GC28-6431-0

Program Product

IBM OS Full American National Standard COBOL Compiler and Library, Version 4, Planning Guide

IBM

GC28-6431-0

Program Product

IBM OS Full American National Standard COBOL Compiler and Library, Version 4, Planning Guide

Program Numbers:

5734-CB2 (Compiler & Library)
5734-LM2 (Library only)

The IBM OS Full American National Standard COBOL Compiler and Library, Version 4, is a Program Product that accepts as input source programs written in OS Full American National Standard COBOL, Version 4. Each of the new features of the Version 4 Compiler is described in a separate chapter of this publication. The features are:

- Symbolic Debugging
- Optimized Object Code
- Teleprocessing
- COBOL Library Management Facility
- Dynamic Subprogram Linkage
- Syntax-Checking Compilation
- String Manipulation

System considerations and a description of the COBOL Object-time Subroutine Library are also included.

The Version 4 Compiler also contains all of the features of previous versions and is compatible with the highest level of American National Standard COBOL, X3.23-1968, as approved by ANSI; American National Standard COBOL is compatible with, and identical to, the proposed international standard of the language, Draft ISO Recommendation No. 1989 -- Information Processing -- Programming Language COBOL. The new COBOL language elements of the Version 4 Compiler are IBM extensions to those standards.

This publication is a planning aid for system planners and analysts, and for COBOL programmers. It is intended for use prior to the availability of the Version 4 Compiler, and will be supplemented with reference documentation when the Version 4 Compiler becomes available.

IBM

First Edition (April 1972)

This edition corresponds to Version 4 of the IBM OS Full American National Standard COBOL Compiler.

Changes are periodically made to the specifications herein; any such changes will be reported in subsequent revisions or Technical Newsletters.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York 10020. Comments become the property of IBM.

The IBM OS Full American National Standard COBOL Compiler and Library, Version 4, is a Program Product that accepts as input source programs written in IBM OS Full American National Standard COBOL, Version 4. The Version 4 Object-time Subroutine Library contains COBOL subroutines which, when required, are combined by the Linkage Editor with the object modules produced by the Version 4 Compiler. Also part of the subroutine library is a set of transient routines that can be dynamically fetched during object program execution. The Version 4 Object-time Subroutine Library is also being made available as a separate Program Product.

This publication describes the new features of the Version 4 Compiler and Library, which are:

Symbolic Debugging
Optimized Object Code
Teleprocessing
String Manipulation
COBOL Library Management Facility
Dynamic Subprogram Linkage
Syntax-Checking Compilation

The Version 4 Compiler also contains all of the features of previous versions of the compiler (Version 1, Version 2, and Version 3). The Version 4 Compiler is compatible with the highest level of American National Standard COBOL, X3.23-1968, as approved by ANSI; American National Standard COBOL is compatible with, and identical to, the proposed international standard of the language, Draft ISO Recommendation No. 1989 -- Information Processing -- Programming Language COBOL. The new COBOL language elements of the Version 4 Compiler are IBM extensions to those standards.

Each of the new features of the Version 4 Compiler is described in a separate chapter of this publication. Detailed planning information about new COBOL language elements and general planning information on other factors is given. System considerations and a description of the Version 4 Object-time Subroutine Library are also included. Note that at object program execution time the Object-time Subroutine Library is required online; if the Library Management Facility is not optioned, the subroutine library is also required at link edit time.

This publication is a planning aid for system planners and analysts, and for COBOL programmers. It is intended for use prior to the availability of the Version 4 Compiler, and will be supplemented with reference documentation when the Version 4 Compiler becomes available.

A knowledge of the basic functions provided by the Operating System is necessary for the understanding of this publication. Such information can be found in the following publication:

IBM System/360 Operating System: Introduction, Order No. GC28-6534

The COBOL programmer who uses this publication must be familiar with the capabilities of the operating system implementation of IBM Full American National Standard COBOL. This implementation is described in the publication:

IBM OS Full American National Standard COBOL, Order No. GC28-6396

Detailed information about the Telecommunications Access Method (TCAM) is contained in the publications:

IBM System/360 Operating System:

TCAM Concepts and Facilities, Order No. GC30-2022

TCAM Programmer's Guide and Reference Manual, Order No. GC30-2024

American National Standard COBOL programs may be used as application programs under all versions of the Customer Information Control System (CICS). CICS is a transaction-oriented, multiapplication data base/data communication interface between a System/360 or System/370 operating system and user-written application programs. Further details are given in the publication:

Customer Information Control System (CICS) General Information Manual, Order No. GH20-1028

ACKNOWLEDGMENT

The following extract from Government Printing Office Form Number 1965-0795689 is presented for the information and guidance of the user:

"Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention 'COBOL' in acknowledgment of the source, but need not quote this entire section.

"COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

"No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

"Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

"The authors and copyright holders of the copyrighted material used herein

> FLOW-MATIC (Trademark of Sperry Rand Corporation), Programming for the UNIVAC (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications."

FIGURES

The IBM OS Full American National Standard COBOL Compiler and Library, Version 4, is a Program Product that operates under the MFT and MVT options of the Operating System. (The Version 4 Object-time Subroutine Library is also being made available as a separate Program Product.)

The Version 4 Compiler and Library contains new features and improvements that give the user more powerful and more flexible programming capabilities:

- Advanced Symbolic Debugging provides faster and easier debugging for the COBOL programmer. At abnormal termination a formatted dump, using COBOL source data-names, is produced. Execution-time dynamic dumps at user-specified points in the Procedure Division can also be obtained. When the symbolic debugging feature is requested, optimized object code is automatically provided.

- Optimized Object Code can be requested, resulting in considerably smaller object programs than are produced without optimization. For COBOL programs that are not I/O bound, execution time is reduced.

- COBOL Teleprocessing (TP) programs can now be written, using IBM extensions to American National Standard COBOL. Such programs are device-independent, and can be created more easily than Assembler TP programs. The source language for such programs is a subset of the CODASYL specifications for COBOL TP language.

- COBOL Library Management Facility allows installations running with multiple COBOL regions/partitions to save considerable main storage by sharing some or all of the COBOL library subroutine modules.

- Dynamic Subprogram Linkage gives the user object-time control of main storage. At object time, COBOL subprograms can be loaded under program control; when such a subprogram is no longer needed, the calling program can free the storage it occupies for other use.

- Syntax-Checking Compilation can be requested to save machine time and money while debugging source syntax errors. When unconditional syntax checking is requested, the source program is scanned for syntax errors and such error messages are generated, but no object code is produced. When conditional syntax checking is requested, a full compilation is produced if no messages or only W-level or C-level messages are generated; if one or more E-level or D-level messages are generated, no object code is produced. Selected test cases have shown that when object code is not generated, compilation time may be reduced significantly.

- String Manipulation, providing for more flexible data manipulation, can now be specified in COBOL. Contiguous data can be separated into multiple logical subfields; two or more separate subfields can be concatenated into a single field.

Each of these features is described in a separate chapter of this publication. (Note that all features contained in previous versions of the compiler are retained in the Version 4 Compiler.) System considerations are discussed in a separate chapter. The Version 4 Object-time Subroutine Library is discussed in Appendix A.

Symbolic Debugging, Optimized Object Code, the COBOL Library Management Facility and Syntax-checking Compilation are all implemented through new control card options.

Teleprocessing and String Manipulation are implemented through new COBOL language elements. These elements are a compatible subset of the specifications for these items as approved by the CODASYL Programming Language Committee.

Dynamic Subprogram Linkage is implemented through new control card options as well as through IBM extensions to American National Standard COBOL.

The new Version 4 COBOL language elements described in this publication are all IBM extensions to American National Standard COBOL, X3.23-1968.

All of the features developed for Version 3 of the IBM American National Standard Full COBOL compiler are included in Version 4. These Version 3 Features are:

- Support of the Time Sharing Option (TSO) -- for more effective program development. Compiler output, including diagnostic messages, etc., can be directed to a terminal; the programmer can use COBOL debugging facilities at the terminal; execution time display output can be directed to the terminal.

- Improvements in Object Code to save main storage:

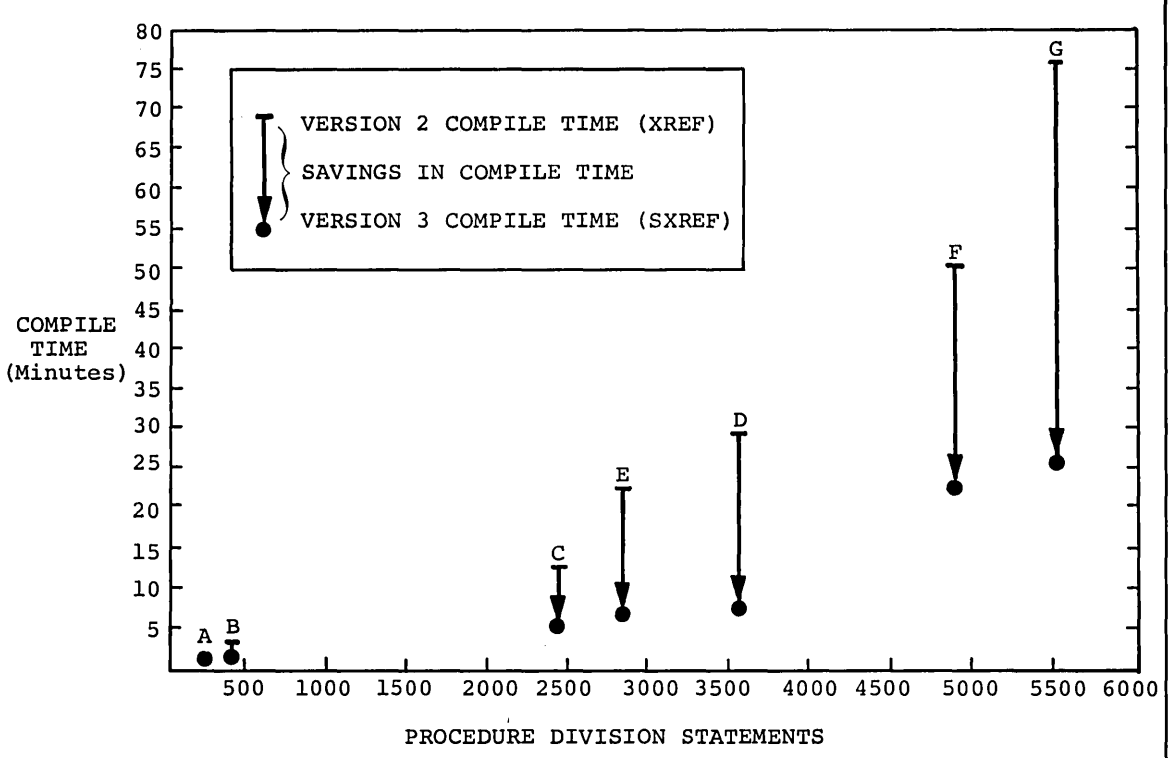
  1. System/370 Support -- can be requested to take advantage of the System/370 instruction set. The System/370 instructions save object program space.

  2. OPEN Statement Improvement -- generated code for the OPEN statement has been modified to give substantial savings in object program space.

  3. Improvements in the MOVE statement and in Comparisons -- when a MOVE statement or a comparison involves a 1-byte literal, generated code for the move and the comparison has been improved.

- Optional Allocation of Compiler Data Sets -- to save compilation time and main storage, SYSLIN, SYSPUNCH, SYSLIB, and SYSTERM are opened by the compiler only if they are specifically requested by the user.

- Alphabetized Cross-Reference Listing -- for easier reference to user-specified names in a program. Performance is much improved over previous source-ordered cross reference listings. Performance of the source-ordered cross-reference has also been significantly improved. Figure 1 shows examples of actual test cases.



Source Program Sizes:

Test Case	Source Cards	Data Division Statements	Procedure Division Statements
A	595	84	261
B	952	199	398
C	4216	824	2386
D	4771	852	3514
E	5991	1669	2814
F	8726	3131	4876
G	10077	3238	5508

Other Options in Effect:

For Version 2: SIZE=120000,BUF=12288,LINECNT=57,SPACE1,FLAGE, SEQ,NOSOURCE,NODMAP,NOPMAP,NOCLIST,NOSUPMAP, NOLOAD,NODECK,APOST,NOTRUNC

For Version 3: All of the above options are in effect, plus the following additional Version 3 options.

NOFLOW,NOTERM,NONUM,NOBATCH,NONAME,COMPILE=01,NOSTATE

System Configuration: MVT1 initiator on Model 50, SYSIN on 2311, separate 2314 SYSUT files

Figure 1. Performance Improvement Using Version 3 SXREF

- Debugging Facilities that are more powerful and flexible:

  1. Flow Trace Option -- a formatted trace can be requested for a variable number of procedures executed before abnormal termination.

  2. Statement Number Option -- provides information about the COBOL statement being executed at abnormal termination.

  3. Expanded CLIST and DMAP -- gives more detailed information about the Data Division and Procedure Division.

  4. Working-Storage Location and Size -- when the CLIST, DMAP, or PMAP options are in effect, the starting address and size of Working-Storage are printed.

- Batch Compilation -- saves machine time by allowing compilation of more than one program with a single invocation of the compiler.

- Separately Signed Numeric Data Type -- for more flexible numeric data description. The sign can be a separate character or an overpunch, and can be leading or trailing.

- Dynamic Record Length Specification -- for more programming flexibility. Specification of record size for QSAM or QISAM files can be deferred until execution time.

- Generic Key for Indexed Files -- use of a partial or full search argument allows more efficient record retrieval.

- RERUN Facility at End-of-Volume -- for sequentially accessed files, checkpoints can be automatically taken at end-of-volume.

- ON Statement Improvement -- count-conditional operands can be identifiers, allowing the programmer to change the conditions of execution of the statement at object time.

- Error Declarative Enhancement -- the GIVING phrase (which requests statistics about the existing error) can now be specified for all options of the Error Declarative.

- Increased I/O Error Diagnostic Facility -- I/O errors not handled by the COBOL programmer are diagnosed by the system and an indicative message is displayed.

- New Sort Features -- COBOL Sort Special Registers have new and improved functions when the IBM OS Version 3 Compiler is used in conjunction with the Program Product Sort/Merge (Program Number 5734-SM1). The COBOL programmer can dynamically specify maximum sort core size, dynamically terminate a sort at an intermediate point, and dynamically specify an alternate destination for sort printer messages.

- ASCII Support -- allows creation and retrieval of tapes written in the American National Standard Code for Information Interchange (ASCII).

More information on these Version 3 features can be found in the Program Product publication IBM OS Full American National Standard COBOL Compiler and Library, Version 3, General Information, Order No. GC28-6407.

Programs written for previous versions of IBM American National Standard COBOL give identical functional results when compiled using the Version 4 Compiler. However, the user must note that:

- The Version 4 Object-time Subroutine Library must be used in conjunction with object programs produced by the Version 4 Compiler; it must always be available (online) at program execution time. It is also being made available as a separate Program Product. (The Version 4 Object-time Subroutine Library is described in Appendix A.)

- Static main storage requirements are decreased when Optimized Object Code is requested.

- Dynamic storage requirements are increased when certain Version 4 Compiler features are used. That is, in a region/partition, storage in addition to that required by the COBOL object module will be needed at object time. The following items must be considered:

  1. Buffers and control blocks for any TCAM or BSAM Teleprocessing files present (see the chapter on Teleprocessing).

  2. COBOL object-time subroutines not loaded into the link pack area/resident reusable routine area (see the chapter on the COBOL Library Management Facility).

  3. Dynamically called user subprograms (see the chapter on Dynamic Subprogram Linkage).

  4. Dynamic storage obtained by certain library subroutines.

- A COBOL source program written for previous versions of the compiler may contain names that appear in the Version 4 reserved word list. (The Version 4 reserved words are listed in Appendix B.) Such reserved words must be changed before the program can be recompiled using the Version 4 Compiler. (New Version 4 reserved words are given a warning message by the Version 3 Compiler.)

