCBINQ2")
#define GETCRI_ERROR 0xffffffff

#pragma map(find_prog,"DXCBINQ3")  /* sample assembler function */
#pragma linkage(find_prog,OS)
void * find_prog(char * name);

/* define macro to get co0ic info pointer for a CRI  */

#define showcri(cri)\
     comic(COMIC_CRI, COMIC_SYS, ((char *)(CRI))+1, NULL, 0);

/* define macro to get co0ic info pointer for a CRN  */

#define showcrn(crn)\
     comic(COMIC_CRN, COMIC_SYS, crn , NULL, 0);

/* define macro to test for a valid ALCS program name     */

#define valid_program_name(prog) \
  ( isalpha(prog[0]) &1 isalnum(prog[1]) &1\
    isalnum(prog[2]) &1 isalnum(prog[3])      )



/* define macro to set up RCPL and routc() a unsolicited message */
/* parameters are: pointer to rcpl (struct rc0pl *),             */
/*                 level containing the message (D0 .. DF),      */
/*                 cri to send message to (int)                  */

#define rout_umsg(rcpl,level,cri,ori) \
{\
  memset(rcpl,0,sizeof(ecbptr()->ce1rcpl));  /* clear RCPL to zeros */\
  memcpy(RCPL->rcplorg,((char *)(&ori))+1,3);/* insert origin     */\
  memcpy(RCPL->rcpldes,((char *)(CRI))+1,3); /* insert destination*/\
  rptr->rcplctl0 |= RCPL0MTY;                 /* set unsolicited bit*/\
  routc(rcpl,level);                          /* send the message   */\
}


/*  definitions for logic values  */

#define TRUE  1
#define FALSE 0

/*  definitions for monitor       */

#define MONITOR_PROG  "TSTC"     /* name of monitor program for cretc */
#define MONITOR_TIME  1          /* monitor at   1 minute interval    */
#define MONITOR_ID    "ROC     " /* monitor messages to RO CRAS       */
#define ROUT_LEVEL    D2         /* ecb level used for routc()        */

struct monitor_data
{
  unsigned status :  1;
  unsigned unused :  7;
  unsigned cri    : 24;
};
```

DXCBINQ0

This is the main entry point of the sample application, activated when ALCS enters TST1. This program calls one of three functions: `sampled`, `samplem`, or `samplep`, depending on the first character of the message (the primary action code) being 'D', 'M', or 'P'. When the first character is not one of these, the function displays a help message.

```
/*
*====================================================================*
*       DXCBINQ0 -- SAMPLE C APPLICATION FOR ALCS V2                 *
*====================================================================*
* Purpose:                                                           *
*         Sample application to demonstrate ALCS V2 C support        *
*                                                                    *
*         This is an input message editor program and selects the    *
*         required function by the first character (primary action   *
*         code) of the message.                                      *
*                                                                    *
*====================================================================*
*/

#include "dxcbinqh.h"

static char usage[] = {"valid options are:\n\
\nDtermid  - display a terminal\
\nMtermid  - monitor a terminal\
\nPprog    - display the address of a program\
\n           termid is CRI or CRN \
 "};


void DXCBINQ0()
{
static char buf[180];
char *p;

      gets(buf);
      p=buf;

      /* if zero length message - clear the screen     */
      /* using the ALCS screen formatting program CFMT */

      if (*p == 0 ) entdc("CFMT",NULL);  /* does not return */

      printf("%s\nDXCBINQ0 started: ecb=%08X\n",
              buf,
              ecbptr() );

      /* select on primary action code */

      switch(*p++)
      {
        case 'D': dxcbinqd(p);
                  break;

        case 'M': dxcbinqm(p);
                  break;

        case 'P': dxcbinqp(p);
                  break;

        default : puts(usage);
      }

      puts("DXCBINQ0 ended");

      exit(0);
}
```

DXCBINQ1

This program is called by dynamic linkage (rather than static linkage). It has a 4-character name known to ALCS. The function in DXCBINQM calls this program by using the CRETC function, specifying a delay of MONITOR_TIME (#defined in DXCBINQH as 1 minute).

This program produces a display when the status of the terminal has changed.

```
/*
*====================================================================*
*       DXCBINQ1 -- SAMPLE C APPLICATION FOR ALCS V2                 *
*====================================================================*
* Purpose:                                                           *
*         Sample application to demonstrate ALCS V2 C support        *
*                                                                    *
*         This program is activated via a cretc().                   *
*                                                                    *
*         4 Bytes of data are passed in the ECB at ebw000            *
*                                                                    *
*         ebw000    : the last noted status of the terminal          *
*         ebw001/2/3: the CRI of the terminal                        *
*                                                                    *
*         If the status of the terminal has changed an               *
*         message is sent.                                           *
*                                                                    *
*====================================================================*
*/


#include "dxcbinqh.h"

void dxcbinqc()
{
  int                   cri,length,origin;
  int                   monitor_cri;
  struct    monitor_data *mptr;
  struct    co0ic        *cptr;
  struct    rc0pl        *rptr;
  struct    cm1cm        *cmptr;
  #define   CM1TXT       cmptr->cm1txt
  #define   CM1CCT       cmptr->cm1cct
  char                   serr_msg[60];

  /* access the 4 bytes passed in ebw000 of created ecb */

  mptr = (struct monitor_data *) ecbptr()->ebw000;

  cri  = mptr->cri;
  cptr = showcri(cri);

  if ( (int)cptr < 0)
  {
    serrc_op(SERRC_EXIT, 0xFFFF01, "monitoring invalid CRI", NULL);
  }


  if (cptr->icescst != mptr->status)
  {
    /* status has changed send a message  using routc() */

    mptr->status = cptr->icescst;     /* update status of terminal */

    /* build the message in a storage block             */

    cmptr = getcc(ROUT_LEVEL,GETCC_TYPE,L1); /* get block for message */
    sprintf(CM1TXT,"DXCBINQ1: TERMINAL %8.8s/%06X NOW %s",
            cptr->icecrn, cptr->icecri,
            (cptr->icescst)?"ACTIVE":"INACTIVE");

    length = strlen(CM1TXT);
```

```
    CM1TXT[length++] = EOM;               /* add EOM */
    CM1CCT = length + 5;                  /* set count in block */

    /* get the cri of the terminal specified as MONITOR_ID   */

    monitor_cri = getcri(MONITOR_ID);
    if (monitor_cri == GETCRI_ERROR)
    {
      sprintf(serr_msg,"invalid MONITOR_ID '%8.8s'", MONITOR_ID);
      serrc_op(SERRC_EXIT, 0xFFFF01, serr_msg, NULL);
    }

    /* get the originating terminal address from ebrout       */
    origin = ecbptr()->ebrout;

    /* set up the RPL for an unsolicted message to MONITOR_ID */

    rptr = (struct rc0pl *) ecbptr()->ce1rcpl;  /* use ecb's rcpl */
    rout_umsg(rptr,ROUT_LEVEL,monitor_cri,origin);

  }

  /* schedule next monitor check */

  cretc(CRETC_MINUTES,MONITOR_PROG,MONITOR_TIME,mptr);

  exit(0);

}
```

DXCBINQ2

This function returns an integer containing a CRI. The input is either a CRI, CRN, or an asterisk (*) character (which represents the user's own terminal). This function is documented here in a similar way to the ALCS functions described in the *ALCS Application Programming Reference - C Language*.

Format

```
#include "sampleh.h"

int getcri(char *termid)
```

Where:

termid
A pointer to either a CRN (1 to 8 alphanumeric characters) or a CRI (6 hexadecimal digits).

Description

This function returns the CRI of terminal when it is supplied with a terminal name. The supplied terminal name can be either a CRN or a CRI. The function tests if the terminal exists by using the macro `showcrn` or `showcrn` (`#defined` in DXCBINQH) which calls the `comic` function. (See *ALCS Application Programming Reference - C Language* for a description of this function..)

Note: As both a CRN and CRI could start with the characters 'A' through 'F', the function tests if the text is a valid CRN before converting it from hexadecimal to binary (and then testing if this CRI exists).

Normal return

An `int` field containing the CRI of the terminal (in binary).

Error return

If the terminal does not exist, the function returns a value of GETCRI_ERROR (#defined in DXCBINQH).

Loss of control

This function does not cause the entry to lose control.

Example

This section of code (extracted from SAMPLD) uses a pointer to a terminal name in **ptr* and outputs the current status of the terminal, if it exists. If the terminal name does not refer to a valid terminal, it outputs an error message.

```
⋮
    if( (cri = getcri(ptr)) == GETCRI_ERROR)
    {
      PUTERR("terminal ID not a valid CRI/CRN ");
    }
    else
    {
      /* display the status of the terminal */

      cptr = showcri(cri);

      printf("CRN=%8.8s CRI=%06X Status=%s\n",
             cptr->icecrn,
             cptr->icecri,
             (cptr->icescst)?"active":"inactive");
    }
  }
```

Related information

This function uses the macros showcri, and showcrn, #defined in DXCBINQH. It also uses the ALCS functions ecbptr and, indirectly through the macros showcrn and showcrn, the comic function. (See *ALCS Application Programming Reference - C Language* for a description of these functions.)

```
/*
*=====================================================================*
*       DXCBINQ2 -- SAMPLE C APPLICATION FOR ALCS V2                  *
*=====================================================================*
* Usage:                                                              *
*       int getcri(char * termid);                                    *
*                                                                     *
* Purpose:                                                            *
*         the parameter is a pointer to text representing a CRI       *
*         or a CRN. The numeric value of the CRI is returned          *
*         or GETCRI_ERROR if the text is not a valid CRI or CRN.      *
*                                                                     *
*=====================================================================*
*/

#include "dxcbinqh.h"

int getcri( char * termid)
{
  int cri = GETCRI_ERROR;
  int work;
  struct co0ic * cptr = (void *) 0xffffffff;
  char buf[20];

  #ifdef DEBUG
    printf("DXCBINQ2/getcri started: parms are:%s>>\n",termid);
  #endif
```

```
  if (*termid == '*')
  {
    work = ecbptr()->ebrout;        /* use CRI from ecb   */
    cptr = showcri(work);           /* get info for CRI   */

    #ifdef DEBUG
      printf("DXCBINQ2/getcri * showcri(%06X)",work ,cptr);
    #endif
  }

  if (isalpha(*termid))
  {
    /* valid CRN must start with alphabetic character   */

    /* move CRN to buffer and pad with blanks */
    memcpy(buf,termid,8);
    strcat(buf,"        ");
    cptr = showcrn(buf);            /* get info for CRN */

    #ifdef DEBUG
      printf("DXCBINQ2/getcri showcrn(%8.8s):%x\n",buf ,cptr);
    #endif
  }


  if ( (! ((int)cptr > 0)) && isxdigit(*termid)  )
  {
    /* if termid is not a valid CRN and starts with a hex digit */

    work = -1;
    sscanf(termid,"%x",&work);    /* convert hex to binary */
  #ifdef DEBUG
    printf("DXCBINQ2/getcri comic CRI %x\n",work);
  #endif

    if (work > 0)                 /* disallow CRI 000000  */
    {
      cptr = showcri(work);       /* get info for CRI     */
    }

    #ifdef DEBUG
      printf("DXCBINQ2/getcri CRI showcri(%06x):%x\n",work,cptr);
    #endif
  }
  if ((int)cptr >  0)
  {
    /* found a valid terminal */

    cri = cptr->icecri;

  }

  #ifdef DEBUG
    printf("DXCBINQ2/getcri ended: %x\n",cri);
  #endif

  return(cri);
}
```

DXCBINQ3

This is an assembler routine which uses an ALCS assembler macro (FINPC) to find the storage address of a program. The routine is mapped to a C function by a `#pragma map` in the application header file DXCBINQH.

```
*====================================================================*
*       DXCBINQ3 -- SAMPLE C APPLICATION FOR ALCS V2                 *
*====================================================================*
```

```
* Usage:                                                               *
*       #pragma map(find_prog,"DXCBINQ3")                             *
*       #pragma linkage(find_prog,OS)                                 *
*       void * find_prog(char * name);                                *
*                                                                     *
* Purpose:                                                            *
*         this is an example of how a C function can be implenmented  *
*         in assembler. This routine uses the FINPC macro to get      *
*         address of an ALCS program. If the program does not exist   *
*         a NULL value is returned.                                   *
*                                                                     *
*---------------------------------------------------------------------*
*        PARAMETER    USAGE                                           *
*        ---------    ------------------------------------------      *
*        1   0(R06)   POINTER TO NAME OF PROGRAM                      *
*                                                                     *
*        MUST BE ASSEMBLED WITH 'SYSPARM(DXCLIBC)'                    *
*                                                                     *
*=====================================================================*
         BEGIN NAME=DXCBINQ3
         ICPLOG ,
         L     R01,0(R06)          GET POINTER TO PROGRAM NAME
         SPACE 1
         FINPC PROGRAM=(R01)       FIND THE PROGRAM
         SPACE 1
         LR    R06,R14             RETURN ADDRESS OR NULL
         ICELOG ,                  RETURN TO CALLER
         LTORG ,
         SPACE 1
         FINIS ,
         END   ,
```

DXCBINQD

This file contains a function, called by DXCBINQ0 that displays information about a terminal. The information displayed consists of the CRI, the CRN, and the terminal status.

```
/*
*=====================================================================*
*       DXCBINQD -- SAMPLE C APPLICATION FOR ALCS V2                  *
*=====================================================================*
* Purpose:                                                            *
*         Sample application to demonstrate ALCS V2 C support         *
*                                                                     *
*=====================================================================*
*/

#include "dxcbinqh.h"

/* define macro for error conditions             */

#define PUTERR(msg)  puts(msg); error = TRUE;

static char usage[] = {"\
\n display terminal status\
\n\
\n  parameters : cri\
\n    or         crn\
\n\
\n  the CRN, CRI and logged on status of the terminal are displayed\
"
};


void dxcbinqd( char *ptr)
{
  int cri = 0;
  char error = FALSE;
  struct co0ic *cptr;
```

```
  #ifdef DEBUG
    printf("DXCBINQD started: parms are:%s\n",ptr);
  #endif


  if( *ptr == 0)
  {
    PUTERR("terminal ID not specified");
  }
  else
  {
    if( (cri = getcri(ptr)) == GETCRI_ERROR)
    {
      PUTERR("terminal ID not a valid CRI/CRN ");
    }
    else
    {
      /* display the status of the terminal */

      cptr = showcri(cri);

      printf("CRN=%8.8s CRI=%06X Status=%s\n",
             cptr->icecrn,
             cptr->icecri,
             (cptr->icescst)?"active":"inactive");
    }
  }

  if (error)
  {
    puts(usage);
  }

  #ifdef DEBUG
    puts("DXCBINQD ended");

  #endif
  return;

}
```

DXCBINQM

This file contains a function `samplem`, which monitors a terminal. It is called from DXCBINQ0. The information displayed consists of the CRI, the CRN, and the terminal status.

The function uses the `cretc` function to create a new entry. It passes 4 bytes of data (the terminal status and the terminal CRI) to the program (TSTC) that processes the new entry. This program is provided in file DXCBINQ1.

```
/*
*====================================================================*
*       DXCBINQM -- SAMPLE C APPLICATION FOR ALCS V2                 *
*====================================================================*
* Purpose:                                                           *
*         Sample application to demonstrate ALCS V2 C support        *
*                                                                    *
*====================================================================*
*/


#include "dxcbinqh.h"

/* define macro for error conditions             */

#define PUTERR(msg)  puts(msg); error = TRUE;

static char usage[] = {"\
```

```
\n monitor terminal status\
\n\
\n  parameters : cri\
\n    or         crn\
\n\
\n  the CRN, CRI and logged on status of the terminal are displayed\
"
};


void dxcbinqm( char *ptr)
{
  int cri = 0;
  char error = FALSE;
  struct co0ic *cptr;
  struct monitor_data data;

  #ifdef DEBUG
    printf("DXCBINQM started: parms are:%s\n",ptr);
  #endif

  if( *ptr == 0)
  {
    PUTERR("terminal ID not specified");
  }
  else
  {
    if( (cri = getcri(ptr)) == GETCRI_ERROR)
    {
      PUTERR("terminal ID is not valid CRI/CRN");
    }
  }

  if (error)
  {
    puts(usage);
  }
  else
  {


    /*============================================================*/
    /*                                                            */
    /* start the monitor by issuing a time initiated create     */
    /* to the TSTC monitor program                                */
    /*                                                            */
    /*============================================================*/

    cptr = showcri(cri);

    data.cri    = cri;
    data.status = cptr->icescst;
    data.unused = 0;
    cretc(CRETC_MINUTES,MONITOR_PROG,MONITOR_TIME,&data);

    /* display the status of the terminal */

    printf("started monitor: CRN=%8.8s CRI=%06X Status=%s\n",
           cptr->icecrn,
           cptr->icecri,
           (cptr->icescst)?"active":"inactive");
  }

  #ifdef DEBUG
    puts("DXCBINQM ended");
  #endif
  return;
}
```

DXCBINQP

This function is called by DXCBINQ0. It display the storage address of a program, given the 4-byte program name.

To do this, it calls the assembler program DXCBINQ3, which has been mapped to the C function `find_prog` in the header file DXCBINQH.

```
/*
*====================================================================*
*       DXCBINQP -- SAMPLE C APPLICATION FOR ALCS V2                 *
*====================================================================*
* Purpose:                                                           *
*         Sample application to demonstrate ALCS V2 C support        *
*                                                                    *
*====================================================================*
*/

#include "dxcbinqh.h"


static char usage[] = {"\
\nUsage: display the address of an ALCS program\
\n\
\n  parameters : prog\
\n\
\n  where prog is the 4 character program name\
"
};


void  dxcbinqp( char *prog)
{

  unsigned char *address;

  #ifdef DEBUG
    printf("DXCBINQP started parameters:%s\n", prog);
  #endif

  address = find_prog(prog);

  if (address != NULL)
  {
    printf("program %4.4s is at address %08X\n",
            prog, address);
  }
  else
  {
    if ( valid_program_name(prog) )
    {
      printf("program '%4.4s' not found\n",prog);
    }
    else
    {
      if (prog[0] == 0)
      {
        printf("no program name specified\n");
      }
      else
      {
        printf("'%-4.4s' is not a valid program name\n",prog);
      }
      puts(usage);
    }

  }

  return;
```

```
}
```

Running the application

Before you can use this sample application, you or the system programmer must define the application to ALCS as a communication resource. See the *ALCS Installation and Customization*.

The remainder of this section assumes that you define the application with a CRN of SAMP by coding a line similar to the following in your ALCS communication generation:

```
         COMDEF CRN=SAMP,PROG=TST1,...
```

To run the application, you need to do the following:

1. Ensure that the application is active

You can test if the application SAMP is active by keying in at any CRAS terminal the command:

```
   zdcom n=samp
```

If the reply says that SAMP is active, as in the example below, continue from "2. Ensure that your terminal is routed to the application" on page 198, below.

```
zdcom n=samp
21.33.00 DCOM
Resource CRN      CRI    Ordinal  Routing  Status
ALCSAPPL SAMP     040010 00000470 TST1     ACTIVE
```

If SAMP is inactive, you can make it active by keying in the following sequence of commands on the Prime CRAS terminal:

```
   zdcom n=samp
   zacom n=samp,act
   zdcom n=samp
```

These commands are explained in the *ALCS Operation and Maintenance*. The result should be similar to that shown below:

```
zdcom n=samp
21.32.33 DCOM
Resource CRN      CRI    Ordinal  Routing  Status
ALCSAPPL SAMP     040010 00000470 TST1     INACTIVE
zacom n=samp,act
21.32.45 ACOM  Resource set active
zdcom n=samp
21.33.00 DCOM
Resource CRN      CRI    Ordinal  Routing  Status
ALCSAPPL SAMP     040010 00000470 TST1     ACTIVE
```

2. Ensure that your terminal is routed to the application

You must next route input messages from your own terminal to the communication resource SAMP (the application program TST1). You must also load the application load module DXCBINQ. To do this, key in the following sequence of commands at your own terminal:

```
   zpctl l t dxcbinq
   zrout samp
```

These commands are explained in the *ALCS Operation and Maintenance*. The result should be similar to that shown below:

```
zpctl l t dxcbinq
21.31.03 PCTL  MODULE LOADED
zrout samp
21.27.36 ROUT  OK
```

Your terminal should now respond to input in a similar way to that shown in “Sample output” on page 184.

Debugging statements

Some program source code files contain statements such as the following:

```
#ifdef DEBUG
  puts("DXCBINQM ended");
#endif
```

These statements show how you can incorporate debugging statements into your program. To activate these statements, compile the program source file using the compiler parameter `DEF(DEBUG)`.

When you omit the `DEF(DEBUG)` parameter, the compiler ignores the statements between the `#ifdef` and `#endif`. Compile the program in this way when you have debugged it and want to create a normal working version.

Appendix C. Sample application: COBOL

This appendix lists a complete sample application written in COBOL. The application uses callable services written in C and assembler

Purpose of the application

The DXCQCOB sample program demonstrates how you can write COBOL programs that run under ALCS. A COBOL program can access ALCS services through callable services written in C or assembler. COBOL programs can also make use of LE callable services.

DXCQCOB takes as input a 2-character country identifier. It uses LE callable services and this identifier to display the current time in the appropriate national time and date format. If LE does not recognize the country code the program displays the time using a default format. The 2-character country codes, and the LE callable services, are documented in the Language Environment *Programming Reference*.

DXCQCOB also displays information about the terminal that is being used, the version of LE that is being used, and the ALCS system time.

Running the application

The example assumes that the DXCQCOB program is loaded and activated under ALCS as application TST1. For example, on prime CRAS:

```
zasys norm
DXC8235I CMD M  07.03.22 ASYS
ALCS state change from IDLE to NORM starting
zpctl l s dxcqcob
DXC8340I CMD M  07.03.35 PCTL
Module MODN-'DXCQCOB' loaded by CRN-'PGBS21EV'
zacom n=tst1 act
DXC8124I CMD M  07.03.47 ACOM  Resource set active
```

You could execute DXCQCOB by routing a terminal to the TST1 application:

```
zrout tst1
DXC8523I CMD M  07.04.42 ROUT  Terminal routing updated
```

Sample output

Some example transations are shown below:

```
us
Terminal is: PYESSZO2 Ordinal is: 000000266 DBCS is: not supported
LE Version is: 000000120 Platform ID is: 000000003
MVS date/time in US      format: 05/06/94 3:15:13 PM
local ALCS time/date: 07.05.09 89.09.16
local MVS  time/date: 15.15.13 94.05.06
ALCS has been up: 00 hrs 03 mins 09 secs


xx
Terminal is: PYESSZO2 Ordinal is: 000000266 DBCS is: not supported
LE Version is: 000000120 Platform ID is: 000000003
MVS date/time in DEFAULT format: 1994-05-06 15:15:59
local ALCS time/date: 07.06.00 89.09.16
local MVS  time/date: 15.15.59 94.05.06
ALCS has been up: 00 hrs 04 mins 00 secs
```

Source files

The program source files are related as shown in Figure 115 on page 201.

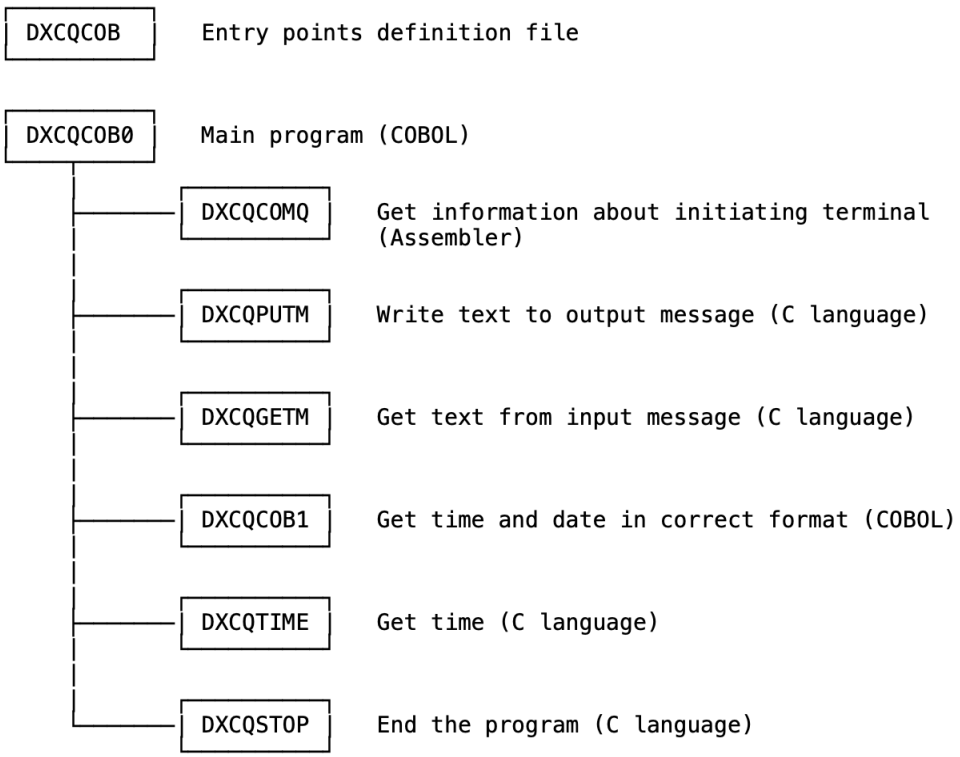


Figure 115. Modules in the high-level language sample application

Note: DXCQCOB1 uses LE callable services CEE*xxx*.

The files are listed in the order shown in Figure 115 on page 201.

DXCQCOB

```
*================================================================*
*       DXCQCOB  -- SAMPLE ENTRY POINTS DEFINITION FILE          *
*================================================================*
* PURPOSE:                                                       *
*                                                                *
*   DEFINE THE ENTRY POINT IN THE SAMPLE APPLICATION AS AN       *
*   ALCS APPLICATION PROGRAM.                                    *
*                                                                *
*   WHEN ALCS EXECUTES THE PROGRAM TST1 THE SAMPLE APPLICATION   *
*   IS EXECUTED AT ENTRY POINT DXCQCOB0 IN PROGRAM DXCQCOB0      *
*                                                                *
*================================================================*
*
       BEGIN NAME=TST1,VERSION=00
       TRANV NAME=TST1,ENTRY=DXCQCOB0
             MODULE=(DXCQCOB0,DXCQCOB1,DXCQCOMQ,DXCQTIME,
                     DXCQPUTM,DXCQGETM,DXCQSTOP)
```

DXCQCOB0 (COBOL source)

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID. DXCQCOB0.
*================================================================*
*       DXCQCOB0 -- SAMPLE ALCS APPLICATION IN COBOL             *
*================================================================*
*                                                                *
* PURPOSE:                                                       *
*                                                                *
*  A SAMPLE COBOL PROGRAM TO ILLUSTRATE THE USE OF               *
*  CALLABLE SERVICES PROVIDED BY LE AND USER WRITTEN             *
*  CALLABLE SERVICES FOR ALCS IN C AND ASSEMBLER                 *
*                                                                *
*  INPUT:                                                        *
*                                                                *
*  INPUT TO THE PROGRAM IS A 2-CHARACTER COUNTRY IDENTIFIER      *
*  WHICH IS USED TO RETRIEVE A SPECIFIC TIME AND DATE FORMAT     *
*                                                                *
*  OUTPUT:                                                       *
*                                                                *
*  THIS APPLICATION DISPLAYS:                                    *
*                                                                *
*   - THE TERMINAL'S CRN AND ORDINAL AND DBCS SUPPORT STATUS     *
*   - THE LE VERSION NUMBER AND PLATFORM IDENTITY                *
*   - THE DATA AND TIME IN THE SPECIFIED COUNTRY FORMAT          *
*   - ALCS AND MVS DATE/TIME INFORMATION AND ALCS UPTIME         *
*                                                                *
*  EXTERNAL CALLS:                                               *
*                                                                *
*  THE FOLLOWING CALLABLE SERVICES ARE USED:                     *
*                                                                *
*  - CEEGPID  -- GET THE LE VERSION NUMBER AND PLATFORM ID       *
*  - DXCQCOB1 -- GET TIME/DATE STAMP IN COUNTRY SPECIFIC FORMAT  *
*  - DXCQCOMQ -- GET INFORMATION ABOUT THE ORIGINATING TERMINAL  *
*  - DXCQGETM -- GET TEXT FROM INPUT MESSAGE                     *
*  - DXCQPUTM -- PUT TEXT TO OUTPUT MESSAGE                      *
*  - DXCQSTOP -- EXIT THE PROGRAM WITH/WITHOUT A DUMP            *
*  - DXCQTIME -- GET TIME/DATE INFORMATION FROM ALCS             *
*                                                                *
*  THE LE CALLABLE SERVICES START WITH 'CEE' AND ARE             *
*  DESCRIBED IN THE LE PROGRAMMING GUIDE                         *
*                                                                *
*----------------------------------------------------------------*
/----------------------------------------------------------------*
*                                                                *
*  ERROR RETURN:                                                 *
*                                                                *
*  THE FOLLOWING ERROR DUMPS CAN OCCUR:                          *
*                                                                *
*  - FFFF00 -- ERROR-COMQ-RC                                     *
*              UNEXPECTED ERROR OCCURRED IN THE DXCQCOMQ SERVICE *
*                                                                *
*  - FFFF01 -- ERROR-PUTM-RC                                     *
*              UNEXPECTED ERROR OCCURRED WRITING TEXT TO THE     *
*              OUTPUT MESSAGE                                    *



*                                                                *
*  - FFFF02 -- ERROR-GETM-RC                                     *
*              UNEXPECTED ERROR OCCURRED READING TEXT FROM THE   *
*              INPUT MESSAGE                                     *
*                                                                *
*  - FFFF03 -- ERROR-GPID-RC                                     *
*              UNEXPECTED ERROR OCCURRED IN THE CEEGPID SERVICE  *
*                                                                *
*  - FFFF04 -- ERROR-NOT-SCR                                     *
*              THE ORIGINATING TERMINAL IS NOT A DISPLAY SCREEN  *
*                                                                *
*================================================================*
```

```
/
 ENVIRONMENT DIVISION.

 DATA DIVISION.
   WORKING-STORAGE SECTION.

   01   TIMSTP        PIC X(80).

   01   FC.
   02   SEV           PIC S9(4) COMP.
   02   MSGNO         PIC S9(4) COMP.
   02   FLGS          PIC X(1).
   02   FACID         PIC X(3).
   02   ISI           PIC S9(9) COMP.

   01   COMQ-DATA.
   02   CRI           PIC S9(9) COMP VALUE ZERO.
   02   CRN           PIC X(8)       VALUE SPACES.
   02   ORDINAL       PIC S9(9) COMP VALUE ZERO.
   02   ACRI          PIC S9(9) COMP VALUE ZERO.
   02   TERM-TYPE     PIC X(1)       VALUE SPACES.
   02   DBCS-SUPPORT  PIC X(1)       VALUE SPACES.

*  DEFINE WORKING VARIABLES AND BUFFERS

   77   RC            PIC S9(9) COMP.
   77   VERSION       PIC S9(9) COMP.
   77   PLATID        PIC S9(9) COMP.
   77   COUNTRY       PIC X(2).

   77   BLENGTH       PIC S9(9) COMP.
   77   TLENGTH       PIC S9(9) COMP.
   77   TIME-BUFFER1  PIC X(40).
   77   TIME-BUFFER2  PIC X(40).
   77   TIME-BUFFER3  PIC X(40).

*  DEFINE SUCCESFUL RETURN CODE

   77   EXIT-OK-RC    PIC S9(9) COMP VALUE ZERO.

/  DEFINE ERROR NUMBERS - X'FFFF00' TO X'FFFF04'

   77   ERROR-COMQ-RC PIC S9(9) COMP VALUE 16776960 .
   77   ERROR-PUTM-RC PIC S9(9) COMP VALUE 16776961 .
   77   ERROR-GETM-RC PIC S9(9) COMP VALUE 16776962 .
   77   ERROR-GPID-RC PIC S9(9) COMP VALUE 16776963 .
   77   ERROR-NOT-SCR PIC S9(9) COMP VALUE 16776964 .



*  DEFINE MESSAGES ISSUED

   77   MSGL00        PIC 9(9) COMP VALUE 66.
   01   MSG00.
   02   FILLER        PIC X(13) VALUE 'Terminal is: '.
   02   MSG00-CRN     PIC X(8).
   02   FILLER        PIC X(13) VALUE ' Ordinal is: '.
   02   MSG00-ORDINAL PIC 9(9).
   02   FILLER        PIC X(10) VALUE ' DBCS is: '.
   02   MSG00-DBCS    PIC X(13) VALUE 'not supported'.

   77   MSGL01        PIC 9(9) COMP VALUE 50.
   01   MSG01.
   02   FILLER        PIC X(15) VALUE 'LE Version is: '.
   02   MSG01-VER     PIC 9(9).
   02   FILLER        PIC X(17) VALUE ' Platform ID is: '.
   02   MSG01-PID     PIC 9(9).

   77   MSGL02        PIC 9(9) COMP VALUE 78.
   01   MSG02.
   02   FILLER        PIC X(17) VALUE
                      'MVS date/time in '.
```

```
   02   MSG02-COUNTRY PIC X(7).
   02   FILLER        PIC X(9) VALUE
                      ' format: '.
   02   MSG02-TIMSTP  PIC X(45).

   77   MSGL04        PIC 9(9) COMP VALUE 60.
   01   MSG04.
   02   FILLER        PIC X(60) VALUE
        'DXCQCOB1 is unable to get and/or format the date/time'.

 LINKAGE SECTION.
/
 PROCEDURE DIVISION.

*-----------------------------------------------------------------*
*    GET COMMUNICATIONS INFORMATION FROM ALCS                     *
*-----------------------------------------------------------------*
     CALL 'DXCQCOMQ' USING COMQ-DATA , RC.
     PERFORM CHECK-RC-COMQ.
*-----------------------------------------------------------------*
*    GET COMMUNICATIONS INFORMATION FROM ALCS                     *
*-----------------------------------------------------------------*
     MOVE CRN     TO MSG00-CRN.
     MOVE ORDINAL TO MSG00-ORDINAL.
     IF DBCS-SUPPORT IS EQUAL TO 'D' THEN
       MOVE 'supported' TO MSG00-DBCS.
     CALL 'DXCQPUTM' USING MSG00 MSGL00 RC.
     PERFORM CHECK-RC-PUTM.
*-----------------------------------------------------------------*
*    GET AND DISPLAY THE LE VERSION AND PLATFORM IDENTITY         *
*-----------------------------------------------------------------*
     CALL 'CEEGPID' USING VERSION PLATID FC.
     PERFORM CHECK-FC-GPID.
     MOVE VERSION TO MSG01-VER.
     MOVE PLATID  TO MSG01-PID.
     CALL 'DXCQPUTM' USING MSG01 MSGL01 RC.
     PERFORM CHECK-RC-PUTM.



*-----------------------------------------------------------------*
*    GET THE COUNTRY IDENTIFIER FROM THE INPUT MESSAGE            *
*-----------------------------------------------------------------*

     MOVE 2    TO BLENGTH.
     CALL 'DXCQGETM' USING COUNTRY BLENGTH TLENGTH RC.
     PERFORM CHECK-RC-GETM.
     IF TLENGTH NOT EQUAL TO 2 THEN
       MOVE SPACES TO COUNTRY.

*-----------------------------------------------------------------*
*    CALL THE COBOL SUBROUTINE 'DXCQCOB1' TO GET AND              *
*    FORMAT THE DATE AND TIME                                     *
*-----------------------------------------------------------------*
     CALL 'DXCQCOB1' USING COUNTRY TIMSTP FC.

*    IF AN ERROR OCCURED IN DXCQCOB1 DISPLAY A MESSAGE            *
     IF  SEV > 0 THEN
       CALL 'DXCQPUTM' USING MSG04 MSGL04 RC
       PERFORM CHECK-RC-PUTM.

     MOVE TIMSTP TO MSG02-TIMSTP.

     IF  COUNTRY IS EQUAL TO SPACES THEN
       MOVE 'DEFAULT' TO MSG02-COUNTRY
     ELSE
       MOVE COUNTRY   TO MSG02-COUNTRY.

     CALL 'DXCQPUTM' USING MSG02 MSGL02 RC.
     PERFORM CHECK-RC-PUTM.
/-----------------------------------------------------------------*
*    GET TIME INFORMATION FROM ALCS                               *
```

```
*-----------------------------------------------------------------*
     CALL 'DXCQTIME' USING TIME-BUFFER1
                           TIME-BUFFER2
                           TIME-BUFFER3.

     MOVE 40 TO BLENGTH.
     CALL 'DXCQPUTM' USING TIME-BUFFER1 BLENGTH RC.
     PERFORM CHECK-RC-PUTM.
     CALL 'DXCQPUTM' USING TIME-BUFFER2 BLENGTH RC.
     PERFORM CHECK-RC-PUTM.
     CALL 'DXCQPUTM' USING TIME-BUFFER3 BLENGTH RC.
     PERFORM CHECK-RC-PUTM.

*-----------------------------------------------------------------*
*    END OF PROGRAM - CALL DXCQSTOP TO SEND REPLY AND EXIT       *
*-----------------------------------------------------------------*
     CALL 'DXCQSTOP' USING EXIT-OK-RC.
/-----------------------------------------------------------------*
*    PROCEDURE TO CHECK THE RETURN CODE (RC) AFTER USING         *
*    DXCQCOMQ AND EXIT WITH ERROR IF RC IS NOT ZERO              *
*    ALSO CHECK THAT THE TERMINAL IS A DISPLAY SCREEN            *
*    AND EXIT WITH ERROR IF IT IS NOT A DISPLAY SCREEN           *
*-----------------------------------------------------------------*
 CHECK-RC-COMQ.
     IF RC IS NOT EQUAL TO ZERO THEN
       CALL 'DXCQSTOP' USING ERROR-COMQ-RC.

     IF TERM-TYPE IS NOT EQUAL TO 'S' THEN
       CALL 'DXCQSTOP' USING ERROR-NOT-SCR.



*-----------------------------------------------------------------*
*    PROCEDURE TO CHECK THE RETURN CODE (RC) AFTER USING         *
*    DXCQPUTM AND EXIT WITH ERROR IF RC IS NOT ZERO              *
*-----------------------------------------------------------------*
 CHECK-RC-PUTM.
     IF RC IS NOT EQUAL TO ZERO THEN
       CALL 'DXCQSTOP' USING ERROR-PUTM-RC.

*-----------------------------------------------------------------*
*    PROCEDURE TO CHECK THE RETURN CODE (RC) AFTER USING         *
*    DXCQGETM AND EXIT WITH ERROR IF RC IS GREATER THAN 4        *
*-----------------------------------------------------------------*
 CHECK-RC-GETM.
     IF RC IS GREATER THAN 4 THEN
       CALL 'DXCQSTOP' USING ERROR-GETM-RC.

*-----------------------------------------------------------------*
*    PROCEDURE TO CHECK THE FEEDBACK STRUCTURE AFTER USING       *
*    CEEGPID AND EXIT WITH ERROR IF SEV IS GREATER THAN 4        *
*-----------------------------------------------------------------*
 CHECK-FC-GPID.
     IF SEV IS NOT EQUAL TO ZERO THEN
       CALL 'DXCQSTOP' USING ERROR-GPID-RC.

*-----------------------------------------------------------------*
*    END OF PROGRAM DXCQCOB0                                     *
*-----------------------------------------------------------------*
```

DXCQPUTM

```
/*==================================================================*/
/*       DXCQPUTM -- SAMPLE HLL CALLABLE SERVICE IN C               */
/*==================================================================*/
/* Purpose:                                                         */
/*                                                                  */
/*   callable service to write message text to stdout               */
/*                                                                  */
/*   the text is written to the output file                         */
```

```
/*                                                                   */
/*   parameters are:                                                 */
/*                                                                   */
/*     - address of buffer containing the text (input)               */
/*     - length of the message                 (input)               */
/*     - pointer to a word for the return code (output)              */
/*                                                                   */
/*       the value of the return code is:                            */
/*         0 - if the text was sucessfully added to stdout           */
/*         8 - if an error occured when adding the text              */
/*                                                                   */
/*===================================================================*/

#pragma strings(readonly)           /* strings are constants         */
#pragma options(NOSTART,RENT)       /* options for compiler          */
#pragma csect(CODE,"DXC_PUTM")      /* name for code CSECT           */
#pragma csect(STATIC,"DXC$PUTM")    /* name for static data CSECT    */

#pragma linkage(dxcqputm,OS)        /* use OS style parameter list   */

#include <tpfeq.h>                  /* include ALCS header files     */
#include <tpfapi.h>
#include <tpfio.h>

#include <stdio.h>                  /* include standard headers      */
#include <stddef.h>
#include <string.h>
#include <ctype.h>

#include <leawi.h>                  /* for LE     interlanguage      */
                                    /* data type definitions         */

void dxcqputm(
  _POINTER    Buffer,
  _INT4       Count,
  _INT4       rc
)
{


  /*-----------------------------------------------------------------*/
  /* initialize the return code to zero                              */
  /*-----------------------------------------------------------------*/
  rc = 0;

  /*-----------------------------------------------------------------*/
  /* write the message to the output file                            */
  /*-----------------------------------------------------------------*/

  if ( printf("%-*.*s\n",Count,Count,Buffer) < 0 )
  {
    rc = 8;    /* error occurred  in printf */
  }

  return;
}
```

DXCQGETM

```
/*===================================================================*/
/*       DXCQGETM -- SAMPLE HLL CALLABLE SERVICE IN C                */
/*===================================================================*/
/* Purpose:                                                          */
/*                                                                   */
/*   callable service to get message text from the input message     */
/*                                                                   */
/*   parameters are:                                                 */
/*                                                                   */
/*     - address of buffer to receive the text (input/output)        */
/*     - size of the buffer                    (input)               */
```

```
/*     - length of text returned in the buffer (output)             */
/*     - the return code                       (output)             */
/*                                                                  */
/*   text is read from the input message until the buffer is        */
/*   full or an end of line/end of file/error occurs                */
/*                                                                  */
/*   the number of characters stored in the buffer is returned      */
/*                                                                  */
/*   the remainder of the buffer is set to binary zeros             */
/*   (note: if the buffer is full a terminating character of        */
/*          binary zero is not stored)                              */
/*                                                                  */
/*   the return code indicates if an error or end of file occured   */
/*                                                                  */
/*   the value of the return code is:                               */
/*                                                                  */
/*         0 - if the text was sucessfully read from stdin          */
/*         4 - if the end of file was reached                       */
/*         8 - if an error occured when reading the text            */
/*                                                                  */
/*==================================================================*/

#pragma strings(readonly)           /* strings are constants        */
#pragma options(NOSTART,RENT)       /* options for compiler         */
#pragma csect(CODE,"DXC_GETM")      /* name for code CSECT          */
#pragma csect(STATIC,"DXC$GETM")    /* name for static data CSECT   */

#pragma linkage(dxcqgetm,OS)        /* use OS style parameter list  */

#include <tpfeq.h>                  /* include ALCS header files    */
#include <tpfapi.h>
#include <tpfio.h>

#include <stdio.h>                  /* include standard headers     */
#include <stddef.h>
#include <string.h>
#include <ctype.h>

#include <leawi.h>                  /* for LE     interlanguage     */

void dxcqgetm(
  _POINTER    Buffer,
  _INT4       Buffer_Size,
  _INT4       Text_Length,
  _INT4       rc
)


{
int i, ch;
char *p;

  /*--------------------------------------------------------------*/
  /* initialize return code and length read to zero               */
  /*--------------------------------------------------------------*/
  Text_Length = rc = 0;

  /*--------------------------------------------------------------*/
  /* initialize buffer to all zeros                               */
  /*--------------------------------------------------------------*/
  memset(Buffer, 0, Buffer_Size);

  /*--------------------------------------------------------------*/
  /* read message text from the input file                        */
  /*--------------------------------------------------------------*/

  p = Buffer;  /* point to the start of Buffer */

  /*--------------------------------------------------------------*/
  /* read characters until buffer full, end of file or new line */
  /*--------------------------------------------------------------*/
  for (i=0;
```

```
       i < Buffer_Size             &1       /* end of buffer ? */
           (ch = getchar()) != EOF &1       /* end of file   ? */
            ch != '\n' ;                     /* new line      ? */
       ++i )
  {
    *p++ = ch;                       /* add character to Buffer */
  }

  if (ch == EOF)
  {
    /*-------------------------------------------------------------*/
    /* end-of-file or error occurred  - set the return code        */
    /*-------------------------------------------------------------*/

    if (feof(stdin))
    {
      rc = 4;  /* EOF   */
    }
    else
    {
      rc = 8;  /* error */
    }
  }

  Text_Length = i;   /* set the length */
  return;
}
```