This publication is a planning aid only. It is intended for use prior to the availability of the Version 4 Compiler and Library, and will be supplemented by reference documentation when the Compiler and Library become available.

The Version 4 Compiler and Library operates under the control of the MFT or MVT options of the Operating System. For further processing, object modules produced by this compiler require subroutines from the OS Full American National Standard COBOL Object-time Subroutine Library, Version 4 (see Appendix A). At execution time, programs using the SORT statement (with ASCII-encoded files, separately signed numeric data, or that use the 3330 device) require the Program Product OS Sort/Merge Version 5, Program Number 5734-SM1.

Operation of the Version 4 Compiler requires at least 80K bytes of main storage, as did previous versions of the compiler. (For MVT, an 86K region is no longer required.)

The machine configuration remains the same as for previous versions of the compiler, and can be found in the publication:

IBM OS Full American National Standard COBOL Compiler and Library, Version 2, Programmer's Guide, Order No. GC28-6399.

Use of the symbolic debugging feature requires an additional data set, SYSUT5. Either enough direct-access storage space must be available to contain this data set, or an additional tape unit assigned to SYSUT5 must be available to contain it. When the feature is used, this data set must be available at both compile time and execution time. If the feature is specified at compile time, but the user does not wish symbolic debug output at execution time, this additional data set need not be present.

If object programs are executed using the teleprocessing feature of the compiler, the minimum Operating System configuration for TCAM is required. The user must supply a Message Control Program (MCP) in Assembler language using TCAM macro instructions. Further information on writing an MCP can be found in the following publications:

IBM System/360 Operating System:

TCAM Concepts and Facilities, Order No. GC30-2022

TCAM Programmer's Guide and Reference Manual, Order No. GC30-2024

For queued sequential data sets, the RECFM subparameter of the DD statement may optionally be specified at object time, permitting the programmer to specify the standard block option (for data sets with recording mode F) or the track overflow option for the data set. (The track overflow option is equivalent to writing an APPLY RECORD-OVERFLOW clause in the source program.) Use of the standard block option (particularly for direct-access devices having the Rotational Positional Sensing feature) results in significant I/O performance improvement. Fixed-block single volume data sets as created by COBOL are standard (except possibly when extended using the DISP=MOD parameter of the DD statement). Multivolume data sets as created by COBOL are standard if the volume switching occurs through automatic end-of-volume procedures. If, however, the programmer issues a CLOSE REEL/UNIT statement, then he must ensure that the number of logical records in the volume is an integral multiple of n, where a BLOCK CONTAINS n RECORDS clause (or an equivalent BLOCK CONTAINS CHARACTERS clause) has been specified in the source program. The standard block option and the track overflow option are mutually exclusive.

PERFORMANCE CONSIDERATIONS

Optimized object code results in savings in main storage. For COBOL programs that are not I/O-bound, execution time is reduced. Compilation time is slightly increased if the feature is specified.

Use of the COBOL library management facility may result in savings in main storage, secondary storage, and link edit time.

Dynamic invocation and release of COBOL subprograms results in savings in main storage.

Programs using the symbolic debugging feature take longer to compile, link edit, and execute, and require more main storage. The decrease in object-time performance is directly proportional to the number of dynamic symbolic dumps requested and the amount of data produced for each debug request.

Syntax-only compilation saves machine time. Depending on the compiler options chosen, and on source program characteristics, compilation time may be reduced significantly.

In a TSO System, the Version 4 Compiler can most conveniently be invoked using the current release of TSO COBOL Prompter, Program Number 5734-CP1. The symbolic debugging features are available from the TSO terminal when executing in a TSO region. The TSO user can enter symbolic debugging control statements in card image format, and can receive symbolic debugging output at the terminal.

COMPATIBILITY

Programs written for previous versions of the IBM System/360 Operating System Full American National Standard COBOL compiler can be compiled on the Version 4 Compiler without modification, providing that new Version 4 reserved words have not been specified as user-defined names. Object modules produced by the Version 3 Compiler can be link edited with object modules produced by the Version 4 Compiler, and with the Version 4 Object-time Subroutine Library. Object modules produced by the Version 4 Compiler must be executed in conjunction with the Version 4 Object-time Subroutine Library.

Source programs written for the COBOL E and F Compilers must be converted before compiling them on the Version 4 Compiler. The Language Conversion Program described in the following publication facilitates such conversions:

IBM Conversion Aids: COBOL-to-American National Standard COBOL Language Conversion Program, Order No. GC28-6400.

The differences in language and in implementation between COBOL E and COBOL F and American National Standard COBOL are described in the publication:

IBM COBOL Differences; American National Standard COBOL Conversion, Order No. GC28-6395.

Data set compatibility exists going to the Version 4 Compiler from the other IBM Operating System Compilers: previous versions of the Full American National Standard COBOL Compiler, the COBOL E Compiler, and the COBOL F Compiler. That is, data sets created by a program compiled on one of these compilers can be processed by a program compiled on the Version 4 Compiler.

COMPILER OPTIONS

During the scan of compiler options, when an option and its negation (such as XREF and NOXREF) are both specified, the last encountered is the one chosen. This allows the programmer to change one of many options without repunching the entire EXEC job control statement.

After completion of the scan of compiler options, the following actions are performed in the following order. No diagnostic messages are given.

1. If batch compilation is requested, symbolic debugging is negated.
2. If symbolic debugging is requested, optimized object code is also requested, and the statement number option (STATE) is negated.
3. If dynamic subprogram linkage is requested, the COBOL library management facility is also requested.
4. If conditional syntax-checking compilation is requested, then unconditional syntax-checking is negated.
5. If unconditional syntax-checking compilation is requested, then the incompatible compiler options are suppressed; or, if all such options are not requested, then unconditional syntax-checking compilation is requested.

See the chapters on Syntax-checking Compilation, Dynamic Subprogram Linkage, and COBOL Library Management Facility.

SYSTEM/370 DEVICE SUPPORT

All System/370 devices are supported by the Version 4 Compiler; the device for any data set is assigned through the DD statement for that data set. (The device field in the ASSIGN clause system-name is treated as comments.)

Special COBOL programming considerations for the 3505 and 3525 card devices are given in Appendix C.

The symbolic debugging feature produces a symbolic formatted dump of the object program's data area when the job step terminates abnormally. At execution time, the user can also request dynamic dumps similar in content to the abnormal termination dump at any point in the Procedure Division. Therefore, by requesting a dump at a STOP RUN, EXIT PROGRAM, or GOBACK statement, an end-of-job dump can be obtained. If a job step terminates abnormally, then a formatted dump is produced for all COBOL programs compiled with the symbolic debugging feature, which could include the abnormally terminating program and its callers, up to and including the main program.

The abnormal termination dump consists of four parts:

1. Abnormal termination message -- including the program-name, and the COBOL sequence number of the statement and of the verb being executed.

2. An Optional Flow trace -- if requested, a time-sequenced trace of the names of the last "n" COBOL procedures executed.

3. Selected areas in the Task Global Table.

4. Formatted dump of the Data Division including:

   a. For SD's, the sort record

   b. For FD's, the data record

   c. For RD's, the report line, page counter, and line counter

   d. For CD's (Communication Description entries), the CD itself in its implicit format, and the area containing the message data currently being buffered

No source language changes are required for the symbolic debugging feature.

At compile time a new option on the EXEC control card signals that the program currently being compiled is to be executed using symbolic debugging. An additional data set is needed to compile a COBOL source program when symbolic debugging is requested. The fifth data set, SYSUT5, contains the dictionary and statement number information needed to produce the symbolic dump at execution time.

For each COBOL object program link edited together, the user provides a set of control cards requesting a symbolic formatted dump of the object program's data area in case of abnormal step termination. He can also request dynamic dumps at specific points in the Procedure Division. If he requests a dynamic dump when a STOP RUN, EXIT PROGRAM, or GOBACK statement is encountered, an "end-of-job" dump results.

When the symbolic debugging feature is requested, optimized object code is automatically provided.

When the symbolic debugging feature is requested, the user can specify that the sequence numbers of the input program be checked; if cards are out of sequence, the compiler resequences them with an increment of one.

Figure 2 gives portions of a sample program compiled using the symbolic debugging feature, and shows an example of the abnormal termination dump that can be requested.

Performance Considerations

When the symbolic debugging feature is used, load module size is increased by three factors:

1. Space needed for the symbolic debugging routine.
2. Additional inline instructions in the object program for each branch out of the object code main line (such as branches to object-time subroutines, data management routines, etc.).
3. Data and table space, which depends on the number of symbolic debugging control cards specified.

Figure 2. Example of Symbolic Debugging Output

Portions of TESTRUN source program, compiled using the Symbolic Debugging Feature

```
                                        .
                                        .
                                        .
00038   000370  WORKING-STORAGE SECTION.

                                        .
                                        .
                                        .
00055   000530  01  RECORDA.
00056   000535      02 A PIC S9(4) VALUE 1234.
00057   000540      02 B REDEFINES A PIC S9(7) COMPUTATIONAL-3.
                                        .
                                        .
                                        .
00059   000550  PROCEDURE DIVISION.
                                        .
                                        .
                                        .
00066   000620  STEP-1. OPEN OUTPUT FILE-1. MOVE ZERO TO COUNT, NUMBR.
00067   000630  STEP-2. ADD 1 TO COUNT, NUMBR.
00068   000640      MOVE ALPHA (COUNT) TO NO-OF-DEPENDENTS.
00069   000650      COMPUTE B = B + 1.
                                        .
                                        .
                                        .
```

Field B does not contain valid COMPUTATIONAL-3 data.

Therefore, source statement 00069 is in error.

Portions of symbolic formatted dump produced at abnormal termination.

Message giving source statement and verb number causing abnormal termination.

Portion of Data Division dump.

Data present in RECORDA.

Fields A & B. (Invalid numeric positions in field B shown as asterisks (*).

```
                                                        COBOL ABEND DIAGNOSTIC AIDS

 PROGRAM TEST RUN    LAST PSW BEFORE ABEND ...

COMPLETION CODE 0C7

LAST CARD NUMBER/VERB NUMBER EXECUTED -- CARD NUMBER 000069/VERB NUMBER 01.
                                        .
                                        .
                                        .
          000055 01  RECORDA
003E40                                                  (HEX)       F1F2F3C4
003E40    000056 02  A                                      ND-OT   +1234
003E40    000057 02  B                                      NP-S    **1*2*3*
```

ND-OT = numeric display overpunch sign trailing

NP-S = numeric packed decimal—signed

Note: In the complete dump, an explanation of the data codes used, and selected areas of the TGT are printed after the statement number message and before the Data Division dump.

The object code generated by the Version 4 Compiler can be optionally optimized as compared with code generated by previous American National Standard COBOL compilers. If optimization is requested, a program compiled with the Version 4 Compiler results in fewer machine instructions than it would contain if it had been compiled with a previous version of the compiler.

Use of the feature results in a considerable reduction in object program main storage usage; selected test cases have exhibited main storage reductions of up to 33%. The reduction in size is dependent upon source program size and content. In general, the larger the number of source statements, the larger the percentage of reduction. Compilation time is increased. Execution time for COBOL object programs that are not I/O-bound is decreased.

Optimization is requested at compile time through a new parameter in the PARM field of the EXEC job control statement.

Optimized object code is automatically provided when symbolic debugging is specified. Optimized object code may be additionally requested, but is not automatically provided, when the Flow Trace or the Statement Number options are requested.

The Teleprocessing (TP) feature permits the COBOL programmer to create device-independent message processing programs for teleprocessing applications.

A teleprocessing network consists of a central computer, remote (or local) station(s), and the communication lines connecting such station(s) to the central computer.

The central computer consists of the central processing unit (CPU), and the equipment by which the CPU is connected to the communications lines -- such as transmission control units (TCU) and line adapters.

A remote station consists of a control unit and one or more input/output devices. A remote station may be a terminal device, or it may be another computer. With TCAM a program within the same CPU can be considered a "remote station", and can send data to and receive data from another program. (However, in this discussion references to "the computer" are to the central computer.)

Communications lines connect the central computer and the remote stations. The communications lines can be nonswitched or switched.

A nonswitched line links the remote station to the computer either continuously or for regularly recurring periods. Such lines are usually for the exclusive use of one customer. A nonswitched line can connect the central computer either with a single remote station, or with several remote stations.

A switched line is one in which the connection between the central computer and the remote station is established by dialing. As with a telephone system, each remote station and each CPU has a unique number.

In TP applications, data flow into the system is random and proceeds at relatively slow speeds. Data in the system exists as messages from remote stations, or as messages generated by internal programs. Once delivered to the computer, the messages can be processed at computer speeds. Thus, TP applications require a Message Control Program (MCP) that acts as an interface between the COBOL program and the remote stations.

THE MESSAGE CONTROL PROGRAM (MCP)

The Message Control Program (MCP) is a user-written Assembler Language program using TCAM macro instructions. The MCP acts as the logical interface between the COBOL program and the entire network of communications devices, in much the same manner as the system acts as an interface between the COBOL object program and conventional input/output devices. The MCP must also perform device-dependent tasks such as character translation, and insertion of control characters, so that the COBOL program itself is device-independent.

By using Assembler Language TCAM macro instructions, the user can produce an MCP tailored to his application requirements. Information on planning an MCP is given in the publications:

IBM System/360 Operating System:

TCAM Concepts and Facilities, Order No. GC30-2022

TCAM Programmer's Guide and Reference Manual, Order No. GC30-2024

The MCP and the COBOL TP program operate asynchronously; that is, there is no fixed time relationship between the receipt of a message by the MCP and its subsequent processing by the COBOL TP program. COBOL processes messages sequentially, with the MCP storing each message until the complete message is received, and until it can dispose of the message -- either by sending it on to the remote stations, or by transferring it to the COBOL TP program.

To store the messages until they can be processed, the MCP uses destination queues, which are similar to sequential data sets. The queues act as buffers between the COBOL TP program and the remote stations. Thus, the COBOL TP program can accept messages from MCP destination queues and place messages into MCP destination queues as if the queues were sequential files within a conventional COBOL program. To the COBOL program the messages appear to be contiguous; however, they are always under the control of TCAM, and may actually be physically discontinuous, and placed either in core storage and/or disk storage.

TCAM considers all queues as destination queues, with the destination being either a remote terminal or a COBOL TP program. To the COBOL TP program, however, the MCP queue from which it accepts messages is logically an input queue; the queue into which it places messages is logically an output queue. In this publication, these terms are used with this meaning.

Queue Processing

As the interface between the COBOL object program and remote stations, the MCP places messages from remote stations into input queues which the COBOL program may subsequently access. Messages constructed by the COBOL program are placed into output queues by the MCP for later transmission to the remote stations. The names of both types of queues must be defined to the MCP at some time before the execution of the COBOL object program. In the COBOL source program, the user must specify symbolic names for terminal destinations that are known to the MCP. In the MCP, the user controls, through TCAM macro instructions, the input queue to which a message is enqueued.

The names of the input queues as defined to the MCP, and the symbolic names of the message input queues in the COBOL program, need not be the same, since their relationship is defined at object time through data definition (DD) statements.

There is not necessarily a one-to-one relationship between a remote station and a destination. Through the MCP, the user can establish a list facility so that one symbolic name can represent one or many destinations.

The MCP builds message queues on a FEFO (first ended/first out) basis -- that is, if message A begins transmission to the MCP before message B, but message B completes transmission first, then message B will logically be placed first in the queue. Thus, portions of messages

are not logically available in queues until the entire message is available to the MCP. That is, the MCP will not pass a message to a COBOL object program until all of that message is in the queue. For output messages from the COBOL program, the MCP will not transmit a message to a remote station until all of the message is in the output queue.

A message is a string of characters that can be thought of as analogous to a physical record, consisting of one or more logical records, on tape or disk. When the MCP receives a message from a remote station, the MCP program can extensively edit the data through various TCAM macro instructions (including the translation of terminal code to EBCDIC) and direct the message to a destination, which may be a COBOL TP program.

Although a complete message must be present in an MCP queue before it can be transferred, the COBOL program can request that the work unit be transferred either as a complete message, or as a message segment (a logical portion of the message). (A TCAM record is analogous to a COBOL segment.) Each such transfer of message data has an end key associated with it. For output, the end key may be associated with an end indicator. The end key tells the receiving program -- whether the COBOL program or the MCP -- how to treat this group of message characters. There are four end keys and three end indicators:

End Key Code	Associated End Indicator	Meaning
0	none	Uncompleted segment or message.
1	ESI	End of message segment.
2	EMI	End of message.
3	ETI	Logical end of transmission (EOF has been detected -- MCP has issued SETEOF macro).

An end key of 0 (end indicator omitted) informs the receiving program that more data for this work unit (segment and/or message) is still to be transferred. Through the use of end keys, the COBOL program need not provide a Data Division work area large enough to contain the requested work unit, nor need it transfer a complete work unit to the MCP. Thus, the COBOL program can receive work units of unknown length, and construct unusually long messages for transmission.

It is possible that a message may be received by the MCP from a remote station prior to the execution of the COBOL object program. As a result, the MCP enqueues the message in an input queue until the COBOL object program requests dequeueing with a RECEIVE statement. It is also possible that a COBOL object program may cause messages to be enqueued in an output queue which cannot be transferred to the remote stations until after the COBOL object program has terminated (as, for example, when the COBOL object program creates output messages faster than the destination(s) can receive them).

INTERFACE BETWEEN THE COBOL PROGRAM AND THE MCP

The COBOL object program communicates with the MCP when it is necessary to send or to receive data, or to determine the status of the input queues.

In the Data Division, provision for the necessary interface is established through the Communication Section. The COBOL programmer specifies a Communication Description (CD) entry, which describes fixed record areas into which control information is placed.

A CD entry FOR INPUT is valid for messages the COBOL program receives from the MCP; this area specifies the queue from which a message is to be received, as well as other control information (such as source, text length, etc.). The input queue may be made up of sub-queues, in which case the sub-queue structures to be used are completely described in a predefined Queue Structure Description (see "Queue Structure Description and Use"). Once the sub-queue structures have been so defined, then at execution time the COBOL program can access one or more levels of the sub-queues through the SYMBOLIC SUB-QUEUE names of the CD FOR INPUT.

A CD entry FOR OUTPUT is valid for messages the COBOL program sends to the MCP for transmission; the CD entry specifies control information such as the destination(s) of the output message, the text length, etc.

In this way, the input and output CD entries establish an interface between the COBOL program and the MCP.

Three Procedure Division statements control the interface thus established:

- The MESSAGE condition, which causes the MCP to indicate to the COBOL program the number of complete messages in the specified input queue.

- The RECEIVE statement, which causes message data from an input queue (or, if specified, from one or more levels of a sub-queue structure) to be passed to a user-specified work area in the COBOL program.

- The SEND statement, which causes data from the COBOL program to be placed in an output queue for subsequent transmission.

In addition, the STRING statement can be used by the COBOL program to construct output messages from noncontiguous multiple subfields, and the UNSTRING statement can be used to separate a single input data field into multiple logical noncontiguous subfields.

The interfaces between the MCP and the COBOL program, and between the MCP and the remote stations, are shown graphically in Figure 3.

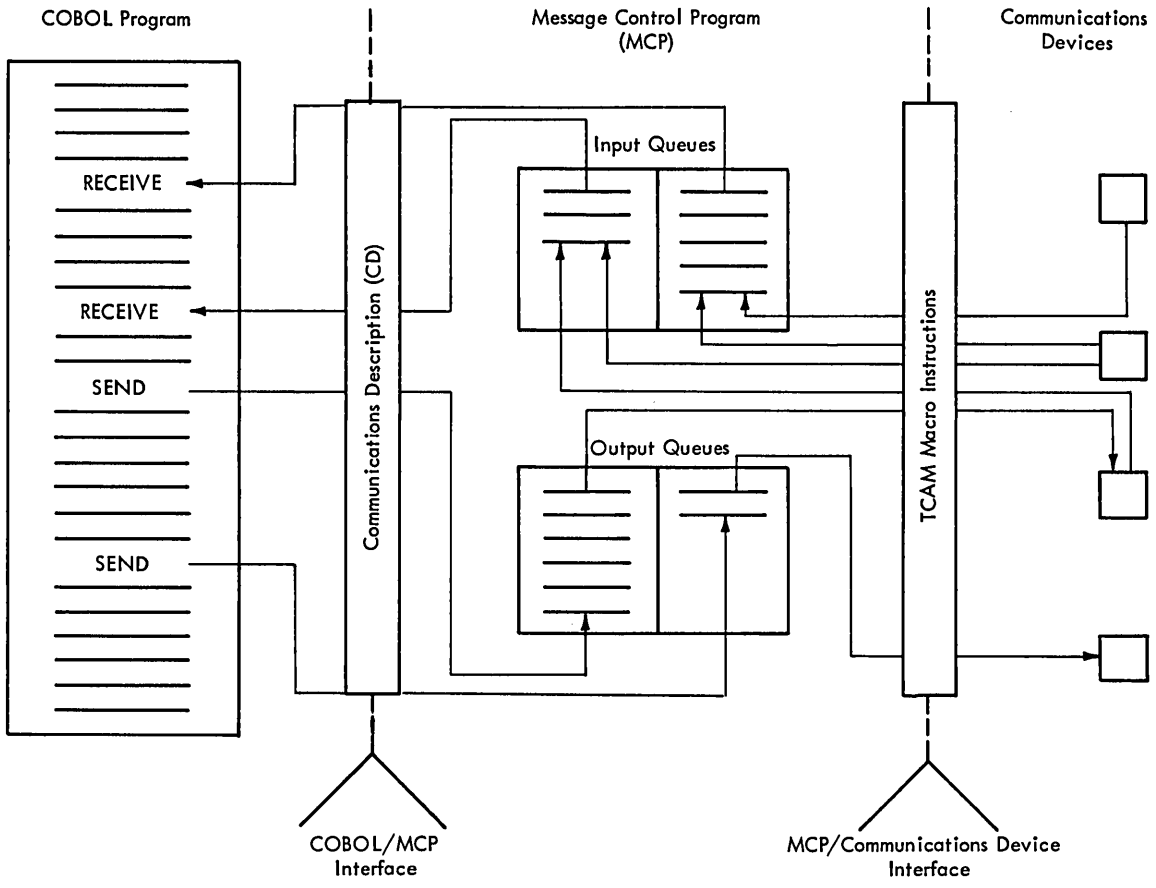


Figure 3. COBOL Communications Environment

COMMUNICATION SECTION

The Communication Section of a COBOL program must be specified if the program is to utilize the TP features of COBOL. The Communication Section, through the definition of Communication Description (CD) entries, establishes the interface between the COBOL object program and the MCP.

When specified, the sections of the Data Division must appear in the following order:

```
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
COMMUNICATION SECTION.
REPORT SECTION.
```

Each of the sections is optional, and, when unnecessary, may be omitted from the source program.

The Communication Section is identified by, and must begin with the section header COMMUNICATION SECTION. The header is followed by Communication Description (CD) entries. Specification of the CD entry causes an implicitly defined data area to be created; that is, the

generated data area has a fixed format. Level-01 record description entries may optionally follow the CD entry; these record description entries implicitly redefine the fixed data areas of the CD.

When it is specified, the Communication Section should contain at least one CD entry. A single CD entry is sufficient if messages are only of one type, that is, only FOR INPUT or only FOR OUTPUT. If the COBOL TP program is to both receive and send messages, then at least two CD entries are required -- one FOR INPUT and one FOR OUTPUT. However, multiple input and/or output CD entries may be specified.

The CD entry is valid only in the Communication Section.

CD Entry

The CD entry represents the highest level of organization in the Communication Section. The Communication Section header is followed by CD entries, each consisting of a level indicator, a data-name, and a series of optional independent clauses.

Format 1

```
CD  cd-name  FOR INPUT

    [[[SYMBOLIC QUEUE           IS data-name-1]
      [SYMBOLIC SUB-QUEUE-1     IS data-name-2]
      [SYMBOLIC SUB-QUEUE-2     IS data-name-3]
      [SYMBOLIC SUB-QUEUE-3     IS data-name-4]
      [MESSAGE DATE             IS data-name-5]
      [MESSAGE TIME             IS data-name-6]
      [SYMBOLIC SOURCE          IS data-name-7]
      [TEXT LENGTH              IS data-name-8]
      [END KEY                  IS data-name-9]
      [STATUS KEY               IS data-name-10]
      [QUEUE DEPTH              IS data-name-11]]

      [data-name-1 data-name-2 ...  data-name-11]].
```

Format 2

```
CD  cd-name  FOR OUTPUT
      [DESTINATION COUNT    IS data-name-1]
      [TEXT LENGTH          IS data-name-2]
      [STATUS KEY           IS data-name-3]
      [ERROR KEY            IS data-name-4]
      [SYMBOLIC DESTINATION IS data-name-5].
```

Format 3

```
CD  cd-name  COPY  library-name

    [REPLACING { word-1       } BY { word-2       }
               { identifier-1 }    { identifier-2 }

      [ { word-3       } BY { word-4       } ] ... ].
        { identifier-3 }    { identifier-4 }
```

The CD entry serves as a storage area through which the COBOL program and the MCP interface. The COBOL programmer moves information about the message into the CD before initiating any request. The MCP, after acting upon the request, returns through the same CD information pertaining to the request.

The CD entry is defined in such a way that any number of message queues may be accessed through the same CD entry. Conversely, different portions of one message may be accessed through multiple CD entries in the same program or in different COBOL subprograms residing in the same region or partition. Thus, any one COBOL TP program need specify only one input CD entry and/or one output CD entry. Rules controlling the accessing of MCP queues are specified in the detailed descriptions of both input (Format 1) and output (Format 2) CD entries.

The level indicator CD identifies the beginning of a Communication Description entry, and must appear in Area A. It must be followed in Area B by cd-name. Cd-name follows the rules for formation of a data-name. Cd-name may be followed by a series of optional independent clauses (as shown in Format 1 and Format 2).

The optional clauses may be followed by an optional level-01 record description entry. This record description entry implicitly redefines that of the fixed data area described by the CD entry. The total length of the record description entry must be the same as or less than the fixed data descriptions of the CD entry; if it is not, an error message is produced. However, the MCP always references this data area according to the implicit data descriptions of the CD entry; that is, for an input CD the contents of positions 1 through 12 are always used as the symbolic queue, the contents of positions 13 through 24 are always used as symbolic sub-queue-1, and so forth.

The optional clauses of the CD entry may be written in any order. Since the data areas of both the input CD and the output CD have implicit definitions, the optional clauses are necessary only to assign user names for those areas that the COBOL program will refer to. However, if all the options of either format are omitted, then a level-01 record description entry must follow the CD entry.

Except for a level-88 entry, the level-01 record description entry must not contain any VALUE clauses.

FORMAT 1: This format is required if the CD entry is FOR INPUT. At least one input CD entry must be specified if input messages are to be received from a queue. Any number of queues may be accessed through the same input CD entry. This is accomplished simply by moving a different symbolic queue name into the input CD. Conversely, different portions of one message may be accessed through different CD entries. Thus, CD

entries in the same or different COBOL subprograms in the same run unit may be used to access different portions of one message. The same CD entry may be used to access a message from another queue before the first message is completed. The following restrictions apply:

- Only one region (or partition) can have access to any particular queue at one time.
- The data in a queue must be accessed sequentially. That is, a second message in any queue cannot be accessed until the entire first message in that queue is accessed. (However, a second message from another queue may be accessed before the entire message in the first queue is accessed.)

The specification of an input CD entry results in a record whose implicit description is equivalent to the following:

Equivalent COBOL Record Description			Description of Use
01	data-name-0.		
02	data-name-1	PICTURE X(12).	Symbolic Queue
02	data-name-2	PICTURE X(12)	Sub-queue-1
02	data-name-3	PICTURE X(12)	Sub-queue-2
02	data-name-4	PICTURE X(12)	Sub-queue-3
02	data-name-5	PICTURE 9(6).	Message Date
02	data-name-6	PICTURE 9(8).	Message Time
02	data-name-7	PICTURE X(12).	Symbolic Source
02	data-name-8	PICTURE 9(4).	Text Length
02	data-name-9	PICTURE X.	End Key
02	data-name-10	PICTURE XX.	Status Key
02	data-name-11	PICTURE 9(6).	Queue Depth

For each input CD entry, a record area of 87 contiguous Standard Data Format characters is always generated, implicitly defined as previously specified.

FORMAT 1 -- OPTION 1: The data names corresponding to the various fields of the CD record area may be explicitly defined, through the use of the optional clauses as follows:

SYMBOLIC QUEUE and SUB-QUEUE Clauses: These clauses define data-name-1, data-name-2, data-name-3, and data-name-4 as the names of alphanumeric data items each of 12 characters in length, and occupying character positions within the record as follows:

data-name-1	occupies	character	positions	1	through	12
data-name-2	"	"	"	13	"	24
data-name-3	"	"	"	25	"	36
data-name-4	"	"	"	37	"	48

The contents of the SYMBOLIC QUEUE can be specified as a queue structure. SUB-QUEUE-1, SUB-QUEUE-2, and SUB-QUEUE-3 specify the levels of such a structure. When a given level of such a structure is specified, all higher levels must also be specified. However, no given queue structure need specify all four levels.

For example, if only a three-level queue structure is needed for a given program, then the following COBOL statements adequately specify the levels of the structure:

```
SYMBOLIC QUEUE IS QNAME
     SYMBOLIC SUB-QUEUE-1 IS SUBQ1
          SYMBOLIC SUB-QUEUE-2 IS SUBQ2 ...
```

Since SYMBOLIC SUB-QUEUE-2 is specified, both SYMBOLIC SUB-QUEUE-1 and SYMBOLIC QUEUE must also be specified. (It would be invalid to specify SUB-QUEUE-2 without also specifying SUB-QUEUE-1.)

When symbolic sub-queues are used in the COBOL program, the associated queue structures must be predefined. See "Queue Structure Description and Use" for a detailed description of the methods used.

A RECEIVE statement causes the serial return of the next message (or portion of a message) from the queue specified in data-name-1, and, if SUB-QUEUE clauses are specified, from one of the sub-queues specified in data-name-2, data-name-3, or data-name-4.

Before the RECEIVE statement is executed, the data-name of the queue, and, if specified, of the sub-queue(s) must contain the symbolic name(s) of the wanted queue. All such symbolic names must be previously defined to the MCP. (See "Queue Structure Description and Use" for the specific method.) The symbolic name must match the queue or sub-queue name previously defined in the queue structure. When any sub-queue name is not being used, its contents must be spaces. The compiler initializes the sub-queues to SPACES; if a sub-queue has been accessed, then it is the responsibility of the user to reinitialize each sub-queue name that is not to be used to SPACES.

When the RECEIVE statement is executed, the MCP uses the symbolic name of the wanted queue to access the next message. After execution of the RECEIVE statement the contents of data-name-1 remain unchanged; the contents of data-name-2 through data-name-4 (if applicable) are updated to contain the name of the sub-queue from which the message was received.

MESSAGE DATE Clause: This clause defines data-name-5 as the name of an unsigned 6-digit integer data item, occupying character positions 49 through 54 of the record.

Data-name-5 has the format YYMMDD (year, month, day). Its contents represent the date on which the MCP received this message.

The contents of data-name-5 are updated by the MCP as part of the execution of each RECEIVE statement.

MESSAGE TIME Clause: This clause defines data-name-6 as the name of an unsigned 8-digit integer data item, occupying character positions 55 through 62 of the record.

Data-name-6 has the format HHMMSSTT (hours, minutes, seconds, hundredths of a second). Its contents represent the time of day the message was received into the system by the MCP.

The contents of data-name-6 are updated by the MCP as part of the execution of each RECEIVE statement.

SYMBOLIC SOURCE Clause: This clause defines data-name-7 as the name of an elementary alphanumeric data item of 12 characters, occupying character positions 63 through 74 of the record.