DXCQCOB1

```
 IDENTIFICATION DIVISION.
   PROGRAM-ID. DXCQCOB1.

*=================================================================*
*       DXCQCOB1 -- SAMPLE ALCS CALLABLE SERVICE IN COBOL         *
*=================================================================*
*                                                                 *
* PURPOSE:                                                        *
*                                                                 *
*  DEMONSTRATE A CALLABLE SERVICE WRITTEN AS A COBOL SUBROUTINE   *
*                                                                 *
*  THIS ROUTINE IS CALLED BY THE DXCQCOB0 PROGRAM TO GET THE      *
*  LOCAL DATE AND TIME IN A COUNTRY SPECIFIC FORMAT               *
*                                                                 *
*  THIS PROGRAM DEMONSTRATES THE USE OF DEBUGGING LINES           *
*  (WITH 'D' IN COLUMN 7). THESE LINES ARE ONLY COMPILED WHEN     *
*  'WITH DEBUGGING MODE' IS SPECIFIED ON THE SOURCE COMPUTER      *
*  STATEMENT                                                      *
*                                                                 *
*  THE DIAGNOSTIC MESSAGES PRODUCED BY THE DISPLAY STATEMENTS     *
*  ARE DISPLAYED WHEN THIS PROGRAM IS RUN WITH CONVERSATIONAL     *
*  TRACE                                                          *
*                                                                 *
*  INPUT:                                                         *
*                                                                 *
*  - COUNTRY -- A 2-CHARACTER COUNTRY IDENTIFIER                  *
*  - TIMESTP -- A 80-CHARACTER BUFFER TO HOLD THE RESULT          *
*  - FC      -- A 12-BYTE FEEDBACK CODE FOR USE WITH THE          *
*               LE CALLABLE SERVICES                              *
*                                                                 *
*  OUTPUTS:                                                       *
*                                                                 *
*  - COUNTRY -- UNCHANGED OR SET TO SPACES IF INPUT IS INVALID    *
*  - TIMESTP -- THE FORMATTED TIME STAMP IS SEVERITY IS ZERO      *
*  - FC      -- A 12-BYTE FEEDBACK CODE WITH SEVERITY SET TO      *
*               ZERO IF THIS SERVICE COMPLETED NORMALLY OR        *
*               NON-ZERO IF AN ERROR OCCURED                      *
*                                                                 *
*  EXTERNAL CALLS:                                                *
*                                                                 *
```

```
*  THE FOLLOWING LE CALLABLE SERVICES ARE USED:                    *
*                                                                  *
*  - CEEDATM  -- CONVERT SECONDS TO CHARACTER TIMESTAMP            *
*  - CEEFMDT  -- OBTAIN DEFAULT DATE AND TIME FORMAT               *
*  - CEELOCT  -- GET CURRENT LOCAL TIME                            *
*                                                                  *
*  THE LE CALLABLE SERVICES AND THE USE OF THE FEEDBACK CODE       *
*  ARE DESCRIBED IN THE LE PROGRAMMING GUIDE                       *
*                                                                  *
*  MORMAL RETURN:                                                  *
*                                                                  *
*   - THE SEVERITY FIELD IN THE FEEDBACK CODE IS SET TO ZERO       *
*                                                                  *
*   - THE TIME AND DATE IN THE LE COUNTRY SPECIFIC FORMAT          *
*     IN THE TIMESTP BUFFER                                        *
*                                                                  *


*   - IF THE COUNTRY ID IS NOT RECOGNISED BY THE CEEFMDT SERVICE   *
*     THE TIME AND DATE IS RETURNED IN THE DEFAULT FORMAT          *
*     AND THE COUNTRY PARAMETER IS SET TO SPACES                   *
*                                                                  *
*                                                                  *
*  ERROR RETURN:                                                   *
*                                                                  *
*   IF AN ERROR OCCURS THE FEEDBACK CODE CONTAINS A NON-ZERO       *
*   VALUE IN THE SEVERITY FIELD AND DIAGNOSTIC INFORMATION         *
*   AS SET BY THE FAILING LE     CALLABLE SERVICE CEEDATM OR       *
*   CEELOCT                                                        *
*                                                                  *
*------------------------------------------------------------------*

 ENVIRONMENT DIVISION.

   CONFIGURATION SECTION.
   SOURCE-COMPUTER. IBM-390 WITH DEBUGGING MODE.
   OBJECT-COMPUTER. IBM-390.

 DATA DIVISION.

   WORKING-STORAGE SECTION.

* DEFINE LOCAL VARIABLES

   77   LILLIAN       PIC S9(9) COMP.
   77   SECONDS       COMP-2 VALUE ZERO.
   77   GREGORN       PIC X(17).

   01   PICSTR.
   02   PICSTRL       PIC 9(4) COMP VALUE 80.
   02   PICSTRT       PIC X(80).

 LINKAGE SECTION.

* DEFINE PARAMETERS PASSED FROM/RETURNED TO THE CALLING PROGRAM

   01   COUNTRY       PIC X(2).
   01   TIMSTP        PIC X(80).

   01   FC.
   02   SEV           PIC S9(4) COMP.
   02   MSGNO         PIC S9(4) COMP.
   02   FLGS          PIC X(1).
   02   FACID         PIC X(3).
   02   ISI           PIC S9(9) COMP.
/
 PROCEDURE DIVISION USING COUNTRY TIMSTP FC.

*------------------------------------------------------------------*
*    GET THE SELECTED LE     DATE AND TIME FORMAT                  *
*------------------------------------------------------------------*
```

```
     CALL 'CEEFMDT' USING COUNTRY , PICSTRT , FC.

*    IF ERROR OCCURS INDICATE DEFAULT BY SETTING COUNTRY TO SPACES
     IF SEV > 0 THEN
       MOVE SPACES TO COUNTRY.



*-----------------------------------------------------------------*
*    GET THE LOCAL DATE AND TIME                                  *
*-----------------------------------------------------------------*

     CALL 'CEELOCT' USING LILLIAN , SECONDS , GREGORN , FC.

*    IF ERROR OCCURED RETURN TO CALLER
     IF SEV > 0 THEN
D      DISPLAY 'CEELOCT SEV=' SEV ' MSGNO=' MSGNO
D      DISPLAY 'CEELOCT GREGORN=' GREGORN ' SECONDS=' SECONDS
       GOBACK.

*-----------------------------------------------------------------*
*    CONVERT TO THE DATE TIME IN SELECTED FORMAT                  *
*-----------------------------------------------------------------*

     CALL 'CEEDATM' USING SECONDS , PICSTR , TIMSTP , FC.

D    IF SEV > 0 THEN
D      DISPLAY 'CEEDATM TIMSTP=' TIMSTP
D      DISPLAY 'CEEDATM SEV=' SEV ' MSGNO=' MSGNO.

     GOBACK.

*-----------------------------------------------------------------*
*    END OF PROGRAM DXCQCOB1                                      *
*-----------------------------------------------------------------*
```

DXCQTIME

```
/*====================================================================*/
/*       DXCQTIME -- SAMPLE HLL CALLABLE SERVICE IN C                 */
/*====================================================================*/
/*                                                                    */
/* Purpose:                                                           */
/*                                                                    */
/*   callable service to get time information from ALCS               */
/*                                                                    */
/*   parameters are:                                                  */
/*                                                                    */
/*     - address of 40 byte buffer for local ALCS time text (output) */
/*     - address of 40 byte buffer for local MVS  time text (output) */
/*     - address of 40 byte buffer for ALCS uptime  text    (output) */
/*                                                                    */
/*   on return the buffers have been updated with the                 */
/*   time information obtained with the ALCS timec() service          */
/*                                                                    */
/*====================================================================*/

#pragma strings(readonly)           /* strings are constants         */
#pragma options(NOSTART,RENT)       /* options for compiler          */
#pragma csect(CODE,"DXC_TIME")      /* name for code CSECT           */
#pragma csect(STATIC,"DXC$TIME")    /* name for static data CSECT    */

#pragma linkage(dxcqtime,OS)        /* use OS style parameter list   */

#include <tpfeq.h>                  /* include ALCS header files     */
#include <tpfapi.h>
#include <tpfio.h>

#include <stdio.h>                  /* include standard headers      */
#include <stddef.h>
```

```
#include <string.h>
#include <ctype.h>

#include <leawi.h>                  /* for LE interlanguage          */
                                    /* data type definitions         */

void dxcqtime(
  _POINTER    Buffer1,
  _POINTER    Buffer2,
  _POINTER    Buffer3
)
{
#define TIME_SIZE 20
char time_buf[TIME_SIZE];
int  time_int;
char tempbuf[80];


  /*-----------------------------------------*/
  /* get the local ALCS time and date        */
  /*-----------------------------------------*/

  memset(time_buf,0,TIME_SIZE);           /* clear temporary buffer */

  timec(TIMEC_TIMEDATE,                   /* get time information   */
        TIMEC_YMD,
        TIMEC_LOCAL,
        TIMEC_ALCS,
        time_buf);

  /* format time information and move to result buffer 1            */

  sprintf(tempbuf, "local ALCS time/date: %s", time_buf);
  sprintf(Buffer1, "%-40.40s", tempbuf);

  /*-----------------------------------------*/
  /* get the local MVS time and date         */
  /*-----------------------------------------*/

  memset(time_buf,0,TIME_SIZE);           /* clear temporary buffer */

  timec(TIMEC_TIMEDATE,                   /* get time information   */
        TIMEC_YMD,
        TIMEC_LOCAL,
        TIMEC_HOST,
        time_buf);

  /* format time information and move to result buffer 2            */

  sprintf(tempbuf, "local MVS  time/date: %s", time_buf);
  sprintf(Buffer2, "%-40.40s", tempbuf);

  /*-----------------------------------------*/
  /* get the time that ALCS has been rumnning */
  /*-----------------------------------------*/

  timec(TIMEC_SECONDS,                    /* get time information   */
        TIMEC_YMD,
        TIMEC_LOCAL,
        TIMEC_ALCS,
        22:09:06_int);

  /* format time information and move to result buffer 3            */

  sprintf(tempbuf, "ALCS has been up: %02d hrs %02d mins %02d secs",
          time_int/3600,
          (time_int%3600)/60,
          time_int%60 );
  sprintf(Buffer3, "%-40.40s", tempbuf);

  return;
```

```
}
```

DXCQSTOP

```
/*====================================================================*/
/*       DXCQSTOP -- SAMPLE HLL CALLABLE SERVICE IN C                 */
/*====================================================================*/
/* Purpose:                                                           */
/*                                                                    */
/*   callable service to exit the HLL environment                     */
/*                                                                    */
/*   use this service to ensure that any response message             */
/*   in stdout is written to the terminal                             */
/*                                                                    */
/*   parameters are:                                                  */
/*                                                                    */
/*     -  a completion code for exit()                                */
/*        non zero value will produce a system error dump             */
/*                                                                    */
/*   this service does not return to the caller                       */
/*                                                                    */
/*====================================================================*/

#pragma strings(readonly)           /* strings are constants         */
#pragma options(NOSTART,RENT)       /* options for compiler          */
#pragma csect(CODE,"DXC_STOP")      /* name for code CSECT           */
#pragma csect(STATIC,"DXC$STOP")    /* name for static data CSECT    */

#pragma linkage(dxcqstop,OS)        /* use OS style parameter list   */

#include <tpfeq.h>                  /* include ALCS header files     */
#include <tpfapi.h>

#include <leawi.h>                  /* for LE interlanguage          */
                                    /* data type definitions         */
void dxcqstop(_INT4 cc)
{
  exit(cc);
}
```

DXCQCOMQ

```
*====================================================================*
*        DXCQCOMQ --  GET COMMUNICATIONS INFORMATION                 *
*====================================================================*
*                                                                    *
*        A SAMPLE CALLABLE SERVICE WRITTEN IN ASSEMBLER FOR USE BY   *
*        HIGH LEVEL LANGUAGE PROGRAMS                                *
*                                                                    *
*        MUST BE ASSEMBLED WITH PARAMETER "SYSPARM(DXCLIBC)"         *
*                                                                    *
*        THIS SERVICE RECEIVES A OS STYLE PARAMETER LIST IN THE      *
*        FOLLOWING FORMAT:                                           *
*                                                                    *
*        PARAMETER 1                                                 *
*                                                                    *
*          0(R06)   -  POINTER TO A STRUCTURE IN THE FORMAT:         *
*                                                                    *
*                     BYTES    USAGE                                 *
*                     -----    -------                               *
*                       4      CRI                                   *
*                       8      CRN                                   *
*                       4      ORDINAL NUMBER                        *
*                       4      CRI OF ASSOCIATED TERMINAL            *
*                              INDICATORS SET FOR:                   *
*                       1        TERMINAL IS A SCREEN OR PRINTER     *
```

```
*                     1        TERMINAL SUPPORTS DBCS               *
*                                                                   *
*                                                                   *
*        PARAMETER 2                                                *
*                                                                   *
*          4(R06)   -  POINTER TO A FULLWORD TO CONTAIN THE RETURN  *
*                      CODE. THIS IS SET TO ZERO IF THE SERVICE     *
*                      SUCCEEDED OR NON ZERO IF THE SERVICE FAILED. *
*                                                                   *
*        ON COMPLETION AND IF THE RETURN CODE IS ZERO THE           *
*        STRUCTURE HAS BEEN UPDATED WITH THE COMMUNICATIONS TABLE   *
*        INFORMATION FOR THE TERMINAL THAT INITIATED THIS ENTRY.    *
*                                                                   *
*        THE INDICATORS ARE SET TO:                                 *
*          S -- THE TERMINAL IS A DISPLAY SCREEN                    *
*          P -- THE TERMINAL IS A PRINTER                           *
*          D -- DBCS IS SUPPORTED                                   *
*                                                                   *
*===================================================================*
         SPACE 1
         BEGIN  NAME=DXCQCOMQ
         ICPLOG ,
         SPACE 1
******** GET PARAMETERS
         SPACE 1
         L     R05,0(R06)          SET POINTER TO INPUT STRUCTURE
         L     R04,4(R06)          SET POINTER TO RETURN CODE
         XC    0(4,R04),0(R04)     SET RETURN CODE TO ZERO
         SPACE 1
         USING COMQSTR,R05         ESTABLISH ADDRESSABILITY
         SPACE 1
******** GET THE COMMUNICATIONS INFORMATION FOR THE SPECIFIED CRI
         SPACE 1
         XC    COMQSTR(COMQLEN),COMQSTR SET STRUCTURE TO ALL ZEROS
         MVC   COMQCRI+1(3),EBROUT COPY CRI TO STRUCTURE
         SPACE 1
         CO0IC REG=R14             ADDRESS THE RESULT FROM COMIC
         SPACE 1


         COMIC CRI=EBROUT,DATA=SYS,AREA=(0,ICELN2)
         SPACE 1
         BNZ   ERROR               BRANCH IF AN ERROR OCCURED
         SPACE 1
******** COPY THE REQUIRED INFORMATION TO STRUCTURE
         MVC   COMQCRN,ICECRN      CRN
         MVC   COMQARC,ICEARC      ASSOCIATED RECOURCE CRI
         MVC   COMQORN,ICEORN      ORDINAL NUMBER
         TM    ICESDBC,L'ICESDBC   IS DBCS SUPPORTED?
         BZ    NODBCS              BRANCH IF DBCS NOT SUPPPORTED
         SPACE 1
         MVI   COMQDBCS,C'D'       INDICATE DBCS SUPPORTED
         SPACE 1
NODBCS   DC    0H'0'
         SPACE 1
         TM    ICESTPP,L'ICESTPP   IS RESOURCE A PRINTER?
         BZ    NOPRT               BRANCH IF NOT A PRINTER
         SPACE 1
         MVI   COMQTERM,C'P'       INDICATE THIS IS A PRINTER
         SPACE 1
NOPRT    DC    0H'0'
         SPACE 1
         TM    ICESTPS,L'ICESTPS   IS RESOURCE A SCREEN?
         BZ    NOSCR               BRANCH IF NOT A SCREEN
         SPACE 1
         MVI   COMQTERM,C'S'       INDICATE THIS IS A SCREEN
         SPACE 1
NOSCR    DC    0H'0'
         B     ALLDONE             NO MORE TO DO - RETURN TO CALLER
         SPACE 1
ERROR    DC    0H'0'
         LA    R00,8               GET A NON ZERO VALUE
```

```
         ST    R00,0(R04)          STORE IN RETURN CODE
         SPACE 1
ALLDONE  DC    0H'0'
         SPACE 1
******** RESTORE AND RETURN
         SPACE 1
         ICELOG ,                  RETURN TO CALLER
         SPACE 1
         LTORG ,
         SPACE 1
COMQSTR  DSECT ,
COMQCRI  DC    F'0'               CRI OF TERMINAL (LAST 3 BYTES)
COMQCRN  DC    CL8'........'      CRN OF TERMINAL
COMQORN  DC    F'0'               COMMUNICATIONS RESOURCE ORDINAL
COMQARC  DC    F'0'               CRI OF ASSOCIATED TERMINAL
COMQTERM DC    CL1'.'             'S' FOR SCREEN 'P' FOR PRINTER
COMQDBCS DC    CL1'.'             'D' IF DBCS SUPPORTED ELSE X'00'
COMQLEN  EQU   *-COMQSTR          LENGTH OF STRUCTURE
         RSECT ,
         SPACE 1
         FINIS ,
         SPACE 1
         END   ,
```

Appendix D. Messages and message formats

This appendix shows details of the input and output messages and shows details of the message formats used by ALCS communication resources. It also describes the contents of the application routing control parameter list (RCPL).

Input and output messages

In assembler programs you can access the input message (stored on level 0) directly. You can create a scrollable output messages for printer or display terminals by using the DISPC macro. This is the recommended method. Alternatively, you can create an output messages in the form described in “Standard format for output message text” on page 217 and output this to one or more terminals using SENDC or ROUTC.

In assembler programs you can also use 3270 mapping to send or receive messages. See Appendix E, “3270 Screen mapping support for application programs,” on page 226 for more details.

TPF compatibility:

Do not use DISPC in programs that must be compatible with TPF.

In C language programs you can use standard C functions to "read" the input message from `stdin`. Similarly, you can use standard C functions to "write" the reply message to `stdout`. This is the recommended method.

Note: ALCS does not receive or transmit messages character-by-character. When you "read" from `stdin`, your C program retrieves characters from a complete input message (it does *not* read characters directly from the terminal keyboard). Similarly, when you "write" to `stdout`, your C program builds a message that ALCS sends when the program ends (it does not *not* write characters directly to the terminal display).

Transmission codes

Communication resources that connect to ALCS use a variety of transmission codes for character data, for example, EBCDIC, ASCII, BAUDOT, ALC, and so on. They also support a variety of control characters for functions such as new line, end-of-message, and so on.

ALCS application programmers do not normally need to be aware of these differences, because ALCS passes input message text to application programs in a standard format. Similarly, ALCS application programs build output message text in a standard format.

Standard format for input message text

The standard format for input message text is a single EBCDIC character string representing the characters keyed in at the originating terminal - regardless of the transmission code that the terminal uses.

When defining your application to ALCS, the system programmer specifies whether or not the application can process mixed case input (see the description of the COMDEF macro for a local application in *ALCS Installation and Customization*). If your application can not process mixed case input, ALCS translates all alphabetic characters in the input message text to upper case.

If the input message consists of more than one line, the standard format for input message text indicates the end of every line except the last with a single EBCDIC new-line character (X'15'). The last or only line of the input message ends with an ALCS end-of-message character (X'4E').

Note: The ALCS end-of-message character is the EBCDIC plus (+) character. Some old applications were originally developed for use with ALC terminal equipment, which cannot enter or display the plus character. These applications sometimes assume that any plus character in an input message is the end-of-message. This assumption is unsafe with equipment such as IBM 3270 which does support the

© Copyright IBM Corp. 2003, 2019

plus character. Instead, you must use the message length field (AM0CCT or CM1CCT) to determine the message length.

For assembler language programs, ALCS defines the symbols:

Symbol
Meaning

#CAR
New-line (X'15')

#EOM
End-of-message (X'4E')

In C language programs, messages in `stdin` have standard end-of-line (`\n`) and end-of-message (`\0`) delimiters.

Program function keys

Some terminal equipment has program function keys that transmit specially-encoded input messages to ALCS. ALCS recognizes these specially-encoded input messages and replaces them with standard-format input message text before it passes the message to the application program.

Printer acknowledgements (answerbacks)

Printer terminals send a specially-encoded input message, sometimes called an answerback, to acknowledge receipt of text for printing. Application programs do not normally process these messages (ALCS printer support normally receives them). If you are developing a specialized application that does process answerbacks (perhaps a ticket or boarding-pass printing application) you must ask your system programmer for information about the format of these answerback messages.

Double-byte character set (DBCS) messages

Input messages from terminals that support double-byte character sets contain special shift-in (X'0F') and shift-out (X'0E') characters to delimit sequences of pairs of bytes - each pair of bytes represents a single character. For assembler language programs, ALCS defines the symbols:

Symbol
Meaning

#SI
shift-in (X'0F')

#SO
shift-out (X'0E')

In C language programs, you can use `mbstowcs` to convert input message text from DBCS terminals into a wide-character string (`wchar_t`). You can then use C functions (for example, `wcschr`) to process these wide-character strings. When you do this, you do not need to examine the individual characters (for example, the shift-in and shift-out character) of the input message.

WTTY and type B messages

Some specialized applications process XMSG format input messages that conform to the ATA/IATA Interline Conventional Message Format (Type B). ALCS passes these messages to application programs using the standard format for input message text, except:

- The #CAR (X'15') character represents carriage-return, not new-line. The EBCDIC percent (%, X'6C') represents line-feed. Each line of the message (except the last) is terminated by the 2-character sequence: carriage-return, line-feed.

  For assembler language programs, ALCS defines the symbol:

Symbol
Meaning

#LFEED
line-feed (X'6C')

- The text is not terminated with an end-of-message character. Instead it is terminated by an ATA/IATA Interline Conventional Message end-of-message signal (typically 4 consecutive N characters).

Standard format for output message text

The standard format for output message text is a single EBCDIC character string representing the characters to display at the destination terminal - regardless of the transmission code that the terminal uses. Alphabetic characters can be in upper or lower case - if the destination terminal cannot display mixed case text then ALCS translates lower case characters to upper case.

If the output message consists of more than one line, you must indicate the end of each line except the last with a single EBCDIC new-line character (X'15'). Every line of the output message *must contain at least one character*. To include a "blank" line in an output message, use a single EBCDIC space (X'40') character for the line text.

TPF compatibility:

TPF allows output messages to contain consecutive new-line characters. ALCS requires new-line characters to be separated by (at least) a single character. By including a single space character, you can make your programs compatible with both ALCS and TPF.

Starting sequences

Although it is not strictly part of the output message text, ALCS requires you to supply a two-character prefix for output message text in the OMSG or AMSG format. These characters are called the **command word** and the **line address**.

ALCS requires you to store particular values in these two characters as follows:

- For messages to displays: the command word must contain one of the write control characters deined in the table below, the line address must contain either the display line (row) number in binary plus X'80' or new-line (X'15'). New-line displays the output message on the line that immediately follows the input message.
- For messages to printers: the command word must contain no-operation (X'6C'), the line address must contain new-line (X'15').

ALCS defines the symbols:

Character	Hex value	Assembler symbol	C symbol
Write-and-erase	X'61'	#WEW	_WEW
New-line	X'15'	#CAR	_CAR
No-operation	X'6C'	#NOP	_NOP
Write-continue-at-cursor	X'6C'	#WCW	_WCW
Write-on-specified-line	X'12'	#WLA	_WLA

When you use `stdout` you do not need to build a starting sequence for output messages. C language programs can create messages in OMSG or AMSG format. They can do so for printer messages and for messages output using `routc`.