During execution of a RECEIVE statement, the MCP provides in data-name-7 the symbolic name of the terminal that is the source of this message. (The symbolic names the MCP uses are 1 through 8 characters in length; the remaining characters are set to SPACES.) However, if the symbolic name of the source terminal is not known to the MCP, the contents of data-name-7 are set to SPACES.

TEXT LENGTH Clause: This clause defines data-name-8 as the name of an unsigned 4-digit integer data item, occupying character positions 75 through 78 of the record.

The MCP indicates through the contents of data-name-8 the number of main storage bytes of the user's work area filled as a result of the execution of the RECEIVE statement.

END KEY Clause: This clause defines data-name-9 as the name of a 1-character elementary alphanumeric data item, occupying character position 79 of the record.

The MCP sets the contents of data-name-9, as part of the execution of each RECEIVE statement, according to the following rules:

- When RECEIVE MESSAGE is specified, then the contents of data-name-9 are:

  3 if end-of-transmission has been detected

  2 if end-of-message has been detected

  0 if less than a message has been moved into the user-specified area

- When RECEIVE SEGMENT is specified, then the contents of data-name-9 are:

  3 if end-of-transmission has been detected

  2 if end-of-message has been detected

  1 if end-of-segment has been detected

  0 if less than a message segment has been moved into the user-specified area

- When more than one of the above conditions is satisfied simultaneously, the rule first satisfied in the order listed determines the contents of data-name-9. An End Of Transmission is a logical End Of File condition caused by the user coding a TCAM SETEOF macro in the MCP. In general, depending on the size of the work unit and the work area provided, End Keys may be associated with a text length of zero. Most frequently, this will be the case for End Of Transmission.

STATUS KEY Clause This clause defines data-name-10 as the name of a 2-character elementary alphanumeric data item, occupying character positions 80 and 81 of the record.

The contents of data-name-10 indicate the status condition of the previously executed RECEIVE or IF MESSAGE statement. The program should, therefore, check the STATUS KEY immediately after each RECEIVE operation to verify the status. The values data-name-10 can contain, and their meanings, are defined in Figure 4.

Figure 4 indicates the possible values that the STATUS KEY field (for both input and output CD entries) may contain at the completion of execution for each statement. An X on a line in a statement column indicates that the associated code on that line is possible for that statement.

STATUS KEY Code	Meaning	RECEIVE	SEND	IF MESSAGE
00	No error detected. Request completed.	X	X	X
20	Destination unknown. Data-name-5 gives unknown destination. Request cancelled.		X	
20	1) In use by another partition/region. 2) Queue name unknown (No DD card). 3) Invalid queue structure. Request cancelled.	X		X
21	Insufficent storage available for control blocks and/or buffers. Request cancelled.	X	X	X
22	Queue name unknown (No DD card). Request cancelled.		X	
29	An input/output error has occurred. Request cancelled.	X		X
50	Character count greater than sending field. Request ignored.		X	
60	Partial segment with either zero character count or no sending area specified. Request ignored.		X	

Figure 4. STATUS KEY Field -- Possible Values

QUEUE DEPTH Clause: This clause defines data-name-11 as the name of an unsigned 6-digit integer data item, occupying character positions 82 through 87 of the record.

The contents of data-name-11 indicate the number of messages that exist in an input queue. The MCP updates the contents of data-name-11 only as part of the execution of an IF MESSAGE statement.

FORMAT 1 -- OPTION 2: The second option of Format 1 allows the programmer to specify data-name-1 through data-name-11 without the descriptive clauses. If any data-names are to be omitted, the word FILLER must be substituted for each omitted name, except that FILLER need not be specified for any data-name that comes after the last name referred to.

For example, if the programmer wishes to refer to the SYMBOLIC QUEUE as QUEUE-IN and to the MESSAGE DATE as DATE-IN, he can write the input CD entry as follows:

```
CD  INPUT-AREA  FOR INPUT
    QUEUE-IN FILLER FILLER FILLER DATE-IN.
```

In this case, data-name-6 through data-name-11 can be omitted; FILLER need not be written in their place.

The same input CD entry can be written as follows (In this case, an optional level-01 record description entry redefining the data areas is also included.):

```
CD  INPUT-AREA  FOR INPUT
    SYMBOLIC QUEUE IS QUEUE-IN
    MESSAGE DATE IS DATE-IN.
01  INAREA-RECORD.
    05  FILLER          PICTURE X(78).
    05  ENDKEY-CODE     PICTURE X.
        88  PARTIAL-SEGMENT   VALUE "0".
        88  END-SEGMENT       VALUE "1".
        88  END-MESSAGE       VALUE "2".
        88  END-TRANSMISSION  VALUE "3".
    05  FILLER          PICTURE X(8).
```

By naming the SYMBOLIC QUEUE and MESSAGE DATE fields of the CD the programmer can refer to these data areas within his program without further defining them. By redefining the END KEY data area, the programmer can use condition-names to refer to the values contained in that area.

FORMAT 2: This format is required if the CD entry is FOR OUTPUT. At least one output CD entry must be specified if messages are to be placed into an output queue. A number of output CD entries in the same program or in different subprograms in the same run unit may be used to send different portions of the same message, so that parts of one message may be transferred to the MCP using different CD entries.

Until the transfer of a first message from the COBOL program to the MCP has been completed, the transfer of a second message may not begin. Changing the destination before indicating End Of Message causes unpredictable results.

The specification of an output CD entry always results in a record whose implicit description is equivalent to the following:

Equivalent COBOL Record Description			Description of Use
01	data-name-0.		
02	data-name-1	PICTURE 9(4).	Destination Count
02	data-name-2	PICTURE 9(4).	Text Length
02	data-name-3	PICTURE XX.	Status Key
02	data-name-4	PICTURE X.	Error Key
02	data-name-5	PICTURE X(12).	Symbolic Destination

For each output CD entry, a record area of 23 contiguous Standard Data Format character positions is always generated. It is implicitly defined as previously illustrated. Through the use of the optional clauses, user data-names may be explicitly associated with the output CD subfields as follows:

DESTINATION COUNT Clause: The DESTINATION COUNT clause defines data-name-1 as the name of an unsigned 4-digit integer data item, occupying character positions 1 through 4 of the record. The CODASYL

specification for teleprocessing defines the DESTINATION COUNT clause as shown in Format 2. However, since COBOL allows only one destination, the DESTINATION COUNT clause, if specified, is treated as comments.

TEXT LENGTH Clause: This clause defines data-name-2 as the name of an unsigned 4-digit integer data item, occupying character positions 5 through 8 of the record.

As part of the execution of a SEND statement, the MCP interprets the contents of data-name-2 as the user's indication of the number of leftmost bytes of main storage of the identifier named in the SEND statement to be transferred (see SEND statement).

STATUS KEY Clause: This clause defines data-name-3 as the name of a 2-character elementary alphanumeric data item, occupying character positions 9 and 10 of the record.

The contents of data-name-3 indicate the status condition of the previously executed SEND statement. The values data-name-3 can contain, and their meanings, are defined in Figure 4.

ERROR KEY Clause This clause defines data-name-4 as the name of a 1-character elementary alphanumeric data item, occupying character position 11 of the record.

If, during the execution of a SEND statement, the MCP determines that the specified destination is unknown, the MCP updates the contents of data-name-4. Data-name-4 will contain:

1 if the symbolic destination contained in data-name-4 is unknown to the MCP.

0 if the symbolic destination is known to the MCP.

Note: The ERROR KEY field is updated only when the destination is unknown (that is, when the STATUS KEY is anything other than zero). Therefore, the programmer should not examine the ERROR KEY unless the STATUS KEY field contains a nonzero value.

SYMBOLIC DESTINATION Clause: This clause defines data-name-5 as the name of a 12-character elementary alphanumeric data item, occupying character positions 12 through 23 of the record.

Data-name-5 contains a symbolic destination. The first 1 through 8 characters of data-name-5 must be previously defined to the MCP.

The following example illustrates an output CD entry, with an optional level-01 record description entry redefining the data areas:

```
CD  OUTPUT-AREA FOR OUTPUT
    TEXT LENGTH IS MSG-LGTH
    SYMBOLIC DESTINATION IS Q-OUT.
01  OUTAREA-RECORD.
    05  FILLER          PICTURE X(10).
    05  ERRKEY-CODE     PICTURE X.
        88  KNOWN         VALUE "0".
        88  UNKNOWN       VALUE "1".
    05  FILLER          PICTURE X(12).
```

By naming the TEXT LENGTH and SYMBOLIC DESTINATION fields of the CD entry, the programmer can refer to those data areas within his program without further defining them. By redefining the ERROR KEY data area, the programmer can use condition-names to refer to the values contained in that area.

Note: When a message is being sent to a remote station, TCAM adds the proper End of Transmission line control character.

FORMAT 3: The CD entry may be pre-written and included in the user-created library. The entry may then be included in a COBOL source program by means of a COPY statement. (See "COPY Statement" in the chapter on the Source Program Library Facility in the publication IBM OS Full American National Standard COBOL, Order No. GC28-6396.)

PROCEDURE DIVISION

In the Procedure Division, there is an additional condition which may be used by a COBOL TP program: the message condition.

There are two additional input/output statements used by a COBOL TP program to communicate with the MCP: the RECEIVE statement and the SEND statement.

Each of these language elements is described in the sections that follow.

Message Condition

The message condition determines whether or not one or more complete messages exist in a designated queue of messages. The condition can then be specified in an IF statement.

Format
[NOT] MESSAGE FOR cd-name

The cd-name must specify an input CD entry.

At the time of the test, the CD entry must contain the name of the SYMBOLIC QUEUE to be tested.

A MESSAGE condition exists only if one or more complete messages are present in the named queue. A NOT MESSAGE condition exists if there are no complete messages in the named queue.

Execution of the message condition causes the QUEUE DEPTH field of the named input CD to be updated with the number of complete messages present in the input queue or queue structure. Executing a message condition to a queue structure returns a count of the number of complete messages in the entire structure. Thus, the COBOL TP program can check a queue or queue structure for a predetermined message count before invoking a specific TP processing program.

When using compound IF statements, care must be taken to ensure that the message condition is actually tested, so that the QUEUE DEPTH field will actually be updated. For example, suppose the programmer writes:

```
IF A = B AND MESSAGE FOR QUEUE-IN ...
```

then when A is not equal to B, the message condition is not tested, and the QUEUE DEPTH field for QUEUE-IN is not updated. To ensure that the message condition is tested, the programmer must always write it as the first condition tested within a multiple condition.

When the message condition is executed, the STATUS KEY field of the named input CD is set as follows:

'00' for a valid request

'20' invalid queue name or queue structure

'21' insufficient storage for system control blocks

'29' input/output error

(See Figure 4 for a complete explanation.)

When a STATUS KEY other than '00' is returned, the QUEUE DEPTH field is unchanged.

(For information on the message condition and testing the COBOL TP program using BSAM, see "Testing the COBOL TP Program.")

RECEIVE Statement

The RECEIVE statement makes available to the COBOL program a message, message segment, or a portion of a message or message segment, and pertinent information about that message data from a queue maintained by the MCP.

Format

```
RECEIVE  cd-name { MESSAGE } } INTO  identifier-1
                 { SEGMENT }

     [NO DATA imperative-statement]
```

The cd-name must specify an input CD entry.

Before a RECEIVE statement is executed, this input CD entry must contain, in its SYMBOLIC QUEUE field, a name of up to 12 characters. The first 1 through 8 characters of this name must be unique, and must match the DDname of the DD statement that specifies the queue.

Upon execution of the RECEIVE statement, data is transferred to the receiving character positions of identifier-1, aligned to the left without any SPACE fill and without any data format conversion. The following data items in the input CD are appropriately updated when the RECEIVE statement is executed: MESSAGE DATE field, MESSAGE TIME field, SYMBOLIC SOURCE field, TEXT LENGTH field, END KEY field, STATUS KEY field (see Figure 4, STATUS KEY Field -- Possible Values), and, if the message was retrieved through a queue structure, SYMBOLIC SUB-QUEUE-1 through SYMBOLIC SUB-QUEUE-3.

A complete message need not be received before another MCP queue is accessed. Thus, messages from different MCP queues may be processed at the same time by a COBOL program. (Note, however, that a message is not made available to the COBOL program until it is completely received by the MCP and placed in a queue.)

A single execution of a RECEIVE statement never returns more than a single message (when the MESSAGE phrase is used) or a single segment (when the SEGMENT phrase is used), regardless of the size of the receiving area.

When the MESSAGE phrase is used the end-of-segment condition, if present, is ignored, and the end-of-segment indicator is treated as a data character. (This occurs only when the user, through the MCP, segments the message, and the COBOL program uses MESSAGE mode to RECEIVE the message.) The following rules apply to the data transfer:

- If a message is the same size as identifier-1, the message is stored in identifier-1.
- If a message size is smaller than identifier-1, the message is aligned to the leftmost character position of identifier-1 with no SPACE fill.
- If a message is larger than identifier-1, the message fills identifier-1 left to right, starting with the leftmost character of the message. The remainder of the message can be transferred to identifier-1 with subsequent RECEIVE statements referencing the same queue. Either the MESSAGE or the SEGMENT option may be specified for the subsequent RECEIVE statements.

When the SEGMENT phrase is used, the end-of-segment condition, if present (or the end-of-message condition, if present), determines the end of data transfer. In this case, the end-of-segment indicator is not treated as a data character, and is not transferred with the data. The following rules apply to the data transfer:

- If a segment is the same size as identifier-1, the segment is stored in identifier-1.
- If the segment size is smaller than identifier-1, the segment is aligned to the leftmost character position of identifier-1 with no SPACE fill.
- If a segment size is larger than identifier-1, the segment fills identifier-1 left to right starting with the leftmost character position of the segment. The remainder of the segment can be transferred to identifier-1 with subsequent RECEIVE statements referencing the same queue. Either the MESSAGE or the SEGMENT option may be specified for the subsequent RECEIVE statements.

Once the execution of a RECEIVE statement has returned a portion of a message, only subsequent execution of RECEIVE statements in that run unit can cause the remaining portions of the message to be returned.

After the execution of a STOP RUN statement, or of a GOBACK statement in a main program, the disposition of the remaining portions of any message only partially obtained is not defined.

When the NO DATA option is specified and the queue is empty (that is, there are no complete messages in the input queue), then control passes to the imperative-statement specified in the NO DATA option.

When the NO DATA option is not specified and the queue is empty, execution of the COBOL object program is suspended (that is, placed in wait status) until data is made available in identifier-1.

(For information on the RECEIVE statement and testing the COBOL TP program using BSAM, see "Testing The COBOL TP Program.")

SEND Statement

The SEND statement causes a message, a message segment, or a portion of a message or message segment to be released to the Message Control Program.

Format 1

```
SEND  cd-name  FROM identifier-1
```

Format 2

```
                                    ( WITH identifier-2 )
SEND  cd-name  [FROM  identifier-1] ( WITH ESI          )
                                    ( WITH EMI          )
                                    ( WITH ETI          )
```

Messages may be transferred to the MCP in segments, as complete messages, or in parts of segments or messages. However, data is never transmitted to the named destination until a complete message has been transferred to the MCP.

The cd-name must specify an output CD entry.

Before a SEND statement is executed, this output CD entry must contain:

- In the TEXT LENGTH field, the number of leftmost bytes of contiguous data to be transferred to the output queue from identifier-1.
- In the SYMBOLIC DESTINATION field, the symbolic identification of the remote station(s) that are to receive the message. (The first 1 through 8 characters of this field must be previously defined to the MCP.)

Upon execution of the SEND statement, data is transferred from identifier-1 to the MCP queue corresponding to the terminal identifier contained in the SYMBOLIC DESTINATION field.

As part of the execution of the SEND statement, the MCP interprets the contents of the TEXT LENGTH field to be the user's indication of the number of leftmost character positions of identifier-1 from which data is to be transferred.

If the contents of the TEXT LENGTH field are zero, no characters of data are transferred from identifier-1. (A zero TEXT LENGTH field is valid only with the Format 2 SEND statement.)

If the contents of the TEXT LENGTH field are outside the range of zero through the size of identifier-1 inclusive, an error is indicated in the STATUS KEY field, no data is transferred, and the name in the SYMBOLIC DESTINATION field is not validated. The contents of the STATUS KEY field are updated by the MCP. (See Figure 4, STATUS KEY Field -- Possible Values.)

If the user causes special control characters to be embedded as data characters within the message, these control characters are enqueued with the message, and it is the user's responsibility to ensure that these characters function as intended.

The disposition of a portion of a message not terminated by a subsequent and associated EMI or ETI is undefined. (However, such a message portion will not be transmitted to the destination.)

Format 2 Considerations: This format of the SEND statement allows the programmer to specify whether or not an end indicator is associated with the message.

If the FROM identifier-1 option is omitted, then an end indicator is associated with the data enqueued by a previous SEND statement.

The hierarchy of end indicators, and their meanings, is as follows:

ETI — End of Transmission Indicator -- the CODASYL specification defines the ETI as indicating that the group of messages to be transmitted is complete. However, for this implementation, the ETI is regarded as equivalent to the EMI. Therefore, if ETI is specified without a preceding EMI, the ETI is regarded as an EMI; if the ETI is specified after a preceding EMI, the ETI is treated as comments (that is, is ignored).

EMI — End of Message Indicator -- the message to be transmitted is complete.

ESI — End of Segment Indicator -- the segment to be transmitted is complete.

An ETI need not be preceded by an EMI or ESI. An EMI need not be preceded by an ESI.

Identifier-2 must reference a 1-character integer without an operational sign. The contents of identifier-2 indicate that the contents of identifier-1 have an end indicator associated with them according to the following codes:

If identifier-2 contains:	Then identifier-1 has associated with it:	Meaning
0	No indicator	No indicator
1	ESI	End of Segment Indicator
2	EMI	End of Message Indicator
3	ETI	End of Transmission Indicator

Any character other than 1, 2, or 3 is interpreted as 0.

If the contents of identifier-2 are other than 1, 2, or 3, and identifier-1 is not specified, then an error is indicated in the STATUS KEY field of the associated CD entry, and no data is transferred.

(For information on the SEND statement and testing the COBOL program using BSAM, see "Testing the COBOL TP Program.")

QUEUE STRUCTURE DESCRIPTION AND USE

In a COBOL TP program, a CD FOR INPUT allows the specification of one through three levels of sub-queues from which data can be received; this allows the COBOL object program, at execution time, to make use of pre-defined queue structures, and to access all or parts of such structures. For TP programs, such queue structures are analogous in function and form to the FD entry and its associated 01 record description for file processing programs. If pre-defined queue structures are used, each lowest level sub-queue name in the structure corresponds to a TCAM queue (and consequently must have a matching TPROCESS entry in the MCP terminal table). Figure 5 shows the configuration of one such queue structure.

During program execution, when the user wishes to receive a message from a queue (or sub-queue) he need not place the names of all sub-queues in the input CD; he can specify only the SYMBOLIC QUEUE name, which may be the name of a pre-defined queue structure, or he can specify that name plus one or more sub-queue names which allows him to access only part of the entire structure. A COBOL object-time subroutine uses the name(s) placed in the input CD to determine which lowest level sub-queue(s) (and corresponding TCAM queue(s)) can be used to fulfill the request.

In order to do this, the user must have previously defined all his queue structures in a form that is acceptable to the COBOL object-time subroutine. A utility program that functions as the Queue Structure Description routine (included in the Version 4 Library) makes this possible. Input to the Queue Structure Description routine consists of a series of statements that define queue structures. The statements are written in a COBOL-like format, similar to an FD entry and its associated record description entry. The Queue Structure Description routine produces as output a partitioned data set with one member for each complete queue structure.

Specifying Queue Structures

A COBOL TP program allows the specification of up to three levels of sub-queues. Figure 5 illustrates one such structure; the queue structure shown can be described to the Queue Structure Description routine as follows (FD entry equivalents are shown in parentheses):

```
QUEUE IS A.                          (FD clause)
  SUB-QUEUE-1 IS B.                  (01 entry)
    SUB-QUEUE-2 IS D.                  (02 entry)
      SUB-QUEUE-3 IS H.                  (03 entry)
      SUB-QUEUE-3 IS I.                  (03 entry)
    SUB-QUEUE-2 IS E.                  (02 entry)
      SUB-QUEUE-3 IS J.                  (03 entry)
      SUB-QUEUE-3 IS K.                  (03 entry)
  SUB-QUEUE-1 IS C.                  (01 entry)
    SUB-QUEUE-2 IS F.                  (02 entry)
      SUB-QUEUE-3 IS L.                  (03 entry)
      SUB-QUEUE-3 IS M.                  (03 entry)
    SUB-QUEUE-2 IS G.                  (02 entry)
      SUB-QUEUE-3 IS N.                  (03 entry)
      SUB-QUEUE-3 IS O.                  (03 entry)
```

In general, the queue structure need not include all three levels of sub-queues; however, if a lower level is specified all higher levels in that leg of the structure must also be specified. Any particular sublevel of the structure should always have more than one sub-queue

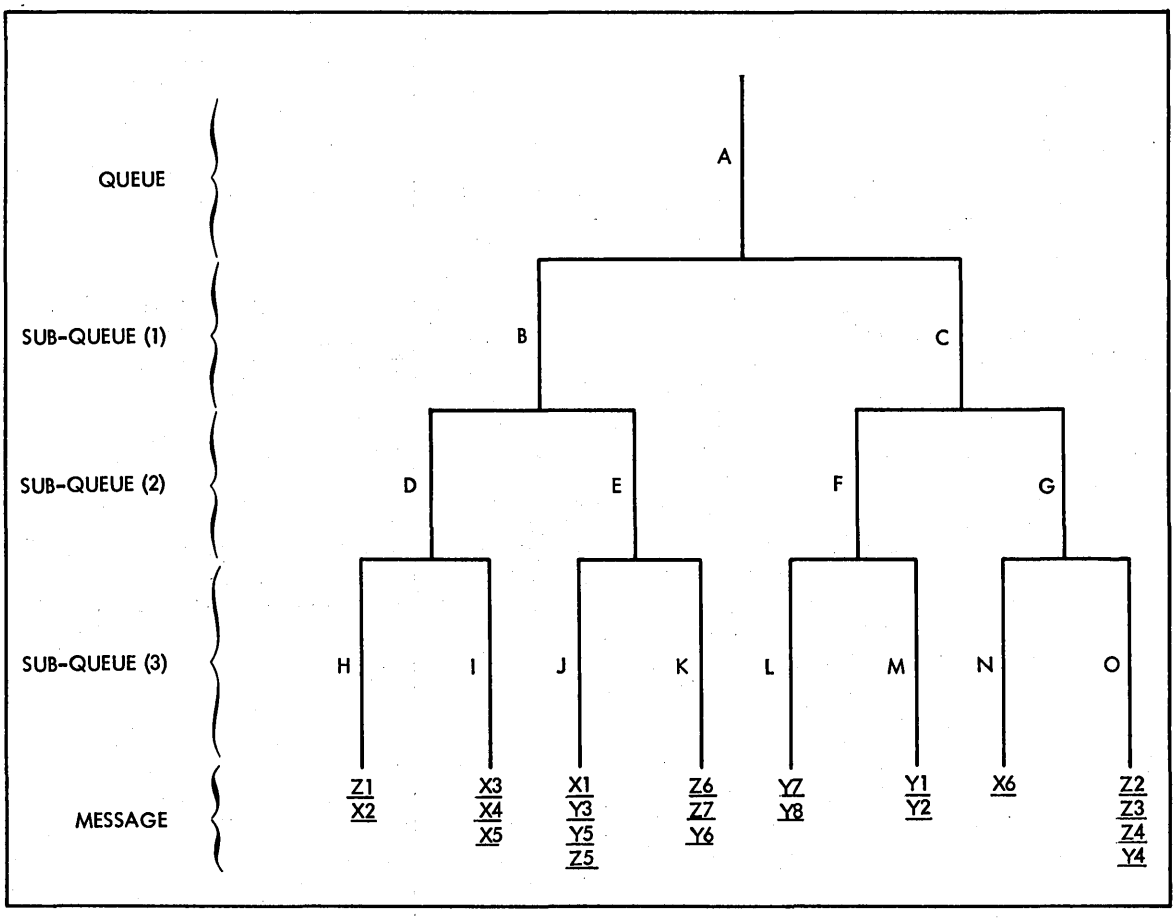


Figure 5. Queue Structure with Three Levels of Sub-Queues

contained within it. (That is, if in the preceding example only sub-queues B through K were subordinate to queue A, then B as a sub-queue would be superfluous, since it would not represent a subdivision of A. In this case the logical presentation would be a structure with sub-queues H through K at the SUB-QUEUE-2 level, with sub-queues D and E at the SUB-QUEUE-1 level, and with either B or A at the QUEUE level.) The lowest level defined (which corresponds to an elementary data description entry) must be used as a DDname at object time to point to an entry in the MCP terminal table, since such entries correspond to TCAM queues.

Accessing Queue Structures Through COBOL

Once the queue structure(s) are defined and stored in a partitioned data set, the user can create COBOL TP programs that utilize these predefined structures. At execution time, the partitioned data set is described on a DD card, and the MCP table entries and the lowest level sub-queue names are linked by DD cards. The name of the DD card may be defined as the sub-queue name itself (in the above example, as H, I, J, K, L, M, N, or O). Alternatively, DDnames that are equivalent to the lowest level sub-queue names may be defined (that is, in the above example, that each sub-queue name (H, I, J, K, L, M, N, or O) would have its equivalent DDname); this permits the COBOL program to reuse the symbolic sub-queue names without ambiguity.

Before a RECEIVE statement is executed, the user places the needed queue and sub-queue name(s) in the CD entry. When the RECEIVE statement is executed, the RECEIVE subroutine first checks for the presence of the partitioned data set describing these queue structures. If the data set is present, the RECEIVE subroutine invokes a Queue Analyzer routine which searches the partitioned data set for a member corresponding to the name in the SYMBOLIC QUEUE field, reads that member into main storage and uses it to validate the SYMBOLIC SUB-QUEUE name(s) in the COBOL program input CD entry. The Queue Analyzer routine then determines the first valid name for the structure specified and gives this name to the RECEIVE routine.

Names at the SUB-QUEUE-1 level take priority over names at the SUB-QUEUE-2 level. Names at the SUB-QUEUE-2 level take priority over names at the SUB-QUEUE-3 level. At any given level, names at the left take priority over, and are completely evaluated before, names at the right. (Taking advantage of this retrieval technique, the user can improve object-time performance by defining his most frequently used sub-queues at the left of the structure.)

The RECEIVE routine then attempts to access the queue specified. If the DD card for this queue is not present, or if there are no messages in the associated MCP queue, the Queue Analyzer provides the RECEIVE routine with another valid name. The procedure is repeated until the RECEIVE routine accesses a message, or until there are no more queues to access.

During a RECEIVE operation, a COBOL program using queue structures need not specify all levels of sub-queues. The highest level (QUEUE) must be specified; that level plus a SUB-QUEUE-1 may also be specified; those two levels plus a SUB-QUEUE-2 may also be specified; or all four levels may be specified. Note that if a lower level is specified, then all higher levels must also be specified.

If the COBOL programmer wishes to access the next message in the queue structure, regardless of which sub-queue that message may be in, he specifies the queue name only, and initializes the sub-queue names to

SPACES. The MCP, when supplying the message, returns to the COBOL object program any applicable sub-queue names via the data items in the associated input CD. If, however, the programmer desires the next message in a given sub-queue, he must specify both the queue name and any applicable sub-queue names. Once a program has begun receiving any part of a message from a queue (or sub-queue), subsequent requests must return all applicable names until end of message is indicated.

Referring to the queue structure shown in Figure 5, the following examples illustrate several message retrieval options when a RECEIVE statement is executed:

- Input CD contains: Queue A. MCP returns: Message Z1.
- Input CD contains: Queue A, sub-queue-1 C. MCP returns: Message Y7.
- Input CD contains: Queue A, sub-queue-1 B, sub-queue-2 E. MCP returns: Message X1.
- Input CD contains: Queue A, sub-queue-1 C, sub-queue-2 G, sub-queue-3 N. MCP returns: Message X6.

Specification of DDnames with Elementary Sub-Queues

An application program is written to accept TP messages as input to an inventory control process. Five different locations will each transmit data on four different parts. To make this relationship explicit, a diagram is prepared showing a queue structure containing all the elements of the input. The queue structure for this application is shown in Figure 6.

Each elementary, or lowest-level, queue in the structure must specify the name of a DD card, which in turn will name a TPROCESS entry or TCAM queue. While the example shown in Figure 6 is unambiguous (that is, INVENTORY.CHICAGO.PARTA is distinct from INVENTORY.LOS-ANGELES.PARTA), the elementary queues by themselves are not (that is, the elementary name PARTA, which corresponds to a DDname, can be any one of five different PARTA's). To eliminate this ambiguity, the user of queue structures can define DDnames in addition to the sub-queue names at the lowest level when he defines the structure to the Queue Structure Description routine. Then the object-time Queue Analyzer routine will automatically make the association from the fully qualified queue structure names to the DDnames required. Thus, in this example:

```
NEW-YORK.PARTA  could have DDname  DD1
NEW-YORK.PARTB    "    "     "     DD2
NEW-YORK.PARTC    "    "     "     DD3
NEW-YORK.PARTD    "    "     "     DD4
CHICAGO.PARTA     "    "     "     DD5
CHICAGO.PARTB     "    "     "     DD6
      .
      .
      .
```

In this way, each elementary queue has a unique designation, yet the COBOL program can refer to the sub-queue names without ambiguity.

Note: If a DDname is assigned to a sub-queue within a queue structure, that same DDname cannot be used as a queue name to access the queue directly.