Ending sequences

The last, or only, line of the output message must end with one of the following ending sequences:

For messages to displays

new-line (X'15'), start-of-message (X'6E'), end-of-message (X'4E'). This sequence positions the cursor at the start of a new-line ready for the end-user to type the next input message.

For messages to printers

new-line (X'15'), end-of-message-unsolicited (X'5F').

ALCS defines the symbols:

Character	Hex value	Assembler symbol	C symbol
New-line	X'15'	#CAR	_CAR
Start-of-message	X'6E'	#SOM	_SOM
End-of-message	X'4E'	#EOM	_EOM
End-of-message-unsolicited	X'5F'	#EOU	_EOU
End-of-message-incomplete	X'6D'	#EOI	_EOI
End-of-text	X'03'	#ETX	_ETX

In C language programs, messages in `stdout` have standard C language end-of-line (`\n`) and end-of-message (`\0`) delimiters.

Restricted character sets

When constructing output messages, note that some terminal equipment supported by ALCS does not support the full EBCDIC character set. As explained above, you can include mixed-case alphabetic text in messages even for terminals that only support upper case (ALCS translates the text to upper case for these terminals). But some punctuation marks do not display correctly at ALC terminals. Also, characters such as currency symbols and accented characters may display differently (or not at all) at different terminals.

ALC/3270 compatibility:

Avoid including percent (%, X'6C') characters in output messages. ALC interprets X'6C' as a "NOP" (no-operation) character and removes them from the output message text.

Presentation space size

When constructing output messages, note that there is a limit to the number of characters that can appear on a single line (row) of a display or printer terminal. Also, for displays there is a limit to the number of lines that can appear on the display. The number of lines and the number of characters per line is often called the **presentation space size**.

Different terminals supported by ALCS have different presentation space sizes. The following table summarizes the presentation space sizes of terminal equipment commonly connected to ALCS systems:

Terminal type	Characters per line	Lines
ALC displays	64	12, 15, or 30
ALC printers	64	-
3270 displays	79 or more	23 or more
3270 printers	120	-
WTTY printers	70	-

Notes:

1. In the above table, "3270 displays" and "3270 printers" refers to IBM 3270 terminal equipment, and compatible terminals and workstations.
2. On an IBM 3270 display, the last line on the screen and the first character of each line is used for control information and cannot be used by applications.
3. ALCS does not restrict the number of characters that you send to printer terminals. The effect of sending lines longer than the printer supports depends on the type of printer. The printer may automatically insert new-line characters as required, or it may simply overtype excess characters.

If you use the ALCS scrolling facility to send your output messages, you do not need to worry about the presentation space size of the display terminal. The ALCS scrolling facility displays as much of your output message as possible, and provides commands that allow the end user to view the rest of the message (if any). In assembler language programs, you use the `DISPC` macroinstruction to pass lines of your output message to the ALCS scrolling facility.

Double-byte character set (DBCS) messages

Output messages to terminals that support double-byte character sets can contain special shift-in (X'0F') and shift-out (X'0E') characters to delimit sequences of pairs of bytes - each pair of bytes represents a single character. For assembler language programs, ALCS defines the symbols:

Symbol
Meaning

#SI
shift-in (X'0F')

#SO
shift-out (X'0E')

In C language programs, you prepare the output message as a wide-character output string (`wchar_t`). You use `wctomb` to convert this to DBCS format before using `puts`, or similar function.

WTTY and type B messages

Some specialized applications send XMSG format output messages that conform to the ATA/IATA Interline Conventional Message Format (Type B). These programs construct the output message text in the standard format, except that:

- The #CAR (X'15') represents carriage-return, not new-line. The EBCDIC percent (%, X'6C') represents line-feed. For assembler language programs, ALCS defines the symbol:

  Symbol
  Meaning

  #LFEED
  line-feed (X'6C')

  (The terminals used for these messages do not support the new-line control character. Instead, they support the carriage-return character which positions the print head at the start of the line without advancing the paper. and the line-feed character which advances the paper one line. You achieve the effect of a new-line character by including a carriage-return character followed by a line-feed character.)
- The text can include special control characters such as figure-shift and letter-shift. Note that you do *not* normally need to include these shift characters in your text. ALCS ensures that shift characters are inserted in the text, if required, during transmission. For example, if the terminal requires it, a figure-shift is automatically inserted before a numeric character that follows an alphabetic character. But if (for example) you need to include a figure-shift character immediately after a numeric character then you must explicitly include it in your output message text.

  For assembler language programs, ALCS defines the symbols:

  Symbol
  Meaning

#FSC
figure-shift (X'5C')

#LSC
letter-shift (X'7C')

- The text is not terminated with an end-of-message character. Instead it is terminated by an ATA/IATA Interline Conventional Message end-of-message signal (typically 4 consecutive N characters).

The different standard input and output message formats are:

IMSG
Standard format for input messages.

OMSG
Standard format for output messages.

AMSG
Standard format for input and output messages. This format is used with ROUTC and the RCPL.

XMSG
Standard format for input and output messages. Used by specialized applications that process messages in the ATA/IATA Interline Conventional Message format (Type B). The following ALCS communication resources use XMSG format:

- WTTY links
- AX.25 links, for Type B messages
- SLC links, for Type B messages
- SLC links, for Type A messages to or from another host system

Message format - detail

This section lists the individual fields in each of the following message formats:

```
IMSG
OMSG
AMSG input
AMSG output
XMSG for WTTY
XMSG for SLC and AX.25
```

For assembler programs, the CM1CM DSECT macro defines names for fields within these formats. For C programs, the <c$cm1cm.h> header file defines names for these fields.

Not all fields defined by CM1CM and <c$cm1cm.h> are general-use programming interfaces (GUPIs). The following sections describe the ones that are.

IMSG

Fields in an input message in IMSG format are as follows:

Assembler name	C name	Length (bytes)	Contents	Notes
CM1BID	cm1bid	2	'IM'	
CM1CCT	cm1cct	2	message text length + 3	length of message, plus 3 bytes
CM1CCX	cm1ccx	4	message text length + 3	length of message, plus 3 bytes. Used instead of CM1CCT/cm1cct when in extended format.

Assembler name	C name	Length (bytes)	Contents	Notes
CM1CRI	cm1cri	3	source terminal CRI	
CM1TXI	cm1txi	variable	message text	see “Standard format for input message text” on page 215

OMSG

Fields in an output message in OMSG format are as follows:

Name (Assembler)	Name (C)	Length (bytes)	Contents	Notes
CM1BID	cm1bid	2	'OM'	
CM1CCT	cm1cct	2	message text length + 5	length of message, plus 5 bytes
CM1CCX	cm1ccx	4	message text length + 5	length of message, plus 5 bytes. Used instead of CM1CCT/ cm1cct when in extended format.
CM1CRI	cm1cri	3	The CRI of the destination resource. ALCS ignores this field for ROUTC messages. The destination address is obtained from the RCPL (see “RCPL contents” on page 223).	
CM1CMW	cm1cmw	1	command word	see “Starting sequences” on page 217
CM1LNA	cm1lna	1	screen line number or new-line (X'15')	see “Starting sequences” on page 217
CM1TXT	cm1txt	variable	message text	see “Standard format for output message text” on page 217

AMSG input

Fields in an input message in AMSG format are as follows:

Assembler name	C name	Length (bytes)	Contents	Notes
AMORID	am0rid	2	'MI'	
AM0CCT	am0cct	2	message text length + 5	length of message, plus 5 bytes
AM0CCX	am0ccx	4	message text length + 5	length of message plus 5 bytes
AM0TXT	am0txt	variable	message text in IMSG format	see “Standard format for input message text” on page 215

AMSG output

Fields in an output message in AMSG format are as follows:

Assembler name	C name	Length (bytes)	Contents	Notes
AM0RID	`am0rid`	2	'OM'	
AM0CCT	`am0cct`	2	message text length + 5	length of message, plus 5 bytes
AM0CCX	`am0ccx`	4	message text length + 5	length of message plus 5 bytes
AM0NP1	`am0np1`	1	no operation code	code #NOP (assembler) or _NOP (C language)
AM0NP2	`am0np2`	1	no operation code	code #NOP (assembler) or _NOP (C language)
AM0TXT	`am0txt`	variable	message text in OMSG format, including a starting sequence	see "Standard format for output message text" on page 217 (including **Starting sequences**)

XMSG for WTTY (input and output)

Fields in an input (WTTY) or output (SENDC T) message in XMSG format are as follows:

Assembler name	Length (bytes)	Contents	Notes
CM1BID	2	'TM'	
CM1SWX	1	indicators (WTTY input)	
CM1TSC	2	WTTY transmitter start code	
CM1CCT	2	message text length + 1	length of message, plus 1 byte
CM1LYN	1	line number	the low order byte of the CRI of the WTTY link
CM1TTY	variable	message segment	see "WTTY and type B messages" on page 216

Note: In a XMSG format message each segment might be a complete message, part of a message, the last part of a message, a garbled message, and so on. This is indicated by bit settings in the header field.

XMSG for SLC and AX.25 (input and output)

Fields in an input (SLC or AX.25) or output (SENDC K) message in XMSG format are as follows:

Assembler name	Length (bytes)	Contents	Notes
CM1BID	2	'TM'	
CM1HEX	2	high-level network exit address (HEX)	Required for SENDC K output type A message only

Assembler name	Length (bytes)	Contents	Notes
CM1TCI	1	terminal circuit identifier (TCID)	Required for SENDC K output messages whose destination is SLC-ID
CM1SWX	2	indicators (SENDC K)	
CM1CCT	2	message text length + 1	length of message, plus 1 byte
CM1LYN	1	line number	The low order byte of the CRI of the SLC or X.25 link
CM1SLC	variable	message text	see "WTTY and type B messages" on page 216

RCPL contents

A RCPL can be in basic format (12 bytes) or expanded format (16 through 98 bytes). Control byte 0 (RCPLTL0) indicates which.

RCPL input

The fields set up by ALCS in the RCPL when it provides an input message are as follows. The last 3 fields are for expanded format only.

Assembler name	C name	Length (bytes)	Contents	Notes
RCPLDES	`rcpldes`	4	destination of message	Either an application name or a CRI (CRI is in the high-order 3 bytes)
RCPLORG	`rcplorg`	4	origin of message	Either an application name or a CRI (CRI is in the high-order 3 bytes)
RCPLCTL0	`rcplctl0`	1	control byte 0 (see note "1" on page 224)	Bit settings indicate if: • Destination is an application name or a CRI • Origin is an application name or a CRI • RCPL is in basic (or expanded) format
RCPLCTL2 (or RCPLMSN)	`rcplctl2`	1	control byte 2 (see note "1" on page 224)	Bit settings indicate whether or not the meassage is: • Recoverable • Possibly a duplicate • Includes a function management header (FMH)
RCPLGDD	`rcplgdd`	1	switches	Top 4 bits (0 through 3) are reserved for ALCS, others can be used by applications

Assembler name	C name	Length (bytes)	Contents	Notes
RCPLCTR	`rcplctr`	1	data length	Number of bytes of data in the next field (0 through 82)
RCPLGDA	`rcplgda`	variable	general data (see note "2" on page 224)	Maximum size is 82 bytes

Notes:

1. *(LBS1)* For the meanings of bit settings in fields RCPLCTL0 and RCPLCT2, see the `rc0pl` DSECT macro description in *ALCS Application Programming Reference - Assembler*. For the bit settings in the corresponding C fields, see *ALCS Application Programming Reference - C Language*.
2. *(LBS2)* The general data in RCPLGDA consists of a 1-byte length field (RCPLDLEN) followed by the data field (RCPLDATA). There are no equivalent C names. This field is provided for compatibility with TPF.

RCPL output

The fields which an application might initialize when it provides an output message to ALCS are as follows. The last 3 fields are for expanded format only.

Note: An application only needs to set up the RCPL when it sends a message to a different terminal from that which input the original message.

Assembler name	C name	Length (bytes)	Contents	Notes
RCPLDES	`rcpldes`	4	destination of message	Either an application name or a CRI (If a CRI, it is held in the high-order 3 bytes)
RCPLORG	`rcplorg`	4	origin of message	Either an application name or a CRI (If a CRI, it is held in the high-order 3 bytes)
RCPLCTL0	`rcplctl0`	1	control byte 0 (see note "1" on page 225)	Bit settings indicate if: • Destination is an application name or a CRI • Message is unsolicited (or a response) • Message is being returned to originator (or not) • RCPL is in basic (or expanded) format
RCPLCTL2 (or RCPLMSN)	`rcplctl2`	1	control byte 2 (see note "1" on page 225)	Bit settings indicate if: • ROUTC requests ALCS to release storage block (or not) • Message is in OMSG/IMSG format (depending on TYPE parameter of ROUTC macro), or AMSG format. • Message includes a function management header (FMH)

Assembler name	C name	Length (bytes)	Contents	Notes
RCPLGDD	`rcplgdd`	1	switches	Top 4 bits (0 through 3) are reserved for ALCS, others can be used by applications
RCPLCTR	`rcplctr`	1	data length	Number of bytes of data in the next field (0 through 82)
RCPLGDA	`rcplgda`	variable	general data (see note "2" on page 225)	Maximum size is 82 bytes

Notes:

1. *(LBS1O)* For the meanings of bit settings in fields RCPLCTL0 and RCPLCT2, see the `rc0pl` DSECT macro description in *ALCS Application Programming Reference - Assembler*. For the bit settings in the corresponding C fields, see *ALCS Application Programming Reference - C Language*.
2. *(LBS2O)* The general data in RCPLGDA consists of a 1-byte length field (RCPLDLEN), followed by the data field (RCPLDATA). There are no equivalent C names. This field is provided for compatibility with TPF.

Appendix E. 3270 Screen mapping support for application programs

This appendix briefly describes the ALCS 3270 screen mapping support, and explains how you can prepare your installation for application programs which use this feature.

Screen mapping enables an application programmer to define the layout of a 3270 screen, and to use it easily from an application. Screen mapping allows an end user to enter a complete transaction on one screen by filling in fields instead of entering the data as a string of messages.

The process for implementing screen mapping in an application includes the following steps:

1. The application programmer codes a **map description** to define the screen layout. A map description consists of a sequence of MAP3270 macroinstructions which specify the positions, lengths, colors, and so on of the various fields on the screen.

   ALCS Application Programming Reference - Assembler explains how to code MAP3270 macroinstructions.

   If an application uses several screen layouts then it requires several map descriptions. Each map description has a unique map name.
2. The application programmer or system programmer stores each map description from step “1” on page 226 as a member of a source statement library, preferably with the same name as the map name.
3. The application programmer writes the application programs that use the screen layouts defined in step “1” on page 226. The programmer must include the map descriptions in the relevant programs by coding assembler COPY instructions to copy the descriptions from the source statement library (see step “2” on page 226).

   The application program must also call the ALCS ECB-controlled programs which extract input fields from the input 3270 data stream and build the output 3270 data stream. See “Screen mapping functions” on page 226.

   You can then assemble the application programs from step “3” on page 226 in the normal way (remember to include in SYSLIB the source statement library that contains the map descriptions - see step “2” on page 226). You can also load the programs on to ALCS, but before you can execute them, you must build **maps** from the map descriptions that the programs use, and load these maps on to the ALCS real-time database, as follows:
4. The system programmer builds an ALCS input sequential file containing the maps by:

   a. Assembling the map descriptions with the assembler parameter SYSPARM(GENERATE), using the MVS JCL para PARM.

   b. Copying the SYSPUNCH output from step “4.a” on page 226 to a sequential data set using the IBM utility program IEBGENER. This data set is the input sequential file for ALCS.

   ALCS includes an ISPF panel to run these two steps. If you prefer to build your own JCL for the process, you can use the panel to create a sample job.
5. The operator uses the ZCMSP command to load the maps from the sequential file (created in step “4” on page 226) onto the ALCS real-time database.

   ALCS Operation and Maintenance describes the ZCMSP command.

Screen mapping functions

Two programs form the interface to ALCS screen mapping support.

CSMI
Maps a 3270 input data stream (the data received from a terminal), into the fixed format described by the map DSECT.

CSMO
Converts the data, in the fixed format described by the map DSECT, into a 3270 output data stream, ready for transmission by SENDC D.

For further information about CSMI and CSMO, see *ALCS Application Programming Reference - Assembler*.

Acronyms and abbreviations

The following acronyms and abbreviations are used in books of the ALCS Version 2 library. Not all are necessarily present in this book.

AAA
agent assembly area

ACB
VTAM access method control block

ACF
Advanced Communications Function

ACF/NCP
Advanced Communications Function for the Network Control Program, usually referred to simply as "NCP"

ACF/VTAM
Advanced Communications Function for the Virtual Telecommunication Access Method, usually referred to simply as "VTAM"

ACK
positive acknowledgment (SLC LCB)

ACP
Airline Control Program

AID
IBM 3270 attention identifier

AIX
Advanced Interactive eXecutive

ALC
airlines line control

ALCI
Airlines Line Control Interconnection

ALCS/MVS/XA
Airline Control System/MVS/XA

ALCS/VSE
Airline Control System/Virtual Storage Extended

ALCS V2
Airline Control System Version 2

AML
acknowledge message label (SLC LCB)

AMS
access method services

AMSG
AMSG application message format

APAR
authorized program analysis report

APF
authorized program facility

API
application program interface

APPC
advanced program-to-program communications

ARINC
Aeronautical Radio Incorporated

ASCU
agent set control unit (SITA), a synonym for "terminal control unit"

AT&T
American Telephone and Telegraph Co.

ATA
Air Transport Association of America

ATSN
acknowledge transmission sequence number (SLC)

BATAP
Type B application-to-application program

BSC
binary synchronous communication

C
C programming language

CAF
Call Attach Facility

CCW
channel command word

CDPI
clearly differentiated programming interface

CEC
central electronic complex

CEUS
communication end-user system

CI
VSAM control interval

CICS
Customer Information Control System

CLIST
command list

CMC
communication management configuration

CML
clear message label (synonym for AML)

COBOL
COmmon Business Oriented Language

CPI-C
Common Programming Interface - Communications

CPU
central processing unit

CRAS
computer room agent set

CRI
communication resource identifier

CRN
communication resource name

CSA
common service area

CSECT
control section

CSID
cross system identifier

CSW
channel status word

CTKB
Keypoint record B

CTL
control system error

CUA
Common User Access

DASD
direct access storage device

DBCS
double-byte character set

DBRM
DB2 database request module

DB2
IBM DB2 for z/OS (refers to DB2)

DCB
data set control block

DECB
ALCS data event control block

DF
delayed file record

DFDSS
Data Facility Data Set Services

DFHSM
Data Facility Hierarchical Storage Manager

DFP
Data Facility Product

DFSMS
Data Facility Storage Management Subsystem

DFT
distributed function terminal

DIX
delete item index

DRIL
data record information library

DSI
direct subsystem interface

DSECT
dummy control section

DTP
ALCS diagnostic file processor

EBCDIC
extended binary-coded decimal interchange code

ECB
ALCS entry control block

EIB
error index byte

EID
event identifier

EJB
Enterprise Java Bean

ENQ
enquiry (SLC LCB)

EOF
end of file

EOM
end of message

EOI
end of message incomplete

EOP
end of message pushbutton

EOU
end of message unsolicited

EP
Emulation Program

EP/VS
Emulation Program/VS

ETX
end of text

EvCB
MVS event control block

EXCP
Execute Channel Program

FACE
file address compute

FIFO
first-in-first-out

FI
file immediate record

FM
function management

FMH
function management header

GB
gigabyte (1 073 741 824 bytes)

GDS
general data set

GFS
get file storage (called pool file storage in ALCS)

GMT
Greenwich Mean Time

GTF
generalized trace facility (MVS)

GUPI
general-use programming interface

HEN
high-level network entry address

HEX
high-level network exit address

HFS
Hierarchical File System

HLASM
High Level Assembler

HLL
high-level language

HLN
high-level network

HLS
high-level system (for example, SITA)

HTTP
Hypertext Transfer Protocol

IA
interchange address

IASC
International Air Transport Solution Centre

IATA
International Air Transport Association

IATA5
ATA/IATA transmission code 5

IATA7
ATA/IATA transmission code 7

ICF
integrated catalog facility

ICSF
Integrated Cryptographic Service Facility

ID
identifier

ILB
idle (SLC LCB)

IMA
BATAP acknowledgement

IMS
Information Management System

IMSG
IMSG input message format

I/O
input/output

IOCB
I/O control block

IP
Internet Protocol

IPARS
International Programmed Airlines Reservation System

IPCS
Interactive Problem Control System

IPL
initial program load

ISA
initial storage allocation

ISC
intersystem communication

ISO/ANSI
International Standards Organization/American National Standards Institute

ISPF
Interactive System Productivity Facility

ISPF/PDF
Interactive System Productivity Facility/Program Development Facility

ITA2
International Telegraph Alphabet number 2

JCL
job control language

JES
job entry subsystem

JNDI
Java Naming and Directory Interface

JSON
JavaScript Object Notation

KB
kilobyte (1024 bytes)

KCN
link channel number (SLC)

KSDS
VSAM key-sequenced data set

LAN
local area network

LCB
link control block (SLC)

LDB
link data block (SLC)

LDI
local DXCREI index

LEID
logical end-point identifier

LE
Language Environment

LICRA
Link Control - Airline

LMT
long message transmitter

LN
line number (ALCS/VSE and TPF terminology)

LN/ARID
line number and adjusted resource identifier (ALCS/VSE terminology)

LSET
Load set

LSI
link status identifier (SLC)

LU
logical unit

LU 6.2
Logical Unit 6.2

MATIP
Mapping of airline traffic over IP

MB
megabyte (1 048 576 bytes)

MBI
message block indicator (SLC)

MCHR
module/cylinder/head/record

MESW
message switching

MNOTE
message note

MQI
Message Queueing Interface

MQM
Message Queue Manager

MSNF
Multisystem Networking Facility

MVS
Multiple Virtual Storage (refers to MVS) (refers to both MVS/XA and MVS/ESA, and also to OS/390 and z/OS)

MVS/DFP
Multiple Virtual Storage/Data Facility Product

MVS/ESA
Multiple Virtual Storage/Enterprise System Architecture

MVS/XA
Multiple Virtual Storage/Extended Architecture

NAB
next available byte

NAK
negative acknowledgment (SLC LCB)

NCB
network control block (SLC)

NCP
Network Control Program (refers to ACF/NCP)

NCP/VS
Network Control Program/Virtual Storage.