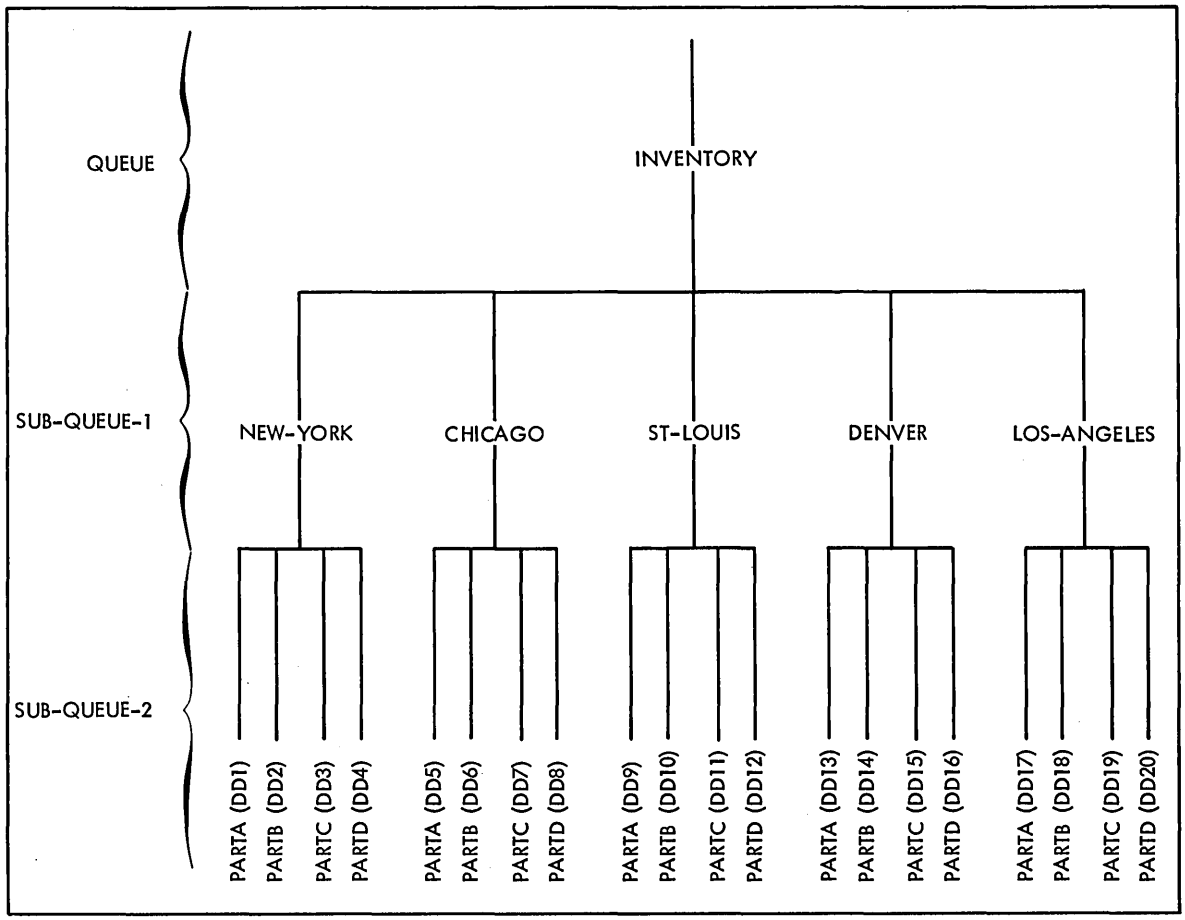


Figure 6. Using DDnames with Queue Structures

Rules For Queue Structure Description

For each member of the partitioned data set, the input to the Queue Structure Description Routine must take the following format:

```
                                Format
{QUEUE} IS data-name-1
{Q    }

{{ {SUB-QUEUE-1} IS data-name-2[(DDname-1)]}...
   {SUBQ1      }

   [[ {SUB-QUEUE-2} IS data-name-3[(DDname-2)]]...
      {SUBQ2      }

      [ {SUB-QUEUE-3} IS data-name-4[(DDname-3)]]... ]} ...
        {SUBQ3      }
```

The clauses of the queue structure may be written free form; however, only one clause may appear on each 80-character record. At least one sub-queue level must be specified; no more than 200 sub-queue names may be specified in one queue structure.

The sub-queues at each level must be specified to the Queue Structure Description routine in left-to-right order. When the queue structure is referred to at object program execution time, names at a higher level take priority over names at a lower level. At a given level in the queue structure, names to the left take priority over names to the right.

A queue structure need not include all levels of sub-queues. However, if a lower level is included in one leg of a queue structure, then that leg must include all higher levels.

Each clause of the structure may optionally be followed by a period.

Data-name-1 is the name of the queue structure, and becomes the name of that member of the partitioned data set.

Data-name-2 through data-name-4 are sub-queue names within the data set member.

Each data-name used as a queue or sub-queue name may be up to 12 alphanumeric characters in length; the first character must not be numeric.

Each DDname, if specified, may be up to 8 alphanumeric characters in length; the first character must not be numeric.

Each data-name at the lowest (elementary) level of a leg of the queue structure may be a DDname; alternatively, each such data-name may be followed by a parenthesized DDname. If a parenthesized DDname follows a sub-queue name, the left parenthesis must immediately follow the sub-queue name with no intervening spaces. There must be no spaces between the parentheses and the DDname.

INTERFACE CONSIDERATIONS

For input, the job control language in the COBOL TP program's job stream must contain a DD card for each input queue accessed by the MCP; the name by which the queue is known to the MCP appears as a parameter on this control card. In addition, for each input message, the MCP must be designed to perform the following functions:

- Inclusion of the date and time of message arrival
- Identification of the source terminal
- When desired, indication of end-of-segment and/or logical end-of-transmission

For output, the COBOL TP program's job stream can contain only one DD card to define an output destination queue. A parameter on this card contains the name by which this queue is known to the MCP. In addition, the destination names used in the COBOL program must be known to the MCP; the destinations should be listed in the MCP terminal tables.

Other items used by COBOL, such as, for example, the TEXT LENGTH field and the STATUS KEY field, are provided by TCAM and need not concern the MCP programmer.

Note: While it is not necessary, it is more efficient if the message processing system is planned so that the MCP and the COBOL TP program treat messages and message segments in a consistent manner.

EXECUTION TIME CONSIDERATIONS

At execution time, one DD statement is required for each queue that the COBOL TP program accesses.

Through the DD statement the programmer equates the name placed in the SYMBOLIC QUEUE field of an input CD entry with the queue name used in the MCP.

For an output CD, the name placed in the SYMBOLIC DESTINATION field must be known to the MCP. All output messages are enqueued to the same TCAM destination queue; thus only one DD statement is needed for output.

(Note that for a COBOL TP program the system regards all records as variable in length and the organization as always physical sequential.)

The COBOL TP system maintains control and status information pertinent to each message being processed, and to each active queue. The information is maintained in Queue Blocks, which are created the first time each queue is accessed by a COBOL TP run unit, whether by an IF MESSAGE, a RECEIVE, or a SEND statement. Each Queue Block is chained to all other Queue Blocks in the same region/partition.

The system provides a single buffer of 200 characters for all queues; the user can override this default size through the DD statement. Note, however, that the size of the buffer places no restriction on the amount of data the COBOL program can request. The amount of data moved into the COBOL work area with each request is either the complete work unit (message or segment), or the size of the work area, whichever is smaller. The amount of data moved out of the COBOL work area with each request is the number of characters specified in the TEXT LENGTH field of the output CD. Within a program, the first reference to any queue (input or output) causes the system to obtain a buffer; the BLKSIZE field of the DD card associated with that queue is used to determine if the default buffer size is to be used.

Before the MESSAGE condition is executed, the name of the DD statement that specifies the input queue or queue structure must be moved into the SYMBOLIC QUEUE field of the input CD entry by the COBOL programmer. When the IF MESSAGE statement is executed, the following actions take place:

- If this is the initial reference in the run unit to the named queue, the system constructs a Queue Block.
- The STATUS KEY field of this input CD entry is updated as shown in Figure 4. If the STATUS KEY is '00' then the QUEUE DEPTH field of this input CD entry is updated to contain the number of complete messages contained in the named input queue.

When a RECEIVE statement is executed, the following actions take place:

- If this is an initial reference in the run unit to the named queue, the system constructs a Queue Block
- The STATUS KEY field is updated as shown in Figure 4. If the STATUS KEY is '00' then:
  1. The appropriate fields of the input CD are updated.
  2. The requested amount of data is transferred to the COBOL work area. (In certain instances, no data will be transferred when ETI (logical end-of-file) is indicated.)

However, if the queue is empty, the following actions take place:

1. If the NO DATA option is specified, control is transferred to the appropriate imperative-statement. If ETI (logical end-of-file) is concurrently indicated, the queue-empty condition is ignored, and ETI is indicated.

2. If the NO DATA option is not specified, the system places the COBOL program into the wait state. If data does not become available before the end of the time interval specified for the job step, the system terminates the run unit.

For a SEND statement the system permits the piecemeal construction of messages and segments by multiple programs within one region/partition. When a SEND statement is executed the following actions take place:

- If this is the initial SEND statement in the run unit the system constructs a Queue Block.
- If this is the initial SEND statement of the message, the leftmost eight characters of the SYMBOLIC DESTINATION field are placed into the appropriate field in the buffer.
- The STATUS KEY field of the output CD entry is updated as shown in Figure 4.
- The number of characters indicated in the TEXT LENGTH field is transferred from the COBOL work area into the output buffer. If ESI is indicated, an end of record (EOR) delimiter is appended as the last data character. The user's EOR delimiter is obtained from the MCP. If ESI is not specified, no EOR delimiter is appended.
- The appropriate code (partial segment, end segment, or end message) is placed in the TCAM control byte of the buffer.

Testing the COBOL TP Program

The user can test his COBOL TP program using physical sequential data sets. By eliminating the MCP queue name on the DD card for a queue, the user links to the Basic Sequential Access Method (BSAM) instead of to TCAM. The BSAM file must conform to the following:

- Each block must be an unblocked V mode record
- Each block may contain no more than one logical record
- A logical record may span a physical block
- End of Segment may be indicated in the TCAM control byte
- The format for each physical record must be as shown in Figure 7 (each block of each message contains the 8-byte source ID field)

For BSAM files the DD card must specify the size of the largest physical block in the file -- including the V prefix, the TCAM control byte, and the source ID. The system obtains the DATE and TIME information for the input CD entry (via the OS TIME macro), and also a unique buffer for the file. The user may intermix TCAM queues and BSAM files in one program. In this instance all TCAM queues share a common buffer as previously described, the buffer size being determined either as the default size, or as the BLKSIZE of the DCB parameter on the DD card for the first TCAM queue; each BSAM file has its own unique buffer, the buffer size being determined from the BLKSIZE of the DCB parameter on the DD card for each file.

The codes used in the TCAM control byte, and their meanings, are:

Code	Meaning
X'F1'	First block of multiblock message
X'F5'	First block of multiblock message, end of segment indicated
X'40'	Intermediate data block
X'F4'	Intermediate data block, end of segment indicated
X'F2'	Last block of multiblock message
X'F6'	Last block of multiblock message, end of segment indicated
X'F3'	Single block message
X'F7'	Single block message, end of segment indicated

Note: If the user prepares a SAM file for input, then he must insert the TCAM V prefix and the TCAM control byte, using the codes specified. The SEND statement can be used to create a properly formatted SAM file for subsequent input; COBOL then adds bytes 1 through 13 automatically.

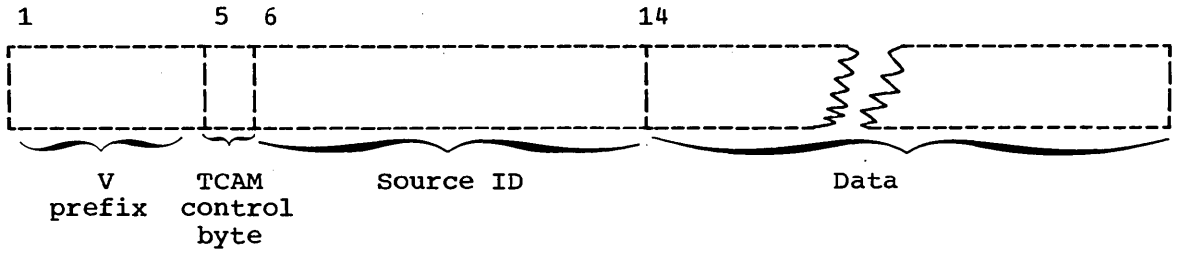


Figure 7. Structure of TCAM Record

Execution of the IF MESSAGE statement, when the named queue is serviced by BSAM, always results in a message count of 1 until end-of-file has been detected. After end-of-file has been detected, a count of 0 is always returned.

When the RECEIVE statement is executed, and the named queue is serviced by BSAM, the CD contains the source for the last block transferred. The BSAM end-of-file condition is treated as an ETI. Execution of a subsequent RECEIVE statement with the NO DATA option causes the appropriate NO DATA exit to be taken. If the NO DATA option is omitted, a STOP RUN is executed.

The user can create properly formatted messages for subsequent test input by using the SEND statement. When the SEND statement is executed, and ESI is indicated, the system sets the appropriate bit of the TCAM prefix ON.

If the output (SEND) file is to be printed, the control characters specified in the DD card for the file must indicate that ASA print control characters are to be used. The TCAM control byte occupies the print control position of the record, and the TCAM control characters are valid ASA characters. The block size of the output record, therefore, should not indicate a value greater than the print line of the printer.

String manipulation statements allow the COBOL programmer greater flexibility in data manipulation. With the STRING statement he can concatenate two or more subfields into a single field. With the UNSTRING statement he can separate contiguous data in a single field into multiple logical subfields. The subfields need not be contiguous.

STRING Statement

The STRING statement provides juxtaposition of the partial or complete contents of two or more data items into a single data item.

Format

```
STRING { identifier-1 } [ identifier-2 ] ... DELIMITED BY { identifier-3 }
       { literal-1    }   [ literal-2    ]                 { literal-3    }
                                                           { SIZE         }

   [ { identifier-4 } [ identifier-5 ] ... DELIMITED BY { identifier-6 } ] ...
     { literal-4    }   [ literal-5    ]                 { literal-6    }
                                                        { SIZE         }

   INTO identifier-7  [WITH POINTER  identifier-8]

   [ON OVERFLOW imperative-statement]
```

All literals must be described as nonnumeric literals. Each literal may be any figurative constant without the optional word ALL.

All identifiers, except identifier-8, must be described implicitly or explicitly as USAGE IS DISPLAY. Identifier-3 and identifier-6 must each reference a fixed length data item.

Identifier-7 must represent an elementary data item without editing symbols. If a SEPARATE SIGN clause is specified, it is ignored during execution of the STRING statement.

Identifier-8 must represent an elementary numeric integer data item of sufficient size to contain a value equal to the size plus 1 of the area referenced by identifier-7.

All references to identifier-1, identifier-2, identifier-3, literal-1, literal-2, and literal-3 apply equally to identifier-4, identifier-5, identifier-6, literal-4, literal-5, and literal-6, respectively, and all repetitions thereof.

Identifier-1, literal-1, identifier-2, and literal-2 represent the sending items. Identifier-7 represents the receiving item.

Literal-3 and identifier-3 indicate the character(s) delimiting the move. If the SIZE phrase is used, the complete data item defined by identifier-1, literal-1, identifier-2, literal-2 is moved.

When a figurative constant is specified as literal-1, literal-2, it refers to an implicit 1-character data item whose USAGE IS DISPLAY.

When the STRING statement is executed, the transfer of data is governed by the following rules:

- Those characters from the sending item(s) are transferred to the receiving item in accordance with the rules for alphanumeric to alphanumeric moves, except that no SPACE filling is provided. (See the MOVE statement in "Procedure Division" of IBM OS Full American National Standard COBOL, Order No. GC28-6396.)

- If the DELIMITED phrase is specified without the SIZE option, the contents of each sending item are transferred to the receiving data item in the sequence specified in the STRING statement, beginning with the leftmost character of the first sending item, and continuing from left to right through each successive sending item until either:

  1. The delimiting character(s) (literal-3/identifier-3, or literal-6/identifier-6) for this sending item are reached, or

  2. The rightmost character of this sending item has been transferred.

  The delimiting character(s) are not transferred into the receiving data item. When the receiving field is filled, or when all of the DELIMITED data in all of the sending fields has been transferred, the operation is ended.

- If the DELIMITED phrase is specified with the SIZE option, the entire contents of each sending item are transferred, in the sequence specified in the STRING statement, to the receiving data item. The operation is ended either when all data has been transferred or when the receiving field is filled.

The POINTER option may be used explicitly by the programmer to designate where data is to be placed in the receiving area. If the POINTER option is specified, identifier-8 is explicitly available to the user, and he is responsible for setting its initial value. The initial value must not be less than one and must not exceed the number of character positions of the receiving item. (Note that the POINTER item must be defined as of sufficient size to contain a value equal to the size of the receiving item plus one. This precludes the possibility of arithmetic overflow when the system updates the pointer.) The following rule applies:

- Conceptually, when the STRING statement is executed, the following actions occur. Characters are transferred into the receiving item one at a time, beginning at the character position indicated by the POINTER value. After each character is positioned, the value of the POINTER item (identifier-8) is increased by one. The value associated with the POINTER item is changed only in this manner. At the termination of any STRING operation, the value in the POINTER item always points to one character beyond the last character moved into the receiving item.

Note: The POINTER value may therefore be used in a subsequent STRING statement to place additional characters immediately to the right of those already placed in the receiving item.

If the POINTER option is not specified, the STRING statement acts as if the user had specified a pointer with an initial value of one. When the statement is executed, the implicit pointer is incremented as described above. The implicit pointer is not available to the user.

At the end of execution of a STRING statement, only that portion of the receiving item referred to during execution of the STRING statement is changed. All other portions of the receiving item contain data that was present before this execution of the STRING statement.

If at any time during or after initialization of the STRING statement, but before execution of the STRING statement is completed, the value associated with the POINTER item is less than one, or exceeds the number of character positions in the receiving item, no (further) data is transferred, and, if specified, the imperative-statement in the ON OVERFLOW option is executed. If the ON OVERFLOW option is not specified and the conditions described above are encountered, control passes to the next statement as written.

UNSTRING Statement

The UNSTRING statement causes contiguous data in a sending field to be separated and placed into multiple receiving fields.

Format

```
UNSTRING  identifier-1

 [DELIMITED BY [ALL] {identifier-2} [OR [ALL]  {identifier-3} ]...]
                     {literal-1   }            {literal-2   }

 INTO identifier-4   [DELIMITER IN  identifier-5]

           [COUNT IN identifier-6]

      [identifier-7  [DELIMITER IN  identifier-8]

           [COUNT IN identifier-9] ] ...

 [WITH POINTER  identifier-10]  [TALLYING IN  identifier-11]

 [ON OVERFLOW  imperative-statement]
```

Each literal must be described as nonnumeric. In addition, each literal may be any figurative constant without the optional word ALL.

Identifier-1, identifier-2, identifier-3, identifier-5, and identifier-8 must each be described, implicitly or explicitly, as an alphanumeric data item.

Identifier-4 and identifier-7 must each be described, implicitly or explicitly, as USAGE DISPLAY.

Identifier-6, identifier-9, identifier-10, and identifier-11 must be described as elementary numeric integer data items.

No identifier may name a level-88 entry.

The DELIMITER IN option and the COUNT IN option may be specified only if the DELIMITED BY option is specified.

All references to identifier-2, literal-1, identifier-4, identifier-5, and identifier-6 apply equally to identifier-3, literal-2, identifier-7, identifier-8, and identifier-9, respectively, and all repetitions thereof.

Identifier-1 represents the sending area.

Identifier-4 represents the data receiving area. Identifier-5 represents the receiving area for delimiters.

Literal-1 or identifier-2 specifies a delimiter. No more than 15 delimiters may be specified.

Identifier-6 represents the count of the number of characters within the sending area isolated by the delimiters for the move into the current receiving area. This value does not include the count of the delimiter character(s).

Identifier-10 contains a value that indicates a relative character position within the sending area.

Identifier-11 is a counter that records the number of receiving areas acted upon during the execution of the UNSTRING statement.

When the ALL option is specified, two or more contiguous occurrences of literal-1 or of identifier-2 are treated as if they were only one occurrence. However, identifier-5 (the receiving area for delimiters) contains as many complete occurrences of the delimiter as are present or as it can hold, whichever is smaller.

When ALL is specified, and two or more delimiters are found, as much of the first occurrence of the delimiter as will fit is moved into identifier-5. Each additional occurrence of the delimiter is moved into identifier-5 only if the complete occurrence will fit.

When ALL is not specified, and the examination encounters two contiguous occurrences of literal-1 or identifier-2, the current receiving area for data is either space-filled or zero-filled, according to the description of the receiving area.

When a figurative constant is used as a delimiter, it stands for a single character nonnumeric literal. Two or more occurrences of the figurative constant are treated as if they were only one occurrence. However, identifier-5 (the receiving area for delimiters) contains as many occurrences of the figurative constant as are present or as it can hold, whichever is smaller.

Literal-1 or identifier-2 may contain any characters in the EBCDIC character set.

Each literal-1 or identifier-2 represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present in contiguous positions in the sending field, and in the sequence specified, to be recognized as that delimiter.

When two or more delimiters are specified in the DELIMITED BY option, an OR condition exists. Each non-overlapping occurrence of any one of them is considered a delimiter, and is applied to the sending field in the sequence specified in the UNSTRING statement. For example, if DELIMITED BY AB OR BC is specified, then an occurrence of either AB or BC in the sending field is considered a delimiter; an occurrence of ABC is considered an occurrence of AB.

When the UNSTRING statement is initiated, the current receiving area is identifier-4. Data is transferred from identifier-1 to identifier-4 according to the following rules:

- If the POINTER option is specified the string of characters in the sending area is examined beginning with the relative character position indicated by the contents of the POINTER item. If the POINTER option is not specified, the character string is examined beginning with the leftmost character position.

- If the DELIMITED BY option is specified, the examination proceeds left to right until a delimiter specified by either literal-1 or the value in identifier-2 is encountered. If the end of the sending item is encountered before the delimiting condition is met, the examination terminates with the last character examined.

- If the DELIMITED BY option is not specified, the number of characters examined is equal to the size of the current receiving area. The size of the receiving area depends on its data category:

  1. If it is alphanumeric or alphabetic (without SPACE insertion characters), its size is equal to the size of the current receiving area.

  2. If it is alphanumeric edited, then its size is equal to the size of the current receiving area less a number equal to the sum of the number of simple, special, and fixed insertion characters.

  3. If it is alphabetic (with SPACE insertion characters), then it is treated as if it were alphanumeric edited.

  4. If it is numeric, then the size is equal to the integer portion of the current receiving field, including scaling characters.

  5. If it is numeric edited, its size is equal to the size of the integer portion of the current receiving field less a number equal to the sum of the number of fixed, simple, and special insertion characters, minus one, if the integer portion contains floating insertion characters. (Simple insertion characters embedded within the floating string are included in the count of simple insertion characters.)

  6. If it is described with one or more P's in its PICTURE, then:

     a. If the P's appear to the left in the PICTURE character string, it is considered to be of zero length.

     b. If the P's appear to the right in the PICTURE character string, its total integer size is considered to be the length of the PICTURE string, including the P's. (For example, if the sending area contains the value 1234 and the receiving area is described with the PICTURE 99PP, then after execution of the UNSTRING statement, the receiving area contains the value 12. The UNSTRING statement bypasses 34.)

  7. If it is described with the SEPARATE SIGN clause, one fewer than the number of digit positions is placed in the receiving area.

- The characters thus examined (excluding the delimiting character(s), if any) are treated as an elementary alphanumeric data item, and are moved into the current receiving area according to the rules for an alphanumeric move. (See the MOVE statement in the Procedure Division chapter of IBM OS Full American National Standard COBOL, Order No. GC28-6396.) Note that if two delimiters are adjacent,

that is, with no data characters between them, the null receiving field is filled with zeros or spaces, depending on its description.

- If the DELIMITER IN option is specified, the delimiting character(s) are treated as an elementary alphanumeric data item and are moved into identifier-5 according to the rules for an elementary move. If the delimiting condition is the end of the sending area, then identifier-5 (the DELIMITER) is space-filled or zero-filled according to its PICTURE character string.

- If the COUNT IN option is specified, a value equal to the number of characters thus examined (excluding the delimiter character(s), if any) is moved into identifier-6 according to the rules for an elementary move.

- If the DELIMITED BY option is specified the string of characters is further examined beginning with the first character to the right of the delimiter. If the DELIMITED BY option is not specified the string of characters is further examined beginning with the character to the right of the last character transferred.

- After data is transferred to identifier-4, the current receiving area becomes identifier-7. The procedure described is then repeated either until all the characters in the sending area have been transferred, or until there are no more unfilled receiving areas.

The initialization of the data items associated with the POINTER phrase and the TALLYING phrase is the responsibility of the user.

The contents of the data item referenced by identifier-10 (the POINTER item) behave as if incremented by one for each character examined in the sending area. When the execution of an UNSTRING statement with a POINTER option is completed, the contents of identifier-10 contain a value equal to the initial value plus the number of characters examined in the sending area.

When the execution of an UNSTRING statement with the TALLYING option is completed, the contents of identifier-11 contain a value equal to the initial value plus the number of data receiving areas acted upon (including null fields).

Either of the following situations causes an overflow condition:

- An UNSTRING statement is initiated, and the value in the POINTER item (identifier-10) is less than one or greater than the size of the sending area.

- If, during the execution of an UNSTRING statement, all receiving areas have been acted upon, and the sending area still contains characters that have not been examined.

When an overflow condition exists, the UNSTRING operation is terminated. If an ON OVERFLOW option is specified, the imperative-statement included in the ON OVERFLOW option is executed. If the ON OVERFLOW option is not specified, control passes to the next statement as written.

Under previous versions of Full American National Standard COBOL, all programs and subprograms, plus their required COBOL library subroutines, were linked into one load module for execution in one partition/region. Thus, many copies of one COBOL library subroutine might be resident in core storage at one time, one in each partition/region. The COBOL Library Management Facility is an optional feature that allows a single copy of the COBOL library subroutines to be shared by all COBOL programs in the same or different partitions/regions.

When the library management facility is used, the COBOL library subroutines may be wholly or partially resident in the MVT Link Pack Area (LPA) or in the MFT Resident Reusable Routine area (RRR), or they may be resident within each partition/region. (Four routines cannot be so placed -- the subroutine used for intraregion/intrapartition communication, the queue structure description routine, a STOP RUN routine, and a special DISPLAY routine.) The actual physical location of these routines is transparent to the executing program.

The primary advantage in the placement of the COBOL Library Subroutines in the LPA/RRR area is the economy it allows in main storage allocation. Though the LPA/RRR area must be made larger to accomodate all the required COBOL library subroutines, each region/partition no longer requires its own copy.

To be able to place the COBOL subroutines in the LPA/RRR area, the user must execute a utility program to add two members to the system parameter library. The members are:

1. A User List -- a list of all names and all aliases for those COBOL subroutines the user wishes to place in the LPA/RRR area.

2. A Linkage Routine that allows the concatenation of the system link library with the COBOL subroutine library, or with a private library containing selected COBOL subroutines. (Note that if the user wishes to place selected COBOL subroutines into his private library, he must execute a utility program to catalog that library.)

At initial program loading (IPL) time, the user identifies the user list to the system. The system then uses the linkage routine to place the listed COBOL subroutines into the LPA/RRR area.

Note: If the user does not wish to place any COBOL subroutines in the LPA/RRR area, he need not execute the utility program mentioned above. He may still make use of the COBOL Library Management facility; however, all library subroutines will be loaded into his own region/partition when they are needed by one or more programs, and deleted when they are no longer needed. Thus, not all library subroutines needed by all programs in the region need be resident at the same time. In this case, however, the user must supply a job control card at execution time pointing to the COBOL subroutine library, or to his own private library of COBOL subroutines.

A complete discussion of load module location is included in the publication:

IBM System/360 Operating System: Supervisor Services, Order No. GC28-6646.

Note that the required COBOL library subroutines are no longer part of the COBOL program load module when this option is selected; therefore, secondary storage needed for the COBOL load module is decreased, due to the smaller size of the module. There is, however, some additional overhead if required subroutines are not resident in the LPA/RRR and must be loaded into the region/partition of the requesting program.

Specifying the COBOL Library Management Facility

The COBOL Library Management Facility is optioned at compile time through the PARM field of the EXEC job control statement. The option is in the following form:

PARM=RESIDENT
which specifies that the COBOL Library Management Facility will be used.

PARM=NORESIDENT
which specifies that the COBOL Library Management Facility will not be used. NORESIDENT is the default option.

In any given region/partition, if the COBOL Library Management Facility is used at all, it must be used by the main program and by all subprograms in that region/partition. Otherwise, multiple copies of COBOL library subroutines may be resident at the same time and cause unpredictable results.

In a region/partition using the COBOL Library Management Facility, a single copy of each COBOL library subroutine selected at IPL time is shared by all programs and subprograms loaded into that region/partition (all such sharing programs must specify the COBOL Library Management Facility). This same single copy is also used by all programs and subprograms loaded into other regions/partitions using the COBOL Library Management Facility. The COBOL library subroutines are placed in the LPA/RRR area.

For a region/partition not using the COBOL Library Management Facility, the COBOL object program and the COBOL library subroutines it uses are link edited together into one load module.

If COBOL library subroutines that were not loaded into the LPA/RRR area at IPL time are required for execution of the program, and the COBOL Library Management Facility is being used, then:

- For a main program, such subroutines are loaded into the region/partition before execution of the main program.
- For a subprogram, those required subroutines that have not yet been loaded are loaded into the region/partition directly before subprogram initialization. Thus, there is only one copy of the subroutine resident in each region/partition.

Programming Considerations

For the dynamic CALL and CANCEL functions, the COBOL Library Management Facility is an implied required feature. (See "Dynamic Subprogram Linkage.")

The dynamic CALL/CANCEL statements are used in conjunction with the COBOL Library Management Facility. PARM=DYNAM implies the library management facility, even when the parameter RESIDENT is omitted or PARM=DYNAM,NORESIDENT is coded.

When NODYNAM and NORESIDENT are specified, or implied by default, and a CALL identifier or CANCEL identifier statement occurs in the source program being compiled, the library management facility is automatically optioned, and a printed indication is given in the compiler output.

Programs written and compiled with previous versions of the IBM OS Full American National Standard COBOL Compiler are compatible without recompilation. Such programs do not utilize the COBOL Library Management Facility.

All programs and subprograms within one region/partition must be compiled either using PARM=RESIDENT or using PARM=NORESIDENT. Otherwise, results may be unpredictable.

A new option of the CALL statement and the addition of the CANCEL statement permit dynamic loading and deletion of COBOL subprograms in the COBOL processing environment.

The CALL statement, as it has previously been specified for IBM OS Full American National Standard COBOL, has been static. That is, the main COBOL program and all subprograms invoked with the CALL statement must have been part of the same load module. Thus, when a subprogram was called it was already core-resident, and a branch to it occurred. Subsequent execution of CALL statements entered that subprogram in its last-used state. If alternate entry points were specified, then any CALL to the subprogram could select any of the alternate ENTRY points at which to enter the subprogram. If the linking of all subprograms with the main program resulted in a load module that required more main storage than was available, then the user could utilize the Segmentation feature. Now, with the implementation of the dynamic CALL and CANCEL statements, the COBOL user can control the modules that are to be core-resident.

For the Version 4 Compiler, the CALL statement can also be specified as dynamic; that is, the called subprogram is not link edited with the main program, but is instead link edited into a separate load module, and at execution time is loaded only if and when it is required (that is, when it is called).

Each subprogram invoked with a dynamic CALL statement may be part of a different load module, which is a member of the system link library or of a user-supplied private library. The execution of the dynamic CALL statement to a subprogram that is not core-resident results in the loading of that subprogram from secondary storage into the region/partition containing the main program, and a branch to the subprogram.

Thus, the first dynamic CALL to a subprogram obtains a fresh copy of the subprogram. Subsequent calls to the same subprogram (either by the original caller or by any other subprogram within the same region/partition) result in a branch to the same copy of the subprogram in its last-used state. However, when a CANCEL statement is issued for that subprogram, the storage occupied by the subprogram is freed, and a subsequent CALL to the subprogram will function as though it were the first. A CANCEL statement referring to a called subprogram may be issued by a program other than the original caller. In order for the CALL statement to function as defined by CODASYL, the user subprograms must be link edited as non-reentrant and non-serially-reusable.

Specifying the Dynamic CALL

Through the PARM field of the EXEC job control statement, the user can specify the mode (static or dynamic) of the CALL literal option. The parameters are specified as follows:

PARM=DYNAM
indicates to the compiler that all subprograms are to be dynamically loaded at object time.

PARM=NODYNAM
indicates to the compiler that subprograms invoked through the CALL literal statement are to be linked with the main program into a single load module. PARM=NODYNAM is the default option. (Note that subprograms invoked through the CALL identifier statement are always dynamically loaded at object time.)

In the EXEC job control statement, when the combination PARM=DYNAM,NORESIDENT occurs, the system overrides the NORESIDENT option, since DYNAM implies the use of the COBOL Library Management Facility.

When the dynamic CALL statement is used at object time, the COBOL Library Management Facility must also be used by the main program and all subprograms in one region/partition. Otherwise, multiple copies of library subroutines may be resident at one time and cause unpredictable results.

User subprograms that are to be invoked at object time with the dynamic CALL statement must be members of the system link library or of a user-supplied private library.

In the sections that follow, the language for both the static and the dynamic CALL statement is described. The CANCEL statement, which functions only for programs that have been dynamically called, is also described.