NEF
Network Extension Facility

NEF2
Network Extension Facility 2

NPDA
Network Problem Determination Application

NPSI
Network Control Program packet switching interface

NTO
Network Terminal Option

OCR
one component report

OCTM
online communication table maintenance

OLA
optimized local adapters

OMSG
OMSG output message format

OPR
operational system error

OSID
other-system identification

OS/2
IBM Operating System/2®

PARS
Programmed Airlines Reservation System

PDF
parallel data field (refers to NCP)

PDM
possible duplicate message

PDS
partitioned data set

PDSE
partitioned data set extended

PDU
pool directory update

PER
program event recording

PFDR
pool file directory record

PL/I
programming language one

PLM
purge long message (name of ALCS/VSE and TPF general tape)

PLU
primary logical unit

PNL
passenger name list

PNR
passenger name record

PP
IBM program product

PPI
program-to-program interface

PPMSG
program-to-program message format

PPT
program properties table

PR
permanently resident record

PRC
prime computer room agent set

PRDT
physical record (block) descriptor table

PRPQ
programming request for price quotation

PR/SM
Processor Resource/Systems Manager

PS
VTAM presentation services

PSPI
product sensitive programming interface

PSW
program status word

PTF
program temporary fix

PTT
Post Telephone and Telegraph Administration

PU
physical unit

PVC
permanent virtual circuit

QSAM
queued sequential access method

RACF
resource access control facility

RB
request block

RBA
relative byte address

RCC
record code check

RCPL
routing control parameter list

RCR
resource control record

RCS
regional control center

RDB
Relational Database

RDBM
Relational Database Manager

REI
resource entry index

RLT
record locator table

RMF
Resource Measurement Facility

RO CRAS
receive-only computer room agent set

RON
record ordinal number

RPL
VTAM request parameter list

RPQ
request for price quotation

RSM
resume (SLC LCB)

RTM
recovery and termination management

RU
request unit

SAA
Systems Application Architecture®

SAL
system allocator list (TPF terminology)

SAM
sequential access method

SDLC
Synchronous Data Link Control

SDMF
standard data and message file

SDSF
System Display and Search Facility

SDWA
system diagnostic work area

SI
DBCS shift in

SITA
Société Internationale de Télécommunications Aéronautiques

SLC
ATA/IATA synchronous link control

SLIP
serviceability level indication processing

SLN
symbolic line number

SLR
Service Level Reporter

SLU
secondary logical unit

SMP/E
System Modification Program Extended

SNA
Systems Network Architecture

SO
DBCS shift out

SON
system ordinal number

SQA
system queue area

SQL
Structured Query Language

SQLCA
SQL Communication Area

SQLDA
SQL Descriptor Area

SRB
service request block

SRG
statistical report generator

SRM
System Resource Manager

STC
system test compiler

STP
stop (SLC LCB)

STV
system test vehicle

SWB
service work block

SYN
character synchronization character

TA
terminal address

TAS
time available supervisor

TCB
task control block

TCID
terminal circuit identity

TCP/IP
Transmission Control Protocol / Internet Protocol

TI
time-initiated record

TOD
time of day

TPF
Transaction Processing Facility

TPF/APPC
Transaction Processing Facility/Advanced Program to Program Communications

TPF/DBR
Transaction Processing Facility/Data Base Reorganization

TPFDF
TPF Database Facility

TPF/MVS
Transaction Processing Facility/MVS (alternative name for ALCS V2)

TP_ID
transaction program identifier

TSI
transmission status indicator

TSN
transmission sequence number

TSO
time-sharing option

TSO/E
Time Sharing Option Extensions

TUT
test unit tape (sequential file)

UCB
unit control block

UCTF
Universal Communications Test Facility

VFA
virtual file access

VIPA
virtual IP address

VM
virtual machine

VM/CMS
virtual machine/conversational monitor system

VS
virtual storage

VSAM
virtual storage access method

VSE
Virtual Storage Extended

VSE/AF
Virtual Storage Extended/Advanced Function

VSE/VSAM
Virtual Storage Extended/Virtual Storage Access Method

VTAM
Virtual Telecommunications Access Method (refers to

VTOC
volume table of contents

WAS
WebSphere Application Server (refers to WebSphere)

WSF
Write Structured Field

WTTY
World Trade Teletypewriter

XMSG
XMSG message switching message format

XREF
ALCS cross referencing facility

Glossary

Notes:

1. Acronyms and abbreviations are listed separately from this Glossary. See "Acronyms and abbreviations" on page 228.
2. For an explanation of any term not defined here, see the IBM *Dictionary of Computing*.

A

AAA hold
See terminal hold.

abnormal end of task (abend)
Termination of a task before its completion because of an error condition that cannot be resolved by recovery facilities while the task is executing.

access method services (AMS)
A utility program that defines VSAM data sets (or files) and allocates space for them, converts indexed sequential data sets to key-sequenced data sets with indexes, modifies data set attributes in the catalog, facilitates data set portability between operating systems, creates backup copies of data sets and indexes, helps make inaccessible data sets accessible, and lists data set records and catalog entries.

activity control variable
A parameter that ALCS uses to control its workload. The system programmer defines activity control variables in the ALCS system configuration table generation.

Advanced Communications Function for the Network Control Program (ACF/NCP)
An IBM licensed program that provides communication controller support for single-domain, multiple-domain, and interconnected network capability.

Advanced Program-to-Program Communications (APPC)
A set of inter-program communication services that support cooperative transaction processing in an SNA network. APPC is the implementation, on a given system, of SNA's logical unit type 6.2 (LU 6.2). See APPC component and APPC transaction scheduler.

Aeronautical Radio Incorporated (ARINC)
An organization which provides communication facilities for use within the airline industry.

agent assembly area (AAA)
A fixed-file record used by IPARS applications. One AAA record is associated with each terminal and holds data that needs to be kept beyond the life of an entry. For example, to collect information from more than one message.

agent set
Synonym for communication terminal.

agent set control unit (ASCU)
Synonym for terminal interchange.

Airline Control Program (ACP)
An earlier version of the IBM licensed program Transaction Processing Facility (TPF).

Airline Control System (ALCS)
A transaction processing platform providing high performance, capacity, and availability, that runs specialized (typically airline) transaction processing applications.

Airline Control System/Multiple Virtual Storage/Extended Architecture (ALCS/MVS/XA)
An ALCS release designed to run under an MVS/XA operating system.

Airline Control System Version 2 (ALCS V2)
An ALCS release designed to run under a z/OS operating system.

Airline Control System/Virtual Storage Extended (ALCS/VSE)
An ALCS release designed to run under a VSE/AF operating system.

airlines line control (ALC)
A communication protocol particularly used by airlines.

Airlines Line Control Interconnection (ALCI)
A feature of Network Control Program (NCP) that allows it to manage ALC networks in conjunction with a request for price quotation (RPQ) scanner for the IBM 3745 communication controller.

Airline X.25 (AX.25)
A discipline conforming to the ATA/IATA AX.25 specification in the ATA/IATA publication *ATA/IATA Interline Communications Manual*, ATA/IATA document DOC.GEN 1840. AX.25 is based on X.25 and is intended for connecting airline computer systems to SITA or ARINC networks.

ALCS command
A command addressed to the ALCS system. All ALCS commands start with the letter Z (they are also called "Z messages") and are 5 characters long.

These commands allow the operator to monitor and control ALCS. Many of them can only be entered from CRAS terminals. ALCS commands are called "functional messages" in TPF.

ALCS data collection file
A series of sequential data sets to which ALCS writes performance-related data for subsequent processing by the statistical report generator or other utility program. See also data collection and statistical report generator.

ALCS diagnostic file
A series of sequential data sets to which the ALCS monitor writes all types of diagnostic data for subsequent processing by the diagnostic file processor.

ALCS diagnostic file processor
An offline utility, often called the "post processor", that reads the ALCS diagnostic file and formats and prints the dump, trace, and system test vehicle (STV) data that it contains.

ALCS entry dispatcher
The ALCS online monitor's main work scheduler. Often called the "CPU loop".

ALCS offline program
An ALCS program that runs as a separate MVS job (not under the control of the ALCS online monitor).

ALCS online monitor
The part of ALCS that performs the services for the ECB-controlled programs and controls their actions.

ALCS trace facility
An online facility that monitors the execution of application programs. When it meets a selected monitor-request macro, it interrupts processing and sends selected data to an ALCS display terminal, to the ALCS diagnostic file, or to the system macro trace block. See also instruction step.

The ALCS trace facility also controls tracing to the MVS generalized trace facility (GTF), for selected VTAM communication activity, and controls tracing of input and output messages to a (wrap around) online trace area for selected communication resources.

ALCS update log file
A series of sequential data sets in which the ALCS monitor records changes to the real-time database.

ALCS user file
A series of sequential data sets to which you may write all types of diagnostic data for subsequent processing by an offline processor. You write the data from an installation-wide monitor exit using the callable service UWSEQ.

allocatable pool
The ALCS record class that includes all records on the real-time database. Within this class, there is one record type for each DASD record size.

The allocatable pool class is special in that ALCS itself can dispense allocatable pool records and use them for other real-time database record classes. For example, all fixed-file records are also allocatable pool records (they have a special status of "in use for fixed file").

When ALCS is using type 2 long-term pool dispense, ALCS satisfies requests for long-term pool by dispensing available allocatable pool records.

See DASD record, real-time database, record class, and record type.

alternate CRAS
A computer room agent set (CRAS) that is not Prime CRAS or receive only CRAS. See computer room agent set, Prime CRAS, and receive only CRAS.

alternate CRAS printer
A CRAS printer that is not receive only CRAS. See CRAS printer and receive only CRAS.

answerback
A positive acknowledgement (ACK) from an ALC printer.

APPC component
The component of MVS that is responsible for extending LU 6.2 and SAA CPI Communications services to applications running in any MVS address space. Includes APPC conversations and scheduling services.

APPC transaction scheduler
A program such as ALCS that is responsible for scheduling incoming work requests from cooperative transaction programs.

application plan
See DB2 application plan.

application
A group of associated application programs that carry out a specific function.

application global area
An area of storage in the ALCS address space containing application data that any entry can access.

The application global area is subdivided into keypointable and nonkeypointable records. Keypointable records are written to the database after an update; nonkeypointable records either never change, or are reinitialized when ALCS restarts.

C programs refer to global records and global fields within the application global area.

application program
A program that runs under the control of ALCS. See also ECB-controlled program.

application program load module
In ALCS, a load module that contains one or more application programs.

application queue
In message queuing with ALCS, any queue on which application programs put and get messages using MQI calls.

assign
Allocate a general sequential file to an entry. The TOPNC monitor-request macro (or equivalent C function) opens and allocates a general sequential file. The TASNC monitor-request macro (or equivalent C function) allocates a general sequential file that is already open but not assigned to an entry (it is reserved).

associated resource
Some ALCS commands generate output to a printer (for example, ZDCOM prints information about a communication resource). For this type of command the printed output goes to the associated resource; that is, to a printer associated with the originating display. There is also a response to the originating display that includes information identifying the associated resource.

asynchronous trace
One mode of operation of the ALCS trace facility. Asynchronous trace is a conversational trace facility to interactively trace entries that do not originate from a specific terminal.

automatic storage block
A storage block that is attached to an entry, but is not attached at a storage level. An assembler program can use the ALASC monitor-request macro to obtain an automatic storage block and BACKC monitor-request macro to release it. C programs cannot obtain automatic storage blocks.

B

backward chain
The fourth fullword of a record stored on the ALCS database, part of the record header. See chaining of records.

When standard backward chaining is used, this field contains the file address of the previous record in the chain, except that the first record contains the file address of the last record in the chain. (If there is only one record, the backward chain field contains zeros.)

balanced path
A path where no single component (channel, DASD director or control unit, head of string, and internal path to the DASD device) is utilized beyond the limits appropriate to the required performance.

bar
In the MVS 64-bit address space, a virtual line called the bar marks the 2-gigabyte address. The bar separates storage below the 2-gigabyte address, called **below the bar**, from storage above the 2-gigabyte address, called **above the bar**.

BATAP
Type B application-to-application program

Binary Synchronous Communication (BSC)
A form of telecommunication line control that uses a standard set of transmission control characters and control character sequences, for binary synchronous transmission of binary-coded data between stations.

bind
See DB2 bind

BIND
In SNA, a request to activate a session between two logical units (LUs). The BIND request is sent from a primary LU to a secondary LU. The secondary LU uses the BIND parameters to help determine whether it will respond positively or negatively to the BIND request.

binder
The program that replaces the linkage editor and batch loader programs that were provided with earlier versions of MVS.

BIND image
In SNA, the set of fields in a BIND request that contain the session parameters.

block
See storage block.

C

catastrophic
A type of system error that results in the termination of ALCS.

chain-chase
See Recoup.

chaining of records
One record can contain the file address of another (usually a pool-file record). The addressed record is said to be chained from the previous record. Chains of records can contain many pool-file records. See forward chain and backward chain.

class
See record class.

clearly differentiated programming interfaces (CDPI)
A set of guidelines for developing and documenting product interfaces so that there is clear differentiation between interfaces intended for general programming use (GUPIs) and those intended for other specialized tasks.

close
Close a sequential file data set (MVS CLOSE macro) and deallocate it from ALCS. For general sequential files this is a function of the TCLSC monitor-request macro (or equivalent C function). ALCS automatically closes other sequential files at end-of-job.

command
See ALCS command.

command list (CLIST)
A sequential list of commands, control statements, or both, that is assigned a name. When the name is invoked the commands in the list are executed.

commit
An operation that terminates a unit of recovery. Data that was changed is now consistent.

common entry point (CEP)
A function in the Transaction Processing Facility Database Facility (TPFDF) product that provides common processing for all TPFDF macro calls issued by ALCS application programs. It also provides trace facilities for TPFDF macro calls.

Common Programming Interface - Communications (CPI-C)
The communication element of IBM Systems Application Architecture (SAA). CPI-C provides a programming interface that allows program-to-program communication using the IBM SNA logical unit 6.2.

Common User Access
Guidelines for the dialog between a user and a workstation or terminal.

communication management configuration (CMC)
A technique for configuring a network that allows for the consolidation of many network management functions for the entire network in a single host processor.

communication resource
A communication network component that has been defined to ALCS. These include each terminal on the network and other network components that ALCS controls directly (for example, SLC links). Resources can include, for example:

SNA LUs (including LU 6.1 links)
ALC terminals
SLC and WTTY links
Applications.

communication resource identifier (CRI)
A 3-byte field that uniquely identifies an ALCS communication resource. It is equivalent to the LN/IA/TA in TPF and the LN/ARID in ALCS/VSE. ALCS generates a CRI for each resource.

communication resource name (CRN)
A 1- to 8-character name that uniquely identifies an ALCS communication resource. For SNA LUs, it is the LU name. The system programmer defines the CRN for each resource in the ALCS communication generation.

communication resource ordinal
A unique number that ALCS associates with each communication resource. An installation can use the communication resource ordinal as a record ordinal for a particular fixed-file record type. This uniquely associates each communication resource with a single record.

For example, IPARS defines a fixed-file record type (#WAARI) for AAA records. Each communication resource has its own AAA record - the #WAARI record ordinal is the communication resource ordinal. See also record ordinal and agent assembly area.

compiler
A program that translates instructions written in a high level programming language into machine language.

computer room agent set (CRAS)
An ALCS terminal that is authorized for the entry of restricted ALCS commands.

Prime CRAS is the primary terminal that controls the ALCS system. Receive Only CRAS (RO CRAS) is a designated printer or NetView operator identifier to which certain messages about system function and progress are sent.

configuration data set
(1) A data set that contains configuration data for ALCS. See also configuration-dependent table .

(2) The ALCS record class that includes all records on the configuration data set. There is only one record type for this class. See record class and record type.

configuration-dependent table
A table, constructed by the ALCS generation process, which contains configuration-dependent data. Configuration-dependent tables are constructed as conventional MVS load modules. In ALCS V2, there are separate configuration-dependent tables for:

System data
DASD data
Sequential file data
Communication data
Application program data.

See also configuration data set.

control byte
The fourth byte of a record stored on the ALCS database, part of the record header. ALCS ignores this byte; some applications, however, make use of it.

control interval (CI)
A fixed-length area of direct access storage in which VSAM stores records. The control interval is the unit of information that VSAM transmits to or from direct access storage.

control transfer
The process that the ALCS online monitor uses to create a new entry and to transfer control to an ECB-controlled program.

conversation_ID:
An 8-byte identifier, used in Get_Conversation calls, that uniquely identifies a conversation. APPC/MVS returns a conversation_ID on the CMINIT, ATBALLOC, and ATBGETC calls; a conversation_ID is required as input on subsequent APPC/MVS calls.

CPU loop
See ALCS entry dispatcher.

CRAS printer
A computer room agent set (CRAS) that is a printer terminal. See computer room agent set.

CRAS display
A computer room agent set (CRAS) that is a display terminal. See computer room agent set.

CRAS fallback
The automatic process that occurs when the Prime CRAS or receive only CRAS becomes unusable by which an alternate CRAS becomes Prime CRAS or receive only CRAS. See also Prime CRAS, receive only CRAS, and alternate CRAS.

create service
An ALCS service that enables an ALCS application program to create new entries for asynchronous processing. The new ECBs compete for system resources and, once created, are not dependent or connected in any way with the creating ECB.

cycling the system
The ALCS system can be run in one of four different system states. Altering the system state is called cycling the system. See SLC link for another use of the term "cycling".

D

DASD record
A record stored on a direct access storage device (DASD). ALCS allows the same range of sizes for DASD records as it allows for storage blocks, except no size L0 DASD records exist.

data collection
An online function that collects data about selected activity in the system and sends it to the ALCS data collection file, if there is one, or to the ALCS diagnostic file. See also statistical report generator.

database request module (DBRM)
A data set member created by the DB2 precompiler that contains information about SQL statements. DBRMs are used in the DB2 bind process. See DB2 bind.

data-collection area
An ECB area used by the ALCS online monitor for accumulating statistics about an entry.

data event control block (DECB)
An ALCS control block, that may be acquired dynamically by an entry to provide a storage level and data level in addition to the 16 ECB levels. It is part of entry storage.

The ALCS DECB is independent of the MVS control block with the same name.

Data Facility Storage Management Subsystem (DFSMS)
An MVS operating environment that helps automate and centralize the management of storage. It provides the storage administrator with control over data class, management class, storage group, and automatic class selection routine definitions.

Data Facility Sort (DFSORT)
An MVS utility that manages sorting and merging of data.

data file
A sequential data set, created by the system test compiler (STC) or by the `ZDATA DUMP` command, that contains data to be loaded on to the real-time database. (An ALCS command `ZDATA LOAD` can be used to load data from a data file to the real-time database.) A data file created by STC is also called a "pilot" or "pilot tape".

data level
An area in the ECB or a DECB used to hold the file address, and other information about a record. See ECB level and DECB level.

data record information library (DRIL)
A data set used by the system test compiler (STC) to record the formats of data records on the real-time system. DRIL is used when creating data files.

DB2 application plan
The control structure produced during the bind process and used by DB2 to process SQL statements encountered during program execution. See DB2 bind.

DB2 bind
The process by which the output from the DB2 precompiler is converted to a usable control structure called a package or an application plan. During the process, access paths to the data are selected and some authorization checking is performed.

DB2 Call Attach Facility (CAF)
An interface between DB2 and batch address spaces. CAF allows ALCS to access DB2.

DB2 for z/OS
An IBM licensed program that provides relational database services.

DB2 host variable
In an application program, an application variable referenced by embedded SQL statements.

DB2 package
Also called application package. An object containing a set of SQL statements that have been bound statically and that are available for processing. See DB2 bind.

DB2 package list
An ordered list of package names that may be used to extend an application plan.

DECB level
When an application program, running under ALCS, reads a record from a file, it must "own" a storage block in which to put the record. The address of the storage block may be held in an area of a DECB called a storage level.

Similarly, there is an area in a DECB used for holding the 8-byte file address, record ID, and record code check (RCC) of a record being used by an entry. This is a data level.

The storage level and data level in a DECB, used together, are called a DECB level.

See also ECB level.

diagnostic file
See ALCS diagnostic file.

dispatching priority
A number assigned to tasks, used to determine the order in which they use the processing unit in a multitasking situation.

dispense (a pool-file record)
To allocate a long-term or short-term pool-file record to a particular entry. ALCS performs this action when requested by an application program. See release a pool-file record.

double-byte character set
A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS-capable.

duplex
A communication link on which data can be sent and received at the same time. Synonymous with full duplex. Communication in only one direction at a time is called "half-duplex". Contrast with simplex transmission.

duplex database
Synonym for duplicated database.

duplicated database
A database where each data set is a mirrored pair. In ALCS, you can achieve this using either ALCS facilities or DASD controller facilities (such as the IBM 3990 dual copy facility). See mirrored pair.

dynamic program linkage
Program linkage where the connection between the calling and called program is established during the execution of the calling program. In ALCS dynamic program linkage, the connection is established by the ALCS ENTER/BACK services. Contrast with static program linkage.

dynamic SQL
SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution. Contrast with embedded SQL.

E

ECB-controlled program
A program that runs under the control of an entry control block (ECB). These programs can be application programs or programs that are part of ALCS, for example the ALCS programs that process operator commands (Z messages). ECB-controlled programs are known as E-type programs in TPF.

ECB level
When an application program, running under ALCS, reads a record from file, it must "own" a storage block in which to put the record. The address of the storage block may be held in an area of the ECB called a storage level.

There are 16 storage levels in the ECB. A storage block with its address in slot zero in the ECB is said to be attached on level zero.

Similarly, there are 16 areas in the ECB that may be used for holding the 4-byte file addresses, record ID, and record code check (RCC) of records being used by an entry. These are the 16 data levels.

Storage levels and data levels, used together, are called ECB levels.

See also DECB level.

embedded SQL
Also called static SQL. SQL statements that are embedded within an application program and are prepared during the program preparation process before the program is executed. After it is prepared, the statement itself does not change (although values of host variables specified within the statement can change). Contrast with dynamic SQL.

Emulation Program/Virtual Storage (EP/VS)
A component of NCP/VS that ALCS V2 uses to access SLC networks.

ENTER/BACK
The general term for the application program linkage mechanism provided by ALCS.

entry
The basic work scheduling unit of ALCS. An entry is represented by its associated entry control block (ECB). It exists either until a program that is processing that entry issues an EXITC monitor-request macro (or equivalent C function), or until it is purged from the system. An entry is created for each input message, as well as for certain purposes unrelated to transactions. One transaction can therefore generate several entries.

entry control block (ECB)
A control block that represents a single entry during its life in the system.

entry dispatcher
See ALCS entry dispatcher.

entry macro trace block
There is a macro trace block for each entry. Each time an entry executes a monitor-request macro (or a corresponding C function), ALCS records information in the macro trace block for the entry.

This information includes the macro request code, the name of the program that issued the macro, and the displacement in the program. The ALCS diagnostic file processor formats and prints these macro trace blocks in ALCS system error dumps.

See also system macro trace block.

entry storage
The storage associated with an entry. It includes the ECB for the entry, storage blocks that are attached to the ECB or DECBs, storage blocks that are detached from the ECB or DECBs, automatic storage blocks, and DECBs. It also includes heap storage (for high-level language or assembler language programs) and stack storage (for high-level language programs).

equate
Informal term for an assignment instruction in assembler languages.

error index byte (EIB)
See SLC error index byte.

extended buffer
A storage area above 2 GB used for large messages.

extended message format
For input and output messages, a message format which includes a 4-byte field for the message length.

Execute Channel Program (EXCP)
An MVS macro used by ALCS V2 to interface to I/O subsystems for SLC support.

F

fetch access
Access which only involves reading (not writing). Compare with store access.

file address
4-byte (8 hexadecimal digits) value or 8-byte value in 4x4 format (low order 4-bytes contain a 4-byte file address, high order 4 bytes contain hexadecimal zeros) that uniquely identifies an ALCS record on DASD. `FIND/FILE` services use the file address when reading or writing DASD records. See fixed file and pool file.

file address compute routine (FACE)
An ALCS routine, called by a monitor-request macro (or equivalent C function) that calculates the file address of a fixed-file record. The application program provides the FACE routine with the fixed-file record type and the record ordinal number. FACE returns the 4-byte file address.

There is also an FAC8C monitor-request macro (or equivalent C function), that will return an 8-byte file address in 4x4 format.

FIND/FILE
The general term for the DASD I/O services that ALCS provides.

fixed file
An ALCS record class - one of the classes that reside on the real-time database. All fixed-file records are also allocatable pool records (they have a special status of "in use for fixed file").

Within this class there are two record types reserved for use by ALCS itself (#KPTRI and #CPRCR). There can also be installation-defined fixed-file record types.

Each fixed-file record type is analogous to a relative file. Applications access fixed-file records by specifying the fixed-file record type and the record ordinal number. Note however that fixed-file records are not physically organized as relative files (logically adjacent records are not necessarily physically adjacent).

See real-time database, record class, and record type. See also system fixed file. Contrast with pool file.

fixed-file record
One of the two major types of record in the real-time database (the other is a pool-file record). When the number of records of a particular kind will not vary, the system programmer can define a fixed file record type for these records. ALCS application programs accessing fixed-file records use the ENTRC monitor-request macro to invoke the 4-byte file address compute routine (FACE or FACS) or use the FAC8C monitor-request macro to compute an 8-byte file address. The equivalent C functions are `face` or `facs` or `tpf_fac8c`.

fixed-file record type
(Known in TPF as FACE ID.) The symbol, by convention starting with a hash sign (#) [3] which identifies a particular group of fixed-file records. It is called the fixed-file record type symbol. The equated value of this symbol (called the fixed-file record type value) also identifies the fixed-file record type.

forward chain
The third fullword of a record stored on the ALCS database (part of the record header). When standard forward chaining is used, this field contains the file address of the next record in the chain, except that the last (or only) record contains binary zeros.

full-duplex
Deprecated term for duplex.

[3] This character might appear differently on your equipment. It is the character represented by hexadecimal 7B.

functional message
See ALCS command.

G

general data set (GDS)
The same as a general file, but accessed by different macros or C functions in ALCS programs.

general file
(1) A DASD data set (VSAM cluster) that is used to communicate data between offline utility programs and the online system. General files are not part of the real-time database.

(2) The ALCS record class that includes all records on the general files and general data sets. Each general file and general data set is a separate record type within this class. See record class and record type.

general file record
A record on a general file.

generalized trace facility (GTF)
An MVS trace facility. See also ALCS trace facility.

general sequential file
A class of sequential data set that is for input or output. ALCS application programs must have exclusive access to a general sequential file before they can read or write to it. See also real-time sequential file.

general tape
TPF term for a general sequential file.

general-use programming interface (GUPI)
An interface intended for general use in customer-written applications.

get file storage (GFS)
The general term for the pool file dispense mechanisms that ALCS provides.

global area
See application global area.

global resource serialization
The process of controlling access of entries to a global resource so as to protect the integrity of the resource.

H

half-duplex
A communication link that allows transmission in one direction at a time. Contrast with duplex.

halt
(1) The ALCS state when it is terminated.

(2) The action of terminating ALCS.

heap
An area of storage that a compiler uses to satisfy requests for storage from a high-level language (for example, `calloc` or `malloc` C functions). ALCS provides separate heaps for each entry (if needed). The heap is part of entry storage. Assembler language programs may also obtain or release heap storage using the CALOC, MALOC, RALOC, and FREEC monitor-request macros.

High Level Assembler (HLASM)
A functional replacement for Assembler H Version 2. HLASM contains new facilities for improving programmer productivity and simplifying assembler language program development and maintenance.

high-level language (HLL)
A programming language such as C or COBOL.

high-level language (HLL) storage unit
Alternative name for a type 2 storage unit. See storage unit.

high-level network (HLN)
A network that provides transmission services between transaction processing systems (for example, ALCS) and terminals. Strictly, the term "high-level network" applies to a network that connects to transaction processing systems using SLC. But in ALCS publications, this term is also used for a network that connects by using AX.25 or MATIP.

high-level network designator (HLD)
The entry or exit point of a block in a high-level network. For SLC networks, it is the SLC address of a switching center that is part of a high-level network. It comprises two bytes in the 7-bit transmission code used by SLC.

HLN entry address (HEN)
The high-level designator of the switching center where a block enters a high-level network.

HLN exit address (HEX)
The high-level designator of the switching center where a block leaves a high-level network.

hold
A facility that allows multiple entries to share data, and to serialize access to the data. The data can be a database record, or any named data resource. This facility can be used to serialize conflicting processes. See also record hold and resource hold.

host variable
See DB2 host variable

HTTP enabler
Part of the z/OS Client Web Enablement Toolkit which provides RESTful services enabling a z/OS application HTTP client to access Web services.

I

information block
See SLC link data block.

initial storage allocation (ISA)
An area of storage acquired at initial entry to a high-level language program. ALCS provides a separate ISA for each entry (if required). The ISA is part of entry storage.

initiation queue
In message queuing, a local queue on which the queue manager puts trigger messages. You can define an initiation queue to ALCS, in order to start an ALCS application automatically when a trigger message is put on the queue. See trigger message.

input/output control block (IOCB)
A control block that represents an ALCS internal "task". For example, ALCS uses an IOCB to process a DASD I/O request.

input queue
In message queuing with ALCS, you can define a local queue to ALCS in order to start an ALCS application automatically when a message is put on that queue. ALCS expects messages on the input queue to be in PPMSG message format. See PPMSG.

installation-wide exit
The means specifically described in an IBM software product's documentation by which an IBM software product may be modified by a customer's system programmers to change or extend the functions of the IBM software product. Such modifications consist of exit routines written to replace an existing module of an IBM software product, or to add one or more modules or subroutines to an IBM software product for the purpose of modifying (including extending) the functions of the IBM software product. Contrast with user exit.

instruction step
One mode of operation of the ALCS trace facility. Instruction step is a conversational trace facility that stops the traced application program before the execution of each processor instruction.

Integrated Cryptographic Service Facility (ICSF)
A facility on z/OS that provides data encryption and decryption services.

Interactive System Productivity Facility (ISPF)
An IBM licensed program that serves as a full-screen editor and dialog manager. ISPF provides a means of generating standard screen panels and interactive dialog between the application programmer and terminal user.

interchange address (IA)
In ALC, the 1-byte address of a terminal interchange. Different terminal interchanges connected to the same ALC link have different interchange addresses. Different terminal interchanges connected to different ALC links can have the same interchange address. See also terminal interchange

International Programmed Airlines Reservation System (IPARS)
A set of applications for airline use. The principal functions are reservations and message switching.

IPARS for ALCS
The ALCS shipment includes IPARS as a sample application, and installation verification aid for ALCS.

J

JSON Parser
Part of the z/OS Client Web Enablement Toolkit which provides a generic, native z/OS JavaScript Object Notation (JSON) parser for z/OS applications

K

KCN
Abbreviation for an SLC channel number. See SLC channel.

keypointable
See application global area.

keypoint B (CTKB)
A record that contains dynamic system information that ALCS writes to DASD when it is updated so that ALCS can restart from its latest status.

L

Language Environment
A common run-time environment and common run-time services for z/OS high level language compilers.

level
See ECB level.

line number (LN)
(1) In ALC, the 1-byte address of an ALC link. Different links connected to the same communication controller have different line numbers. Different links connected to different communication controllers can have the same line number.

(2) Synonym for symbolic line number.

Link Control -- Airline (LICRA)
The name of a programming request for price quotation (PRPQ) to the IBM 3705 Emulation Program (EP/VS). This modifies EP/VS to support SLC networks.

link control block (LCB)
See SLC link control block.

link data block (LDB)
See SLC link data block.

link trace
See SLC link trace.

local DXCREI index (LDI)
The first byte of a communication resource indicator (CRI).

local queue
In message queuing, a queue that belongs to the local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with remote queue.

lock
A serialization mechanism whereby a resource is restricted for use by the holder of the lock. See also hold.

log
See ALCS update log.

logging
The process of writing copies of altered database records to a sequential file. This is the method used to provide an up-to-date copy of the database should the system fail and the database have to be restored. The database records are logged to the ALCS update log file.

logical end-point identifier (LEID)
In NEF2 and ALCI environments, a 3-byte identifier assigned to an ALC terminal.

logical unit type 6.2 (LU 6.2)
The SNA logical unit type that supports general communication between programs in a distributed processing environment; the SNA logical unit type on which Common Programming Interface - Communications (CPI-C) is built.

log in
TPF term for establishing routing between a terminal and an application.

log on
Establish a session between an SNA terminal and an application such as ALCS. See also routing.

logon mode
In VTAM, a set of predefined session parameters that can be sent in a BIND request. When a set is defined, a logon mode name is associated with the set.

logon mode table
In VTAM, a table containing several predefined session parameter sets, each with its own logon mode name.

long message transmitter (LMT)
A part of the IPARS application that is responsible for blocking and queuing printer messages for output. Also called XLMT.

long-term pool
An ALCS record class - one of the classes that reside on the real-time database. Within this class, there is one record type for each DASD record size. All long-term pool-file records are also allocatable pool records. ALCS application programs can use long-term pool records for long-lived or high-integrity data. See pool file, real-time database, record class, and record type.

L0, L1, L2, L3, ..., L8
Assembler symbols (and defined values in C) for the storage block sizes and record sizes that ALCS supports. See DASD record and storage block size.

M

macro trace block
See entry macro trace block and system macro trace block.

Mapping of Airline Traffic over IP (MATIP)
A protocol for transporting traditional airline messages over an IP (Internet Protocol) network. Internet RFC (Request for Comments) number 2351 describes the MATIP protocol.

MBI exhaustion
The condition of an SLC link when a sender cannot transmit another message because all 7 SLC message labels are already "in use"; that is, the sender must wait for acknowledgement of a message

so that it can reuse the corresponding message label. See also SLC link, SLC message label, and SLC message block indicator.

message
For terminals with an Enter key, an input message is the data that is sent to the host when the Enter key is hit. A response message is the data that is returned to the terminal. WTTY messages have special "start/end of message" character sequences. One or more input and output message pairs make up a transaction.

message block indicator
See SLC message block indicator.

message label
See SLC message label.

Message Queue Interface (MQI)
The programming interface provided by the IBM WebSphere MQ message queue managers. This programming interface allows application programs to access message queuing services.

message queue manager
See queue manager.

message queuing
A programming technique in which each program within an application communicates with the other programs by putting messages on queues. This enables asynchronous communication between processes that may not be simultaneously active, or for which no data link is active. The message queuing service can assure subsequent delivery to the target application.

message switching
An application that routes messages by receiving, storing, and forwarding complete messages. IPARS for ALCS includes a message switching application for messages that conform to ATA/IATA industry standards for interline communication *ATA/IATA Interline Communications Manual,* DOC.GEN/1840.

mirrored pair
Two units that contain the same data and are referred to by the system as one entity.

monitor-request macro
Assembler language macro provided with ALCS, corresponding to TPF "SVC-type" or "control program" macros. Application programs use these macros to request services from the online monitor.

MQ Bridge
The ALCS MQ Bridge allows application programs to send and receive messages using WebSphere MQ for z/OS queues, without the need to code MQ calls in those programs. The MQ Bridge installation-wide monitor exits USRMQB0, USRMQB1, USRMQB2, and USRMQB3 allow you to customize the behaviour of the MQ Bridge to suit your applications.

MQSeries
A previous name for WebSphere MQ.

multibyte character
A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multiblock message
In SLC, a message that is transmitted in more than one link data block. See link data block.

Multiple Virtual Storage/Data Facility Product (MVS/DFP)
An MVS licensed program that isolates applications from storage devices, storage management, and storage device hierarchy management.

Multisystem Networking Facility (MSNF)
An optional feature of VTAM that permits these access methods, together with NCP, to control a multiple-domain network.

N

namelist
In message queuing, a namelist is an object that contains a list of other objects.

native file address
For migration purposes ALCS allows two or more file addresses to refer to the same database or general file record. The file address that ALCS uses internally is called the native file address.

NCP Packet Switching Interface (NPSI)
An IBM licensed program that allows communication with X.25 lines.

NetView
A family of IBM licensed programs for the control of communication networks.

NetView operator identifier (NetView operator ID)
A 1- to 8-character name that identifies a NetView operator.

NetView program
An IBM licensed program used to monitor a network, manage it, and diagnose network problems.

NetView resource
A NetView operator ID which identifies one of the following:

- A NetView operator logged on to a terminal.
- A NetView operator ID automation task. One of these tasks is used by ALCS to route RO CRAS messages to the NetView Status Monitor Log (STATMON).

network control block (NCB)
A special type of message, used for communication between a transaction processing system and a high-level network (HLN). For example, an HLN can use an NCB to transmit information about the network to a transaction processing system.

For a network that connects using SLC, an NCB is an SLC link data block (LDB). Indicators in the LDB differentiate NCBs from other messages.

For a network that connects using AX.25, NCBs are transmitted across a dedicated permanent virtual circuit (PVC).

Network Control Program (NCP)
An IBM licensed program resident in an IBM 37xx Communication Controller that controls attached lines and terminals, performs error recovery, and routes data through the network.

Network Control Program Packet Switching Interface (NPSI)
An IBM licensed program that provides a bridge between X.25 and SNA.

Network Control Program/Virtual Storage (NCP/VS)
An IBM licensed program. ALCS V2 uses the EP/VS component of NCP/VS to access SLC networks.

Network Extension Facility (NEF)
The name of a programming request for price quotation (PRPQ P09021) that allows management of ALC networks by NCP; now largely superseded by ALCI.

Network Terminal Option (NTO)
An IBM licensed program that converts start-stop terminal device communication protocols and commands into SNA and VTAM communication protocols and commands. ALCS uses NTO to support World Trade Teletypewriter (WTTY).

O

object
In message queuing, objects define the attributes of queue managers, queues, process definitions, and namelists.

offline
A function or process that runs independently of the ALCS online monitor. For example, the ALCS diagnostic file processor is an offline function. See also ALCS offline program.

online
A function or process that is part of the ALCS online monitor, or runs under its control. For example, all ALCS commands are online functions. See also ALCS online monitor.

open
Allocate a sequential file data set to ALCS and open it (MVS OPEN macro). For general sequential files this is a function of the TOPNC monitor-request macro (or equivalent C function). ALCS automatically opens other sequential files during restart.

optimized local adapters (OLA) for WebSphere Application Server for z/OS (WAS)
Built-in, high-speed, bi-directional adapters for calls between WebSphere Application Server for z/OS and ALCS in another address space on the same z/OS image. OLA allows ALCS customers to support an efficient integration of newer Java-based applications with ALCS-based applications. A set of callable services can be used by ALCS assembler or C/C++ programs for exchanging data with applications running in WebSphere Application Server for z/OS. For more information on the callable services (with names of the form BBOA1*xxx*) see the IBM Information Center for WebSphere Application Server - Network Deployment (z/OS) and search for BBOA1. You can use the USRWAS1 installation-wide monitor to verify the caller's authority and to identify input and output messages.

operator command
See ALCS command. Can also refer to non-ALCS commands, for example, MVS or VTAM commands.

ordinal
See communication resource ordinal and record ordinal.

P

package
See DB2 package

package list
See DB2 package list

padded ALC
A transmission code that adds one or more bits to the 6-bit airline line control (ALC) transmission code so that each ALC character occupies one character position in a protocol that uses 7- or 8-bit transmission codes. See also airlines line control.

padded SABRE
Synonym for padded ALC.

passenger name record (PNR)
A type of record commonly used in reservation systems. It contains all the recorded information about an individual passenger.

path
The set of components providing a connection between a processor complex and an I/O device. For example, the path for an IBM 3390 DASD volume might include the channel, ESCON Director, 3990 Storage Path, 3390 Device Adapter, and 3390 internal connection. The specific components used in a particular path are dynamic and may change from one I/O request to the next. See balanced path.

pathlength
The number of machine instructions needed to process a message from the time it is received until the reponse is sent to the communication facilities.

performance monitor
An online function that collects performance data and stores it in records on the ALCS real-time database. It can produce online performance reports based on current data and historical data.

pilot
See data file.

pool directory update (PDU)
A facility of TPF that recovers long-term pool file addresses without running Recoup . PDU identifies and makes available all long-term pool-file records that have been released.

pool file
Short-term pool, long-term pool, and allocatable pool. Within each pool file class, there is one record type for each record size; for example, short-term pool includes the record type L1STPOOL (size L1 short-term pool records).

Each pool-file record type contains some records that are in-use and some that are available. There is a dispense function that selects an available record, changes its status to in-use, and returns the file address. Also, there is a release function that takes the file address of an in-use pool-file record and changes the record status to available.

To use a pool-file record, a program must:

1. Request the dispense function. This returns the file address of a record. Note that the record contents are, at this stage, unpredictable.
2. Write the initial record contents, using the file address returned by step “1” on page 257.
3. Save the file address returned by step “1” on page 257.
4. Read and write the record to access and update the information as required. These reads and writes use the file address saved in step “3” on page 257.

   When the information in the record is no longer required, a program must:
5. Delete (clear to zeros) the saved copy of the file address (see step “3” on page 257).
6. Request the release function.

See also record class. Contrast with fixed file.

pool file directory record (PFDR)
The ALCS pool file management routine keeps a directory for each size (L1, L2, ...L8) of short-term pool file records and long-term pool-file records. It keeps these directories in pool file directory records.

pool-file record
ALCS application programs access pool-file records with file addresses similar to those for fixed-file records. To obtain a pool-file record, an application program uses a monitor-request macro (or equivalent C function) that specifies a 2-byte record ID or a pool-file record type.

When the data in a pool-file record is no longer required, the application uses a monitor-request macro (or equivalent C function) to release the record for reuse. See pool file.

pool-file record identifier (record ID)
The record ID of a pool-file record. On get file requests (using the GETFC monitor-request macro or equivalent C function) the program specifies the pool-file record ID. This identifies whether the pool-file record is a short-term or long-term pool-file record and also determines the record size (L1, L2, ...L8). (Coding the 2-byte record IDs, and the corresponding pool-file record sizes and types, is part of the ALCS generation procedure.) See also record ID qualifier.

pool-file record type
Each collection of short-term and long-term pool-file records of a particular record size (identified by the symbols L1, L2, ..., L8) is a different record type. Each pool-file record type has a different name. For short-term pool-file records, this is L*n*STPOOL, where L*n* is the record size symbol. For long-term pool-file records the name is L*n*LTPOOL.

post processor
See ALCS diagnostic file processor.

PPMSG
ALCS program-to-program message format, used by the ALCS message router to send and receive messages on a message routing path to another system. In PPMSG message format, the routing control parameter list (RCPL) precedes the message text.

primary action code
The first character of any input message. The primary action code Z is reserved for ALCS commands. See secondary action code.

Prime CRAS
The primary display terminal, or NetView ID, that controls the ALCS system. See also computer room agent set (CRAS).

process definition object
In message queuing, an object that contains the definition of a message queuing application. For example, a queue manager uses the definition when it works with trigger messages.

product sensitive programming interface (PSPI)
An interface intended for use in customer-written programs for specialized purpose only, such as diagnosing, modifying, monitoring, repairing, tailoring or tuning of ALCS. Programs using this interface may need to be changed in order to run with new product releases or versions, or as a result of service.

program linkage
Mechanism for passing control between separate portions of the application program. See dynamic program linkage and static program linkage.

program nesting level
One of 32 ECB areas used by the ENTER/BACK mechanism for saving return control data.

program-to-program interface
In NetView, a facility that allows user programs to send data to, or receive data from, other user programs. It also allows system and application programs to send alerts to the NetView hardware monitor.

P.1024
A SITA implementation of SLC. See SLC.

P.1124
A SITA implementation of SLC. See SLC.

P.1024A
The SITA implementation of airline line control (ALC).

Q

queue manager
A system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. WebSphere MQ for z/OS is an example of a queue manager.

R

real-time database
The database to which ALCS must have permanent read and write access. As an ALCS generation option, the real-time database can be duplicated in order to minimize the effects of a DASD failure.

real-time sequential file
A sequential data set used only for output. ALCS application programs can write to any real-time sequential file without requiring exclusive access to the data set. See also general sequential file.

real-time tape
TPF term for a real-time sequential file.

receive only (RO)
The function of a communication terminal that can receive but not send data. An example is a printer that does not have a keyboard.

receive only CRAS
A printer terminal (or NetView operator ID) that ALCS uses to direct status messages. Commonly known as RO CRAS.

record
A set of data treated as a unit.

record class
The first (highest) level categorization of ALCS DASD records. ALCS defines the following record classes:

Allocatable pool
Application fixed file
Configuration data set
General file
Long-term pool
Short-term pool
System fixed file.

See also record type and record ordinal.

record code check (RCC)
The third byte of any record stored in the ALCS database. It is part of the record header.

The RCC field is intended to help detect the incorrect chaining of records which have the same record ID. This is particularly useful for passenger name records (PNRs), of which there are often hundreds of thousands. A mismatch in RCC values shows that the chain is broken, probably as a result of an application program releasing a record too soon. (A false match cannot be excluded, but the RCC should give early warning of a chaining problem.)

record header
A standard format for the first 16 bytes of a record stored on the ALCS database. It contains the following fields:

Record ID
Record code check
Control byte
Application program name
Forward chain
Backward chain.

Not all records contain forward chains and backward chains. Some applications extend the record header by including extra fields. TPFDF uses an extended record header.

record hold
A type of hold that applies to DASD records. Applications that update records can use record hold to prevent simultaneous updates. See also resource hold.

record identifier (record ID)
The first two bytes of a record stored on the ALCS database, part of the record header.

The record ID should always be used to indicate the nature of the data in the record. For example, airlines reservations applications conventionally store passenger name records (PNRs) as long-term pool-file records with a record ID of 'PR'.

When application programs read such records, they can (optionally) request ALCS to check that the record ID matches that which the application program expects.

When application programs request ALCS to dispense pool file records, ALCS uses the record ID to select an appropriate long-term or short-term pool-file record of the requested record size (L1, L2,...,L8). See also record ID qualifier.

record ID qualifier
A number 0 through 9 that differentiates between record types that have the same record ID.

For compatibility with previous implementations of the record ID qualifier, ALCS also accepts the character qualifiers P and O. P (primary) is equivalent to 0, and O (overflow) is equivalent to 1.

record ordinal
The relative record number within a record type. See record class and record type.

record size
See DASD record.

record type
The second level categorization of ALCS DASD records. Within any one record class, the records are categorized into one or more record types. See also record type number, record type symbol, record class and record ordinal.

record type number
A number that identifies a record type.

record type symbol
The character string that identifies a fixed-file record type (#*xxxxx*), a long-term pool-file record type (LsLTPOOL), a short-term pool-file record type (LsSTPOOL), or a general file (GF-*nnn*). The value of the record type symbol is the record type number.

Recoup
A real-time database validation routine which runs online in the ALCS system. (Note that, while the Recoup routines of TPF consist of a number of phases, some online and some offline, the ALCS Recoup is a single online phase that runs, without operator intervention, in any system state.)

Recoup reads selected fixed-file records in the database, and then follows up all chains of pool-file records in the database, noting that these records are in use and giving a warning of any that have been corrupted or released. It then updates the pool file directory records (PFDRs) to show the status of all records.

The ALCS pool file dispense procedure identifies records not in a chain (and so apparently available for reuse) that have not been released.

recoup descriptors
These describe the structure of the entire real-time database.

reentrant
The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks. All ALCS application programs must be reentrant.

relational database
A database that is in accordance with the relational model of data. The database is perceived as a set of tables, relationships are represented by values in tables, and data is retrieved by specifying a result table that can be derived from one or more base tables.

release (a pool-file record)
To make available a long-term or short-term pool-file record so that it can be subsequently dispensed. An application program requests the release action. See dispense a pool-file record.

release file storage (RFS)
The general term for the pool-file release mechanisms that ALCS provides.

remote queue
In message queuing, a queue that belongs to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with local queue.

remote terminal trace
One mode of operation of the ALCS trace facility. Remote terminal trace is a conversational trace facility to interactively trace entries from a terminal other than your own.

REpresentational State Transfer (REST)
An architecture which defines how data is represented to a client in a format (using pis) that is convenient for that client to access Web services.
Common message protocols used for this purpose are HTTP, JSON and XML. Applications using the REST pis are said to be RESTful applications.

reservations
An online application which is used to keep track of seat inventories, flight schedules, and other related information. The reservation system is designed to maintain up-to-date data and to respond within seconds or less to inquiries from ticket agents at locations remote from the computing system.

IPARS for ALCS includes a sample reservations application for airlines.

reserve
Unassign a general sequential file from an entry but leave the file open, so that another (or the same) entry can assign it. Application programs can use the TRSVC monitor-request macro (or equivalent C function) to perform this action.

resource
Any facility of a computing system or operating system required by a job or task, and including main storage, input/output devices, processing unit, data sets, and control or processing programs. See also communication resource.

resource entry index (REI)
The second and third bytes of a communication resource identifier (CRI).

resource hold
A type of hold that can apply to any type of resource. Applications can define resources according to their requirements, and identify them to ALCS using a unique name. See also record hold.

RO CRAS
See receive only CRAS.

rollback
An operation that reverses all the changes made during the current unit of recovery. After the operation is complete, a new unit of recovery begins.

routing
The connection between a communication resource connected to ALCS (typically a terminal on an SNA or non-SNA network) and an application (running under ALCS or another system). Also sometimes called "logging in", but this must be distinguished from logging on, which establishes the SNA connection (session) between the terminal and ALCS.

routing control parameter list (RCPL)
A set of information about the origin, destination, and characteristics of a message. With each input message, ALCS provides an RCPL in the ECB. An output message that is sent using the ROUTC (routc) service also has an RCPL associated with it.

S

scroll
To move a display image vertically or horizontally to view data that otherwise cannot be observed within the boundaries of the display screen.

secondary action code
The second character of an ALCS command. (ALCS commands are made up of 5 characters: Z followed by a secondary action code.) See primary action code.

sequential file
A file in which records are processed in the order in which they are entered and stored in the file. See general sequential file and real-time sequential file.

serialization
A service that prevents parallel or interleaved execution of two or more processes by forcing the processes to execute serially.

For example, two programs can read the same data item, apply different updates, and then write the data item. Serialization ensures that the first program to start the process (read the item) completes the process (writes the updated item) before the second program can start the process - the second program applies its update to the data item which already contains the first update. Without serialization, both programs can start the process (read the item) before either completes the process (writes the updated item) - the second write destroys the first update. See also assign, lock, and hold.

Serviceability Level Indicator Processing (SLIP)
An MVS operator command which acts as a problem determination aid.

short-term pool
An ALCS record class - one of the classes that resides on the real-time database. Within this class, there is one record type for each DASD record size. All short-term pool-file records are also allocatable pool records (they have a special status of "in use for short-term pool"). ALCS application programs can use short-term pool records for short-lived low-integrity data. See pool file, real-time database, record class, and record type.

simplex transmission
Data transmission in one direction only. See also duplex and half-duplex.

sine in/out
Those applications that provide different functions to different end users of the same application can require the user to sine in [4] to the specific functions they require. The sine-in message can, for example, include an authorization code.

single-block message
In SLC, a message that is transmitted in one link data block. See link data block.

single-phase commit
A method in which a program can commit updates to a message queue or relational database without coordinating those updates with updates the program has made to resources controlled by another resource manager. Contrast with two-phase commit.

SLC
See synchronous link control.

SLC channel
A duplex telecommunication line using ATA/IATA SLC protocol. There can be from 1 to 7 channels on an SLC link.

SLC error index byte (EIB)
A 1-byte field generated by Line Control - Airline (LICRA) and transferred to ALCS with each incoming link control block and link data block. Certain errors cause LICRA to set on certain bits of the EIB. See also Link Control -- Airline (LICRA).

SLC information block
Synonym for SLC link data block.

SLC link
A processor-to-processor or processor-to-HLN connection. ALCS supports up to 255 SLC links in an SLC network.

An SLC link that is in the process of an open, close, start, or stop function is said to be "cycling".

SLC link control block (LCB)
A 4-byte data item transmitted across an SLC link to control communications over the link. LCBs are used, for example, to confirm that a link data block (LDB) has arrived, to request retransmission of an LDB, and so on.

SLC link data block (LDB)
A data item, transmitted across an SLC link, that contains a message or part of a message. One LDB can contain a maximum of 240 message characters, messages longer than this must be split and transmitted in multiple LDBs. Synonymous with SLC information block.

SLC link trace
A function that provides a record of SLC communication activity. It can either display the information in real time or write it to a diagnostic file for offline processing, or both. Its purpose is like that of an NCP line trace, but for the SLC protocol.

SLC message block indicator (MBI)
A 1-byte field in the SLC link data block that contains the SLC message label and the block number. A multiblock message is transmitted in a sequence of up to 16 link data blocks with block numbers 1, 2, 3, ... 16. See also multiblock message, SLC link data block, and SLC message label.

[4] This spelling is established in the airline industry.

SLC message label
A number in the range 0 through 7, excluding 1. In P.1024, consecutive multiblock messages are assigned SLC message labels in the sequence: 0, 2, 3, ... 6, 7, 0, 2, and so on. In P.1124, single-block messages are (optionally) also included in the sequence. See also P.1024, P.1124 and SLC message block indicator.

SLC transmission status indicator (TSI)
A 1-byte field in the SLC link data block that contains the SLC transmission sequence number. See also SLC transmission sequence number.

SLC transmission sequence number (TSN)
A number in the range 1 through 31. Consecutive SLC link data blocks transmitted in one direction on one SLC channel are assigned TSNs in the sequence: 1, 2, 3, ... 30, 31, 1, 2, and so on. See also SLC link data block, SLC channel, and SLC transmission status indicator.

SLC Type A traffic
See Type A traffic.

SLC Type B traffic
See Type B traffic.

Société Internationale de Télécommunications Aéronautiques (SITA)
An international organization which provides communication facilities for use within the airline industry.

SQL Communication Area (SQLCA)
A structure used to provide an application program with information about the execution of its SQL statements.

SQL Descriptor Area (SQLDA)
A structure that describes input variables, output variables, or the columns of a result table used in the execution of manipulative SQL statements.

stack
An area of storage that a compiler uses to allocate variables defined in a high-level language. ALCS provides separate stacks for each entry (if needed). The stack is part of entry storage.

standard message format
For input and output messages, a message format which includes a 2-byte field for the message length.

standby
The state of ALCS after it has been initialized but before it has been started. Standby is not considered one of the system states.

static program linkage
Program linkage where the connection between the calling and called program is established before the execution of the program. The connection is established by the assembler, compiler, prelinker, or linkage editor. Static program linkage does not invoke ALCS monitor services. See also dynamic program linkage.

static SQL
See embedded SQL.

statistical report generator (SRG)
An offline ALCS utility that is a performance monitoring tool. It takes the data written to the ALCS data collection or diagnostic file processor by the data collection function and produces a variety of reports and bar charts. The SRG is the equivalent of TPF "data reduction".

STATMON
See NetView resource.

storage block
An area of storage that ALCS allocates to an entry. It is part of entry storage. See storage block sizes.

storage block size
ALCS allows storage blocks of up to 9 different sizes. These are identified in programs by the assembler symbols (or defined C values) L0, L1, L2, ..., L8. Installations need not define all these block sizes but usually define at least the following:

Size L0 contains 127 bytes of user data
Size L1 contains 381 bytes of user data
Size L2 contains 1055 bytes of user data
Size L3 contains 4000 bytes of user data
Size L4 contains 4095 bytes of user data.

The system programmer can alter the size in bytes of L1 through L4, and can specify the remaining block sizes.

storage level
An area in the ECB or a DECB used to hold the address and size of a storage block. See ECB level and DECB level.

storage unit
The ALCS storage manager allocates storage in units called storage units. Entry storage is suballocated within storage units; for example, one storage unit can contain an ECB and several storage blocks attached to that ECB.

ALCS uses three types of storage units:

- Prime and overflow storage units for entry storage and heap storage (if an entry storage block can be used). Also called type 1 storage units.
- High-level language storage units for stack storage Also called type 2 storage units.
- Storage units for heap storage (if an entry storage block can not be used) for programs. Also called type 3 storage units.

The size of a storage unit, and the number of each type of storage unit, is defined in the ALCS generation. See entry storage.

store access
Access which only involves writing (not reading). Compare with fetch access.

striping
A file organization in which logically adjacent records are stored on different physical devices. This organization helps to spread accesses across a set of physical devices.

Structured Query Language (SQL)
a standardized language for defining and manipulating data in a relational database.

symbolic line number (SLN)
In TPF, a 1-byte address of an ALC link, derived from the line number but adjusted so that all ALC links connected to the TPF system have a different symbolic line number. See also line number.

Synchronous Data Link Control (SDLC)
A discipline conforming to subsets of the Advanced Data Communication Control Procedures (ADCCP) of the American National Standards Institute (ANSI) and High-level Data Link Control (HDLC) of the International Organization for Standardization, for managing synchronous, code-transparent, serial-by-bit information transfer over a link connection.

Transmission exchanges can be duplex or half-duplex over switched or nonswitched links. The configuration of the link connection can be point-to-point, multipoint, or loop.

Synchronous Link Control (SLC)
A discipline conforming to the ATA/IATA Synchronous Link Control, as described in the ATA/IATA publication *ATA/IATA Interline Communications Manual*, ATA/IATA document DOC.GEN 1840.

syncpoint
An intermediate or end point during processing of a transaction at which the transaction's protected resources are consistent. At a syncpoint, changes to the resources can safely be committed, or they can be backed out to the previous syncpoint.

system error
Error that the ALCS monitor detects. Typically, ALCS takes a dump, called a system error dump, to the ALCS diagnostic file. See also ALCS diagnostic file and ALCS diagnostic file processor. See also system error dump, system error message.

system error dump
(1) A storage dump that ALCS writes to the ALCS diagnostic file when a system error occurs. See also ALCS diagnostic file and system error.

(2) The formatted listing of a storage dump produced by the ALCS diagnostic file processor. See also ALCS diagnostic file processor.

system error message
A message that ALCS sends to receive only CRAS when a system error occurs. See also receive only CRAS and system error.

system error option
A parameter that controls what action ALCS takes when it detects a system error. See also system error.

system fixed file
An ALCS record class - one of the classes that reside on the real-time database. All system fixed-file records are also allocatable pool records (they have a special status of "in use for system fixed file").

System fixed-file records are reserved for use by ALCS iteslf. See real-time database, record class, and record type.

system macro trace block
There is one system macro trace block. Each time an entry issues a monitor-request macro (or equivalent C function), ALCS records information in the system macro trace block.

This information includes the ECB address, the macro request code, the name of the program that issued the macro, and the displacement in the program. The ALCS diagnostic file processor formats and prints the system macro trace block in ALCS system error dumps. See also entry macro trace block.

System Modification Program/Extended (SMP/E)
An IBM licensed program used to install software and software changes on MVS systems. In addition to providing the services of SMP, SMP/E consolidates installation data, allows flexibility in selecting changes to be installed, provides a dialog interface, and supports dynamic allocation of data sets.

Systems Application Architecture (SAA)
A set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications with cross-system consistency.

Systems Network Architecture (SNA)
The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of networks.

system sequential file
A class of sequential data sets used by ALCS itself. Includes the ALCS diagnostic file, the ALCS data collection file, and the ALCS update log file or files.

system state
The ALCS system can run in any of the following system states: IDLE, CRAS, message switching (MESW), and normal (NORM).

Each state represents a different level of availability of application functions. Altering the system state is called "cycling the system". See also standby.

system test compiler (STC)
An offline ALCS utility that compiles data onto data files for loading on to the real-time database. STC also builds test unit tapes (TUTs) for use by the system test vehicle (STV).

system test vehicle (STV)
An online ALCS function that reads input messages from a general sequential file test unit tape (TUT) and simulates terminal input. STV intercepts responses to simulated terminals and writes them to the ALCS diagnostic file.

T

terminal
A device capable of sending or receiving information, or both. In ALCS this can be a display terminal, a printer terminal, or a NetView operator identifier.

terminal address (TA)
In ALC, the 1-byte address of an ALC terminal. Different terminals connected to the same terminal interchange have different terminal addresses. Different terminals connected to different terminal interchanges can have the same terminal address. See also terminal interchange.

terminal circuit identity (TCID)
Synonym for line number.

terminal hold
When an ALCS application receives an input message, it can set terminal hold on for the input terminal. Terminal hold remains on until the application sets it off. The application can reject input from a terminal that has terminal hold set on. Also referred to as AAA hold.

terminal interchange (TI)
In ALC, synonym for terminal control unit.

terminate
(1) To stop the operation of a system or device.

(2) To stop execution of a program.

test unit tape (TUT)
A general sequential file that contains messages for input to the system test vehicle (STV). TUTs are created by the system test compiler (STC).

time available supervisor (TAS)
An ALCS or TPF function that creates and dispatches low priority entries.

time-initiated function
A function initiated after a specific time interval, or at a specific time. In ALCS this is accomplished by using the CRETC monitor-request macro or equivalent C function. See create service.

TP profile
The information required to establish the environment for, and attach, an APPC/MVS transaction program on MVS, in response to an inbound allocate request for the transaction program.

trace facility
See ALCS trace facility, generalized trace facility, and SLC link trace.

transaction
The entirety of a basic activity in an application. A simple transaction can require a single input and output message pair. A more complex transaction (such as making a passenger reservation) requires a series of input and output messages.

Transaction Processing Facility (TPF)
An IBM licensed program with many similarities to ALCS. It runs native on IBM System/370 machines, without any intervening software (such as MVS). TPF supports only applications that conform to the TPF interface. In this book, TPF means Airline Control Program (ACP), as well as all versions of TPF.

Transaction Processing Facility Database Facility (TPFDF)
An IBM licensed program that provides database management facilities for programs that run in an ALCS or TPF environment.

Transaction Processing Facility/Advanced Program to Program Communications (TPF/APPC)
This enables LU 6.2 for TPF.

Transaction Processing Facility/Data Base Reorganization (TPF/DBR)
A program which reorganizes the TPF real-time database.

Transaction Processing Facility/MVS (TPF/MVS)
Alternative name for ALCS V2 .

Transaction program identifier (TP_ID)
A unique 8-character token that APPC/MVS assigns to each instance of a transaction program. When multiple instances of a transaction program are running sumultaneously, they have the same transaction program name, but each has a unique TP_ID.

transaction scheduler name
The name of an APPC/MVS scheduler program. The ALCS transaction scheduler name is ALCSx000, where x is the ALCS system identifier as defined during ALCS generation.

transfer vector
An ALCS application program written in assembler, SabreTalk, or C, can have multiple entry points for dynamic program linkage. These entry points are called transfer vectors. Each transfer vector has a separate program name.

transmission status indicator
See SLC transmission status indicator.

transmission sequence number
See SLC transmission sequence number.

trigger event
In message queuing, an event (such as a message arriving on a queue) that causes a queue manager to create a trigger message on an initiation queue.

trigger message
In message queuing, a message that contains information about the program that a trigger monitor is to start.

trigger monitor
In message queuing, a continuously-running application that serves one or more initiation queues. When a trigger message arrives on an initiation queue, the trigger monitor retrieves the message. When ALCS acts as a trigger monitor, it uses the information in the trigger message to start an ALCS application that serves the queue on which a trigger event occurred.

triggering
In message queuing, a facility that allows a queue manager to start an application automatically when predetermined conditions are met.

TSI exhaustion
The condition of an SLC channel when a sender cannot transmit another SLC link data block (LDB) because the maximum number of unacknowledged LDBs has been reached. The sender must wait for acknowledgement of at least one LDB so that it can transmit further LDBs. See also SLC channel, SLC link data block, SLC transmission sequence number, and SLC transmission status indicator.

two-phase commit
A protocol for the coordination of changes to recoverable resources when more than one resource manager is used by a single transaction. Contrast with single-phase commit.

type
See record type.

Type A traffic
ATA/IATA conversational traffic - that is, high-priority low-integrity traffic transmitted across an SLC or AX.25 link.

Type B application-to-application program (BATAP)
In any system (such as ALCS) that communicates with SITA using AX.25 or MATIP, this is the program which receives and transmits type B messages.

Type B traffic
ATA/IATA conventional traffic - that is, high-integrity, low-priority traffic transmitted across an SLC or AX.25 link or a MATIP TCP/IP connection.

type 1 pool file dispense mechanism
The mechanism used in ALCS prior to V2 Release 1.3 (and still available in subsequent releases) to dispense both short-term and long-term pool-file records.

type 1 storage unit
Prime or overflow storage unit for entry storage and small heap storage. See storage unit.

type 2 pool file dispense mechanisms
The mechanisms available since ALCS V2 Release 1.3 to dispense pool-file records (the mechanisms are different for short-term and long-term pool-file records).

IBM recommends users to migrate to type 2 dispense mechanisms as part of their migration process.

type 2 storage unit
High-level language storage unit for stack storage. See storage unit.

type 3 storage unit
Storage unit for heap storage that is used when an entry storage block cannot satisfy a request. See storage unit.

U

unit of recovery
A recoverable sequence of operations within a single resource manager (such as WebSphere MQ for z/OS or DB2 for z/OS). Compare with unit of work.

unit of work
A recoverable sequence of operations performed by an application between two points of consistency. Compare with unit of recovery.

Universal Communications Test Facility (UCTF)
An application used by SITA for SLC protocol acceptance testing.

update log
See ALCS update log.

user data-collection area
An optional extension to the data-collection area in the ECB. Application programs can use the DCLAC macro to update or read the user data-collection area.

user exit
A point in an IBM-supplied program at which a user exit routine can be given control.

user exit routine
A user-written routine that receives control at predefined user exit points. User exit routines can be written in assembler or a high-level language.

V

version number
In ALCS and TPF, two characters (not necessarily numeric), optionally used to distinguish between different versions of a program. Sometimes also used with other application components such as macro definitions.

virtual file access (VFA)
An ALCS caching facility for reducing DASD I/O. Records are read into a buffer, and subsequent reads of the same record are satisfied from the buffer. Output records are written to the buffer, either to be written to DASD - immediately or at a later time - or to be discarded when they are no longer useful.

virtual SLC link
Used to address an X.25 PVC or TCP/IP resource for transmitting and receiving Type B traffic. Some applications (such as IPARS MESW) address communication resources using a symbolic line number (SLN) instead of a CRI. These applications can address X.25 PVC and TCP/IP resources by converting the unique SLN of a virtual SLC link to the CRI of its associated X.25 PVC or TCP/IP resource.

W

WAS Bridge
The ALCS WAS Bridge allows ALCS application programs to send and receive messages using optimized local adapters (OLA) for WebSphere Application Server for z/OS without the need to code those callable services in ALCS programs. The ALCS WAS Bridge installation-wide monitor exits USRWAS3, USRWAS4, USRWAS5, and USRWAS6 allow you to customize the behaviour of the WAS Bridge to suit your applications.

WebSphere MQ for z/OS
An IBM product that provides message queuing services to systems such as CICS, IMS, ALCS or TSO. Applications request queuing services through MQI.

wide character
A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales. For the z/OS XL C/C++ compiler, the character set is DBCS, and the value is 2 bytes.

workstation trace
One mode of operation of the ALCS trace facility. Workstation trace controls the remote debugger facility. The remote debugger is a source level debugger for C/C++ application programs.

World Trade Teletypewriter (WTTY)
Start-stop telegraph terminals that ALCS supports through Network Terminal Option (NTO).

Z

Z message
See ALCS command.

Bibliography

Note that unless otherwise specified, the publications listed are those for the z/OS platform.

Airline Control System Version 2 Release 4.1

- *Application Programming Guide*, SH19-6948
- *Application Programming Reference - Assembler Language*, SH19-6949
- *Application Programming Reference - C Language*, SH19-6950
- *Concepts and Facilities*, SH19-6953
- *General Information Manual*, GH19-6738
- *Installation and Customization*, SH19-6954
- *Licensed Program Specifications*, GC34-6327
- *Messages and Codes*, SH19-6742
- *Operation and Maintenance*, SH19-6955
- *Program Directory*, GI10-2577
- *ALCS World Wide Web Server User Guide*
- *OCTM User Guide*

MVS

- *Data Areas, Volumes 1 through 5*, GA22-7581 through GA22-7585
- *Diagnosis Reference*, GA22-7588
- *Diagnosis Tools and Service Aids*, GA22-7589
- *Initialization and Tuning Guide*, SA22-7591
- *Initialization and Tuning Reference*, SA22-7592
- *Installation Exits*, SA22-7593
- *IPCS Commands*, SA22-7594
- *IPCS User's Guide*, SA22-7596
- *JCL Reference*, SA22-7597
- *JCL User's Guide*, SA22-7598
- *JES2 Initialization and Tuning Guide*, SA22-7532
- *JES2 Initialization and Tuning Reference*, SA22-7533
- *Program Management: User's Guide and Reference*, SA22-7643
- *System Codes*, SA22-7626
- *System Commands*, SA22-7627
- *System Messages, Volumes 1 through 10*, SA22-7631 through SA22-7640

APPC/MVS

- *MVS Planning: APPC/MVS Management*, SA22-7599
- *MVS Programming: Writing Transaction Programs for APPC/MVS*, SA22-7621

DFSMS

- *Access Method Services for Catalogs*, SC26-7394

- *DFSMSdfp Storage Administration Reference*, SC26-7402
- *DFSMSdss Storage Administration Guide*, SC35-0423
- *DFSMSdss Storage Administration Reference*, SC35-0424
- *DFSMShsm Storage Administration Guide*, SC35-0421
- *DFSMShsm Storage Administration Reference*, SC35-0422
- *Introduction*, SC26-7397

RMF

- *RMF Report Analysis*, SC33-7991
- *RMF User's Guide*, SC33-7990

Data Facility Sort (DFSORT)

- *Application Programming Guide*, SC33-4035
- *Messages, Codes and Diagnostic Guide*, SC26-7050

Language Environment

- *Language Environment Concepts Guide*, SA22-7567
- *Language Environment Customization*, SA22-7564
- *Language Environment Debugging Guide*, GA22-7560
- *Language Environment Programming Guide*, SA22-7561
- *Language Environment Programming Reference*, SA22-7562
- *Language Environment Run-Time Messages*, SA22-7566

z/OS XL C/C++

- *Standard C++ Library Reference*, SC09-4949
- *z/OS XL C/C++ Compiler and Run-Time Migration Guide*, GC09-4913
- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ Messages*, GC09-4819
- *z/OS XL C/C++ Programming Guide*, SC09-4765
- *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS XL C/C++ User's Guide*, SC09-4767

COBOL

- *Enterprise COBOL for z/OS and OS/390 Language Reference*, SC27-1408
- *Enterprise COBOL for z/OS and OS/390 Programming Guide*, SC27-1412
- *VisualAge COBOL for OS/390 and VM Language Reference*, SC26-9046
- *VisualAge COBOL for OS/390 and VM Programming Guide*, SC26-9049

PL/I

- *Enterprise PL/I for z/OS and OS/390 Language Reference*, SC27-1460
- *Enterprise PL/I for z/OS and OS/390 Messages and Codes*, SC27-1461
- *Enterprise PL/I for z/OS and OS/390 Programming Guide*, SC27-1457
- *VisualAge PL/I for OS/390 Compile-Time Messages and Codes*, SC26-9478

- *VisualAge PL/I for OS/390 Language Reference*, SC26-9476
- *VisualAge PL/I for OS/390 Programming Guide*, SC26-9473

High Level Assembler

- *Language Reference*, SC26-4940
- *Programmer's Guide*, SC26-4941

CPI-C

- *SAA CPI-C Reference*, SC09-1308

DB2 for z/OS

- *Administration Guide*, SC18-9840
- *Application Programming and SQL Guide*, SC18-9841
- *Codes*, GC18-9843
- *Command Reference*, SC18-9844
- *Installation Guide*, GC18-9846
- *Messages*, GC18-9849
- *SQL Reference*, SC18-9854
- *Utility Guide and Reference*, SC18-9855
- *DB2 for z/OS What's New?*, GC18-9856

ISPF

- *ISPF Dialog Developer's Guide*, SC34-4821
- *ISPF Dialog Tag Language*, SC34-4824
- *ISPF Planning and Customizing*, GC34-4814

WebSphere MQ for z/OS

- *An Introduction to Messaging and Queuing*, GC33-0805
- *Application Programming Guide*, SC34-6595
- *Application Programming Reference*, SC34-6596
- *Command Reference*, SC34-6597
- *Concepts and Planning Guide*, GC34-6582
- *Intercommunication*, SC34-6587
- *Messages and Codes*, GC34-6602
- *MQI Technical Reference*, SC33-0850
- *Problem Determination Guide*, GC34-6600
- *System Administration Guide*, SC34-6585

WebSphere Application Server for z/OS

- IBM Information Center for WebSphere Application Server
- IBM Information Center for WebSphere Application Server - Network Deployment z/OS
- *WebSphere on z/OS - Optimized Local Adapters* (Redpaper), REDP-4550

Tivoli Netview

- *Administration Reference*, SC31-8854
- *Automation Guide*, SC31-8853
- *Installation, Getting Started*, SC31-8872
- *User's Guide*, GC31-8849
- *Security Reference*, SC31-8870

SMP/E

- *Reference*, SA22-7772
- *User's Guide*, SA22-7773

Communications Server IP (TCP/IP)

- *API Guide*, SC31-8788
- *Configuration Guide*, SC31-8775
- *Configuration Reference*, SC31-8776
- *Diagnosis Guide*, SC31-8782
- *Implementation Volume 3: High Availability, Scalability, and Performance*, SG24-7534
- *IP and SNA Codes*, SC31-8791
- *Messages Volume 1*, SC31-8783
- *Messages Volume 2*, SC31-8784
- *Messages Volume 3*, SC31-8785
- *Messages Volume 4*, SC31-8786
- *Migration Guide*, SC31-8773

Web Enablement Toolkit

- *MVS Programming: Callable Services for High-Level Languages*, SA23-1377

TPF

- *Application Programming*, SH31-0132
- *Concepts and Structures*, GH31-0139
- *C/C++ Language Support Users Guide*, SH31-0121
- *General Macros*, SH31-0152
- *Library Guide*, GH31-0146
- *System Macros*, SH31-0151

TPF Database Facility (TPFDF)

- *Database Administration*, SH31-0175
- *General Information Manual*, GH31-0177
- *Installation and Customization*, GH31-0178
- *Programming Concepts and Reference*, SH31-0179
- *Structured Programming Macros*, SH31-0183
- *Utilities*, SH31-0185

TSO/E

- *Administration*, SA22-7780
- *Customization*, SA22-7783
- *System Programming Command Reference*, SA22-7793
- *User's Guide*, SA22-7794

Communications Server SNA (VTAM)

- *IP and SNA Codes*, SC31-8791
- *Messages*, SC31-8790
- *Network Implementation* , SC31-8777
- *Operations*, SC31-8779
- *Programming*, SC31-8829
- *Programmers' LU6.2 Guide*, SC31-8811
- *Programmers' LU6.2 Reference*, SC31-8810
- *Resource Definition Reference*, SC31-8778

Security Server (RACF)

- *RACF General User's Guide*, SA22-7685
- *RACF Messages and Codes*, SA22-7686
- *RACF Security Administrator's Guide*, SA22-7683
- *RACF Callable Services*, SA23-2293
- *z/OS Security Server RACF Data Areas*, GA32-0885

Integrated Cryptographic Service Facility (ICSF)

- *ICSF Administrator's Guide*, SA22-7521
- *ICSF Application Programmer's Guide*, SA22-7522
- *ICSF Messages*, SA22-7523
- *ICSF Overview*, SA22-7519
- *ICSF System Programmer's Guide*, SA22-7520

Other IBM publications

- *IBM 3270 Information Display System: Data Stream Programmer's Reference*, GA23-0059
- *Input/Output Configuration Program User's Guide and Reference*, SB0F-3741
- *NTO Planning, Migration and Resource Definition*, SC30-3347
- *Planning for NetView, NCP, and VTAM*, SC31-7122
- *SNA Formats*, GA27-3136
- *X.25 NPSI Planning and Installation*, SC30-3470
- *z/Architecture Principles of Operation*, SA22-7832

ALCS product kit

- *ALCS PDF Product Kit*, SK3T-6944

SITA publications

- *ACS protocol acceptance tool*, SITA document 032-1/LP-SD-001
- *Communications control procedure for connecting IPARS agent set control unit equipment to a SITA SP*, SITA document P.1024B (PZ.7130.1)
- *P.1X24 automatic testing (with UCTF on DIS)*, SITA document 032-1/LP-SDV-001
- *P.1024 Test Guide*, SITA Document PZ.1885.3
- *Status Control Service Step 2: Automatic Protected Report to ACS*, SITA document 085-2/LP-SD-001
- *Synchronous Link Control Procedure*, SITA Document P.1124

SITA produces a series of books which describe the SITA high level network and its protocols. These may be obtained from:

```
Documentation Section
SITA
112 Avenue Charles de Gaulle
92522 Neuilly sur Seine
France
```

Other non-IBM publications

Systems and Communications Reference Manual (Vols 1-7). This publication is available from the International Air Transport Association (IATA). You can obtain ordering information from the IATA web site <http://www.iata.org/> or contact them directly by telephone at +1(514) 390-6726 or by e-mail at Sales@iata.org.

Index

Special Characters

#pragma map 160

Numerics

3270 screen mapping 30, 226

A

abbreviations, list of 228
acronyms, list of 228
agent set 1
Airlines Control Interconnection (ALCI) xvii
ALCS
    how it processes a message 2
    monitor services xviii
    operating environment 1
    overview 1
    what it provides 1
ALIGN option 162
AMSG input 221
AMSG output 222
answerbacks 216
APPC/MVS
    asynchronous C functions 23
    C function call example 19
    callable services 14
    COBOL call example 20
    conversation services 17
    including calls in application programs 19, 21
application error responses 175
application global area
    serializing access to
        by block-concurrent referencing 45
        for compatibility with loosely coupled systems 49
        using CS and CDS 46
        using global 49
        using resource hold 46
        using SHR and XCL 47
        using SYNCC 49
    sharing access to 45–47
    updating
        by block-concurrent referencing 45
        in a multiprocessing environment 44
        in a multiprogramming environment 44
        in a uniprocessing environment 45, 47
        using CS and CDS 46
        using resource hold 46
        using SHR and XCL 47
application program interfaces 125
application program management 124
application programming languages 124
application programs
    assembler language 126
    assembling 161, 162
    C language 128
application programs *(continued)*
    choosing languages for 130
    high-level languages 129
    language restrictions 125
    logic errors 121
    management of 124
    maximum size 145
    naming conventions 146
    other programming interfaces 124
    performance and design 124
    portability 124
    reentrancy requirements 6
    reentrant 97
    SabreTalk programming language 127
    sharing data between 45
    version numbers 146
applications
    portability of 131
    purchased 130
assembling and link-editing application programs 145
atawait function 22
ATAWAIT macro
    example 22
attac C function 86
automatically-allocated storage 84, 86

B

backout 70
backward chain 38
BEGIN macro
    parameters of 47
BEGIN statement 166
block sizes 7

C

C compiler options 163
C function call example 18
C language 128
C language functions
    global 49
callable services
    developing 133
    example for existing application 138
    example from a C language program 139
    parameters 137
    providing for high-level languages 132
    return codes 136
    serialization considerations 134
    using with existing applications 138
    written in C language 140
calling other functions and programs 142
CALOC 84
CE1CC*n* 80
CE1CR*n* 80
CE1CT*n* 80

CE1FA0 - CE1FAF 79
CE1FX*n* 79
CE1OUT 80
CE1SUD 116
CE1SUG 92, 117
CE1WKA 77
CE1WKB 77
chain chasing 10
chaining 9
choosing languages 130
Client support 33
COBOL 200
COBOL compiler options 164
COBOL example call 18
commit and backout
  services 70
communication
  APPC calls 17
  APPC/MVS C function call 19
  APPC/MVS COBOL call 20
  asynchronous APPC/MVS C functions 23
  ATAWAIT call 22
  CPI-C call in a COBOL program 18
  CPI-C function call 18
  ending the conversation 22
  ZSTAT 17
communication resource identifier (CRI) 12, 80
communication resource name (CRN) 12
communication resources
  how they are identified in ALCS 14
  on other systems 14
communication services 15
compiling 163
compiling and link-editing application programs 145
compiling programs 163
computer room agent set (CRAS) 1
Connection and Request Handles 34
contention 43
control
  loss of 111
control byte 37
conversation
  ending 22
conversational trace 171
converting assembler DSECTs to C language structures 146
converting to C structures 146
CPI-C
  callable services 14
  including calls in application programs 18
CRAS 1
creating entries
  attention 102
  C functions for 103
CREEC (creec) service 101
cremc C function 102
CREMC macro 102
cretc C function 102
CRETC macro 102
CRI, *See* communication resource identifier
CRN, *See* communication resource name
cross-reference facility - DXCXREF 148
CSMI 227
CSMO 227

D

DASD
  file address 36
  input and output 36
  record sizes 7
DASD records
  fixed file 8
  general file 8
  pool file 8
  reading 38, 40
  writing 38, 40
data collection area 84
data event control block, *See* DECB
data in ALCS 36
data levels 79
data levels and storage levels 78
databases
  sharing 171
  test 171
DB2 160
DB2 for z/OS
  bind 161, 162, 168, 169
DCLAC macro 84
DECB
  4x4 format file address 8, 36, 38, 91
  8-byte file address 8, 36, 38, 89, 91
  accessing DASD records 91
  accessing in programs 93
  data level fields 89
  error handling 92
  error indicator 115
  fields 88
  format 88
  functional flow 93
  managing 90
  storage level fields 88
  symbolic name 90
  using storage blocks 91
DECB level 88
DECBC macro 86, 90
DECK option 162, 164
designing databases for use with ALCS 65
detac C function 86, 87
diagnostics 110
direct access files 36
directly-addressable global records 56
dispensing pool-file records 9
double-byte character set (DBCS) 219
double-byte character set (DBCS) messages 216
DSECTs 146
DXCBCLPP 159–161, 165, 166, 168
DXCCDDLD 74
DXCFARIC 74
DXCMRT 74
DXCXREF 148

E

EB0EB DSECT 83
EBCFA*n* 79
EBCID*n* 79
EBCM01 77
EBCM02 77

EBCM03 77
EBCRC*n* 79
EBCSD*n* 116
EBER01 77
EBROUT 80
EBRS01 77
EBSW01 77
EBSW02 77
EBSW03 77
EBXSW0 - EBXSW7 77
ECB
  addressing 83
  C function to access 83
  data level fields 79
  entry origin fields 80
  error indicators 82, 115
  exit intercept information 82
  format 75
  local program work area 75
  reserved areas 83
  routing control parameter list (RCPL) 82
  TCP/IP Listener index number 82
  TCP/IP port number 82
  TCP/IP socket descriptor 82
  user area reserved for system-wide fields 82
  user data collection area 84
  user register save area 80
  work areas 1 and 2 76
ECB-controlled programs 6, 74
ecbptr macro 83
ending sequences 217
ENTER services 166
ENTER/BACK mechanism
  in assembler programs 98
entries
  creating multiple 102
  priority of 102
  serializing 43
entry
  creating 101
entry control block (ECB)
  data levels 79
entry origin fields 80
entry points 158
entry points definition file
  syntax 166
entry storage
  on exit 86
error conditions
  detected by ALCS immediately 110
  detected by application programs 111
  detected later by ALCS 111
error conditions in ALCS 110
error indicators 82, 92, 115
error processing 110
error recovery 122
error responses
  application 175
errors
  application program logic 121
  dealing with 115
  designing recovery routines 123
  multiple 118
errors *(continued)*
  recovery from 121, 122
  sequential file 119
  testing 119
  testing for 117
example code
  note regarding xix
EXEC SQL INCLUDE statement 162
existing applications 124
exit an entry 6

F

FA4X4C macro 36
FAC8C macro 36
face C function 36
FACE macro 36
facs C function 36
FACS macro 36
file address
  4x4 format 8, 36, 91
FINIS macro 127
fixed file
  miscellaneous file records 8
fixed files 67
fixed-file record type 8
fixed-file records 8
flipc C function 86
FLIPC macro 86
FLOAT option 163
forward chain 37
FREEC 84
function names
  long 160

G

general data sets
  offline access 52
general files
  offline access 52
general sequential files
  assigning 53
  reserving 53
  services for 55
  several entries using 53
general-file records 8
getcc C function 85
GETCC macro 85
global area
  attention when using 58
  C functions for using
    global function 61
  directly-addressable global records 56
  directories 56
  general description 56
  indirectly-addressable global records 57
  keypointing global records 57
  logical global records 57
  protected globals 62
  read-only access to records and fields 60
  records and fields 56
  serialization of access 59

global area *(continued)*
serializing access 60
unprotected globals 62
use in C language programs 60
global area fields
locking 49
unlocking 49
updating 49
global area protection
in ALCS 63
in TPF 62
global directories 56
globals
protected 62
unprotected 62
granularity 43

H

heap storage for assembler 84
heap storage for HLL 85
high-level language programs
building program header 166
creating program headers 164
entry points 158
linkage independence in 157
special considerations for 153
high-level languages
getting storage 84
higher-level communication 14
holding a record 42
HTTP 33
HTTP Client calls 34

I

I/O (DASD)
services for 51
I/O services
summary of 113
IDECB DSECT 93
IDECCRW 88, 89
IDECCT0 89
IDECDAD 89
IDECDET 89
IDECDLH 89
IDECFA 89, 91
IDECFX0 90
IDECNAM 88
IDECRCC 89
IDECRID 89
IDECSUD 89, 92, 116
IDECUSR 90
implied-wait services 112
IMSG 220
Including MQI calls in application programs 24
Including OLA calls in application programs 29
indirectly-addressable global records 57
initial storage allocation (ISA) 84
input data 16
input edit program 15
input message
standard format for 215
input message formats 15
input messages
receiving 15
IPARS applications 13

J

JSON Parser calls 35
JSON Parser support 34

K

keypointing global records 57
keys
storage protect 62

L

language restrictions 125
levels
program nesting 98
levta macro 86
levtest C function 86
LIB option 164
libraries 146
library control 146
linking programs
C functions for 99
LIST option 162
load module names 149
load modules
base application 151
choosing what programs to include in 150
how ALCS uses 149
how to update 152
local save stack 77
merging updates 152
naming 148
locks
granularity 43
logical global records 57
long-term pool file 68
long-term pool-file records
recovering 10
LONGNAME option 164
loosely coupled systems 59
loss of control 111, 113

M

MALOC 84
memory files 15
message formats 17, 215, 220
message processing 2
messages 86, 215
miscellaneous file records 8
mixed case input 215
monitor services xviii
MQI
calls with asynchronous notification of completion 25
closing objects 28
example call 25
MQAWAIT call - assembler example 26

multiple entries 102
multiple errors 118
multiple required-wait services 113

N

naming conventions
    programs and transfer vectors 147
nesting levels 98
network extension facility (NEF) xvii
NODYNAM option 164
NOLINK option 162

O

OBJECT option 162, 164
offline access to general files and general data sets 52
OLA 28
OPTIMIZE option 164
Optimized Local Adapters 28
ordinal number 8
organizing data 67
other programming interfaces 124
output edit program 17
output message text
    standard format for 217
output messages 86

P

Parser Instance Handles 35
performance
    transaction response time 43
performance and design 65, 124
pool files 67
pool-file record
    dispensing 9
pool-file records
    chaining 9
    long-term 9
    short-term 9
portability 124, 131
presentation space size 218
printer acknowledgements (answerbacks) 216
priority of created entries 102
program function keys 216
program header
    building 166, 168
    high-level languages 159
program headers
    creating 164
program linkage
    static and dynamic 156
program names 158
program nesting levels 98
program preparation 34, 35, 160
program size 145
program stamp 37
programming language
    functions 132
protect keys 62
protected globals
    accessing 62
protected storage
    application global area 43, 62
protecting data 69
PSW key
    changing 62
    restoring 62

R

RALOC 84
RCPL
    input messages 223
    output messages 224
reading and writing DASD records 38, 40
real-time sequential files 53
record
    standard header 36
record classes 67
record code check 37
record hold 42
record ID 37
record locks 42
Recoup program 10
recovering long-term pool-file records 10
recovery from wait-completion errors 121
reentrancy
    necessity for 97
REENTRANT option 164
reentrant programs 6
register labels
    floating-point 81
    general 80
registers
    general
        saving contents of 81
RELCC macro 85
RENT option 162, 164
required-wait services 113
resource hold 59
resources
    forcing exclusive access to 43
response message
    sending to the originating terminal 16
restricted character sets 218
return codes
    WAS 30
RLD option 162
routing control parameter list (RCPL) 15

S

SabreTalk 127, 159–161
sample applications
    assembler 174
    C language 183
    high-level language 200
    SAVEC macro 88
screen mapping
    calling functions online 226
    general description 226
screen maps
    defining (map description) 226
sending unsolicited messages 17

sequential file errors 119
sequential files
  general 53
  real-time 53
  services for processing 55
  system 53
sequential update technique 71, 72
serialization 41
serialization considerations for callable services 134
services
  that include an implied wait 112
services for DASD processing 49, 51
services for managing DECBs 90
short-term pool file 68
short-term pool-file records 9
SHR parameter, ALCS BEGIN macro 47
single-phase commit 70
special considerations for high-level language programs 153
SQL 161
SQL calls
  coding in application programs 64, 65
SQL communication area (SQLCA) 64
SQL descriptor area (SQLDA) 64
SQL statements 161
SQLCA 64
SQLDA 64
SQLDSECT 64
stack storage 85
standard record header 36
starting sequences 217
static and dynamic program linkage
  deciding which to use 157
  dynamic 156
  static 156
storage
  automatic allocation to entries 86
  protected, *See* protected storage
  unprotected, *See* unprotected storage
storage blocks
  activity control 87
  C functions for handling 87
  detaching and attaching 86
storage levels
  fields 80
  swapping 86
storage protect keys 62
SYNCC macro 49
synchpoint manager
  reasons for non-provision in ALCS 73
SYSPARM option 163
system sequential files 53

T

tasnc C function 110
TASNC macro 110
TCP/IP
  example call 31
test database facility 170
tightly coupled systems 59
tokens
  allocating 48
  allocating resources to 48
  defining 48
TPF compatibility
  note regarding xviii
tpf_decb_create C function 90
tpf_decb_locate C function 90
tpf_decb_release C function 90
tpf_decb_swapblk C function 86
tpf_fa4x4c C function 36
tpf_fac8c C function 36
TPFDF 66, 67, 125
tracing
  conversational 171
  remote debugger for C++ 172
  workstation 172
transaction log 71
transfer of control between programs 99
transfer vector names 158
transfer vectors 127
transferring control between programs 97
transferring control to another program 5
transmission codes 215
TRANV macro 159
TRANV statement 167
two-phase commit 70

U

unprotected storage
  application global area 43, 62
unsolicited messages
  sending 17
updating DASD
  recommanded technique 71, 72
user data collection area 84
using globals - attention 58
using information from previous messages 6
USING(MAP) option 163
utility programs 74

V

version numbers 146, 148
virtual file access (VFA) 12

W

wait mechanism 111
wait-completion errors
  recovery from 121
WAS
  handles 30
  return code 30
WAS OLA
  example call 29
work areas for application programs 76
working storage
  block sizes 7
  obtaining 6
WTTY and type B messages 216, 219

X

XCL parameter, ALCS BEGIN macro 47
XMSG for SLC and AX.25 (input and output) 222

XMSG for WTTY (input and output) 222
XREF option 163

Z

ZSTAT 17

Product Number: 5695-068

SH19-6948-21