CALL Statement

The CALL statement permits communication between a COBOL object program and one or more COBOL subprograms or other language subprograms.

Format 1

```
CALL  literal-1 [USING identifier-2 [identifier-3] ... ]
```

Format 2

```
CALL  identifier-1 [USING identifier-2 [identifier-3] ... ]
```

Literal-1 must be a nonnumeric literal.

Identifier-1 must be defined in such a way that its value can be a program-name.

Literal-1 or the contents of identifier-1 must conform to the rules for the formation of a program-name. The first eight characters of program-name are used as the identifying name of the program and should therefore be unique. Since the system expects the first character of program-name to be alphabetic, the first character, if it is numeric, is converted as follows:

0	to	J
1 through 9	to	A through I

Since the system does not allow the hyphen as a valid character, the hyphen is converted to zero if it appears as the second through eighth character of program-name.

Literal-1 or the contents of identifier-1 must be the name of the program that is being called, or a name of an entry point in the called program. The program in which the CALL statement appears is the calling program. The first eight characters of literal-1 or identifier-1 are used to make the correspondence between the calling program and the called program.

When the called program is to be entered at the beginning of the Procedure Division, literal-1 or identifier-1 must specify the program-name (in the PROGRAM-ID paragraph) of the called program. The called program must have a USING clause as part of its Procedure Division header if there is a USING clause in the CALL statement which invoked it.

When the called program is to be entered at entry points other than the beginning of the Procedure Division, these alternate entry points are identified by an ENTRY statement and a USING option corresponding to the USING option of the invoking CALL statement. In the case of a CALL statement with a corresponding ENTRY, literal-1 or identifier-1 must be a name other than the program-name, but they follow the same rules as those for formation of a program-name.

Identifier-2, identifier-3, etc., specified in the USING option of the CALL statement indicate those data items available to a calling program that may be referred to in a called program. When the called subprogram is a COBOL program, each of the operands in the USING option of the calling program must be defined as a data item in the File Section, Working-Storage Section, Linkage Section, or Communication Section. If the called program is written in a language other than COBOL, the operands of the USING option may additionally be a file-name or a procedure-name. If the operand of the USING option is a file-name, the associated file must be opened in the calling program.

Names in the two USING lists (that of the CALL in the main program and that of the Procedure Division header or of the ENTRY statement in the called program) are paired in a one-to-one correspondence.

There is no necessary relationship between the actual names used for such paired names, but the data descriptions must be equivalent. When a group data item is named in the USING list of a Procedure Division header or of an ENTRY statement, names subordinate to it in the subprogram's Linkage Section may be employed in subsequent subprogram procedural statements.

When group items with level numbers other than 01 are specified, proper word-boundary alignment is required if subordinate items are described as COMPUTATIONAL, COMPUTATIONAL-1, or COMPUTATIONAL-2.

The USING option should be included in the CALL statement only if there is a USING option in the called entry point, which is either included in the Procedure Division header or in an ENTRY statement in the called program. The number of operands in the USING option of the CALL statement should be the same as the number of operands in the USING option of the Procedure Division header or ENTRY statement. If the number of operands in the USING option of the CALL statement is greater than the number in the USING option of the called program, only those specified in the USING option of the called program may be referred to by the called program.

Called programs may contain CALL statements. However, a called program must not contain a CALL statement that directly or indirectly calls the calling program. If it does, the run unit is terminated.

A called program may not be segmented.

Format 1: When the literal-1 option is specified, then, depending on the PARM field of the EXEC job control statement, the CALL statement may be either static or dynamic.

If PARM=NODYNAM is specified in the EXEC statement, then the CALL literal-1 statement is static, and the following considerations apply:

- The programmer may specify literal-1 as a program-name or as an alternate entry point, in any order.
- The first time a called program is entered, its state is that of a fresh copy of the program. Each subsequent time the program is entered, the state is as it was upon the last exit from the program. Thus, the reinitialization of the following items is the responsibility of the user:

  GO TO statements that have been altered
  TALLY
  data items
  ON statements
  PERFORM statements
  EXHIBIT CHANGED statements
  EXHIBIT CHANGED NAMED statements

  (EXHIBIT CHANGED and EXHIBIT CHANGED NAMED operands are compared with the value of the item at the time of its last execution, whether or not that execution was during another CALL to this program. If a branch is made out of a PERFORM statement, after which an exit is made from the program, the range of that PERFORM is still in effect upon a subsequent entry.)
- The CANCEL literal statement may not be specified in this case. The CANCEL identifier statement is accepted; however, the compiler then options the PARM=RESIDENT parameter.

If PARM=DYNAM is specified in the EXEC job control statement, then the CALL literal-1 statement is dynamic, and the following considerations apply:

- A called program is in its initial state the first time it is called within a run unit, and also the first time it is called after a CANCEL statement for the called program has been executed.
- On all other entries into the called program, the state of the called program remains unchanged from its state when last executed. Thus, for such entries, it is the user's responsibility to reinitialize certain items in the subprogram. The description of the static CALL literal statement gives the list of specific items.

- Differing entry points for one subprogram should not be specified unless an intervening CANCEL statement has been executed. (See note after the Format 2 description.)

  (For example, if subprogram A has been called using its program-name as the entry point, then until a CANCEL statement for subprogram A has been executed, subsequent CALL statements for subprogram A should all use the program-name as the entry point. After a CANCEL statement has been executed, however, some alternate entry point for subprogram A may then be specified. That entry point should be the one entry point specified until yet another CANCEL statement has been executed.)

- Names prefixed by ILBO cannot be used as names of called subprograms, or as names of alternate entry points.

Format 2: The CALL identifier-1 statement is always dynamic, even when PARM=NODYNAM is specified in the EXEC job control statement. The following considerations apply:

- A called program is in its initial state the first time it is called within a run unit, and also the first time it is called after a CANCEL statement for the called program has been executed.

- On all other entries into the called program, the state of the called program remains unchanged from its state when last executed. Thus, for such entries, it is the programmer's responsibility to reinitialize certain items in the subprogram. The description of the static CALL literal-1 statement gives the list of specific items.

- Differing entry points for one subprogram should not be specified unless an intervening CANCEL statement has been executed. (See Note at the end of this description.)

- Names prefixed by ILBO cannot be used as names of called subprograms, or as names of alternate entry points.

Note: Linking two load modules together results logically in a single program with a primary entry point and an alternate entry point, each with its own name. (Each name by which a subprogram is to be dynamically invoked must be known to the system; each such name must be specified in linkage editor control statements as either a NAME or an ALIAS of the load module containing the subprogram.) Only if user modules are link edited with the attribute of non-reentrant and non-serially-reusable will a CANCEL statement guarantee a fresh copy of the subprogram upon a subsequent CALL.

Static and dynamic CALL statements may both be specified in the same program. That is, when PARM=NODYNAM is specified, both the CALL literal-1 and CALL identifier-1 options may be used. The CALL literal-1 statement results, in this case, in the subprogram so invoked being link-edited with the main program into one load module. The CALL identifier-1 statement results in the dynamic invocation of a separate load module. When a dynamic CALL statement and a static CALL statement to the same subprogram are issued within one program, a second copy of the subprogram is loaded. Therefore, care must be used to avoid duplicate load modules.

CANCEL Statement

The CANCEL statement releases the main storage occupied by a called subprogram.

Format

```
CANCEL { literal-1    }  [ literal-2    ] ...
       { identifier-1 }  [ identifier-2 ]
```

Each literal specified in the statement must be a nonnumeric literal.

Each identifier specified in the statement must be defined in such a way that its value can be a program-name. (See the rules for program-name in "CALL Statement.")

Each literal or identifier specified in the CANCEL statement must be the same as the literal or identifier specified in the associated CALL statement(s).

The CANCEL literal statement is invalid in a program in which NODYNAM and NORESIDENT are either specified or implied. The CANCEL identifier statement is accepted under the same conditions, but the compiler then options the RESIDENT parameter, and issues a warning message to that effect.

Subsequent to the execution of a CANCEL statement, the program referred to therein ceases to have any logical relationship to the program in which the CANCEL statement appears. A subsequently executed CALL statement by any program in the run unit naming the same program will result in that program being entered in its initial state.

A logical relationship to a cancelled subprogram is established only by execution of a subsequent CALL statement.

A called subprogram is cancelled either by being directly referred to as the operand of a CANCEL statement or by the termination of the run unit of which the program is a member.

No action is taken when a CANCEL statement is executed naming a program that has not been called in this run unit or has been called and is at present cancelled. Control passes to the next statement.

To guarantee the proper execution of the CANCEL statement for a subprogram, then prior to the execution of the CANCEL statement, every CALL statement for that subprogram should name the same entry point. Following the execution of a CANCEL statement, a CALL statement may specify a different entry point.

Called subprograms may contain CANCEL statements. However, a called subprogram must not contain a CANCEL statement that directly or indirectly cancels the calling program itself, or any other program higher than itself in the calling hierarchy. In such a case, the run unit is terminated.

A program named in a CANCEL statement must not refer to any program that has been called and has not yet executed an EXIT PROGRAM or GOBACK statement. A program may, however, CANCEL a program that it did not call, providing that in the calling hierarchy it is higher than or equal to the program it is cancelling. For example, A calls B, and B calls C; when A receives control it can cancel C; or A calls B, and A calls C; when C receives control it can then cancel B.

ENTRY Statement

The ENTRY statement establishes an entry point in a COBOL subprogram.

Format

```
ENTRY  literal-1 [USING  identifier-1 [identifier-2] ... ]
```

Control is transferred to the entry point by a CALL statement in an invoking program.

Literal-1 must be a nonnumeric literal. It must not be the name of the called program, but it is formed according to the rules followed for program-names.

Literal-1 must not be the name of any other entry point or program-name in the run unit.

A called program, once invoked, is entered at that ENTRY point whose operand literal-1 is the same as the literal-1 or identifier-1 specified in the CALL statement that invoked it.

USING Option

The USING option makes data items defined in the calling program available in a called program.

The USING option may also be used at execution time to pass parameters from the EXEC job control statement to a main COBOL program.

The USING option may be specified in a CALL statement, an ENTRY statement, or in the Procedure Division header. The three uses are shown in the following formats:

Format 1 (Within a Calling Program)

```
CALL { literal-1    } [USING  identifier-2 [identifier-3] ... ]
     { identifier-1 }
```

Format 2 (Within a Called Program)
Option 1 ENTRY literal-1 [USING identifier-2 [identifier-3] ...] Option 2 PROCEDURE DIVISION [USING identifier-2 [identifier-3] ...].

When the USING option is specified in the CALL statement, it must appear on either the Procedure Division header of the called program, or in an ENTRY statement in the called program.

The USING option may be present in the Procedure Division header or ENTRY statement if the object program is to function under the control of a CALL statement that contains a USING option. It may also be present on the Procedure Division header when information is to be passed from the EXEC job control statement to the main COBOL program.

The number of operands in the USING option of a called program must be less than or equal to the number of operands in the corresponding CALL statement of the invoking program.

When a called program has a USING option on its Procedure Division header and linkage was effected by a CALL statement, where literal-1 or the contents of identifier-1 is the name of the called program, execution of the called program begins with the first instruction in the Procedure Division after the declaratives section.

When linkage to a called program is effected by a CALL statement, where literal-1 or the contents of identifier-1 is the name of an entry point specified in an ENTRY statement of the called program, that execution of the called program begins with the first statement following the ENTRY statement.

When the USING option is present, the object program operates as if each occurrence of identifier-2, identifier-3, etc., in the Procedure Division had been replaced by the corresponding identifier from the USING option in the calling program CALL statement. That is, corresponding identifiers refer to a single set of data available to the calling program. The correspondence is positional, and not by name.

At execution time, the USING option may be used to pass parameters from the EXEC job control statement to the main COBOL program. In this case, a USING option on the Procedure Division header of a main program may contain identifier-2 as its only operand. Information from the PARM field of the EXEC job control statement is then available in the Linkage Section at the location specified as identifier-2. The first two bytes of identifier-2 contain a count (in binary) of the number of bytes of information in the PARM field, and are set to zero if the PARM field was omitted. The 2-byte field should be described as PIC S9(4) COMP. Immediately following these two bytes is the information in the PARM field.

Each of the operands in the USING option of the Procedure Division header or of the ENTRY statement must have been defined as a data item in the Linkage Section of the program in which this header or ENTRY statement occurs, and must have a level number of 01 or 77. Since the compiler assumes that each level-01 item is aligned on a doubleword boundary, it is the user's responsibility to ensure proper alignment.

The following example shows a program using Format 1 of the CALL statement with the USING option (PARM=NODYNAM has been specified):

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLSTAT.
     .
     .
DATA DIVISION.
     .
     .
WORKING-STORAGE SECTION.
01  RECORD-1.
    05  SALARY    PICTURE S9(5)V99.
    05  RATE      PICTURE S9V99.
    05  HOURS     PICTURE S99V9.
     .
     .
PROCEDURE DIVISION.
     .
     .
    CALL "SUBPROG" USING RECORD-1.
     .
     .
    CALL "PAYMASTR" USING RECORD-1.
     .
     .
    STOP RUN.
```

The following example shows a program achieving the same results with Format 2 -- the CALL identifier-1 option:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLDYNA.
     .
     .
DATA DIVISION.
     .
     .
WORKING-STORAGE SECTION.
77  IDENT  PICTURE X(8).
     .
     .
01  RECORD-1.
    05  SALARY    PICTURE S9(5)V99.
    05  RATE      PICTURE S9V99.
    05  HOURS     PICTURE S99V9.
     .
     .
PROCEDURE DIVISION.
     .
     .
    MOVE "SUBPROG" TO IDENT .
    CALL IDENT USING RECORD-1.
     .
     .
    CANCEL IDENT.
     .
     .
    MOVE "PAYMASTR" TO IDENT.
    CALL IDENT USING RECORD-1.
     .
     .
    STOP RUN.
```

The following is an example of a called subprogram which can be associated with either of the preceding calling programs:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBPROG.
     .
     .
DATA DIVISION.
     .
     .
LINKAGE SECTION.
01  PAYREC.
    10  PAY           PICTURE S9(5)V99.
    10  HOURLY-RATE   PICTURE S9V99.
    10  HOURS         PICTURE S99V9.
     .
     .
PROCEDURE DIVISION USING PAYREC.
     .
     .
    GOBACK.
    ENTRY "PAYMASTR" USING PAYREC.
     .
     .
    GOBACK.
```

Processing begins in the calling program -- which may be either CALLSTAT or CALLDYNA. When the first CALL statement is executed, control is transferred to the first statement of the Procedure Division in SUBPROG, which is the called program.

Note that in each of the calling programs the operand of the USING option is identified as RECORD-1.

When SUBPROG receives control, the values within RECORD-1 are made available to SUBPROG; in SUBPROG, however, they are referred to as PAYREC. Note that the PICTURE descriptions of the subfields within PAYREC (described in the Linkage Section of SUBPROG) are the same as those for RECORD-1.

When processing within SUBPROG reaches the first GOBACK statement, control is returned to the calling program. Processing continues in that program until the second CALL statement is issued.

Note that in CALLSTAT (statically linked) that the CANCEL statement is not valid. In CALLDYNA, however, since the second CALL statement refers to another entry point within SUBPROG, a CANCEL statement is issued before the second CALL statement.

With the second CALL statement in the calling program, control is again transferred to SUBPROG, but this time processing begins at the statement following the ENTRY statement in SUBPROG. The values within RECORD-1 are again made available to SUBPROG through the matching USING operand PAYREC. When processing reaches the second GOBACK statement, control is returned to the calling program at the statement immediately following the second CALL statement.

In any given execution of these two programs, if the values within RECORD-1 are changed between the time of the first CALL and the second, the values passed at the time of the second CALL statement will be the changed, not the original, values. If the user wishes to use the original values, then he must ensure that they have been saved.

With the Version 4 Compiler, the user can request a syntax-checking compilation. Through the PARM field of the EXEC control statement, such compilations can be requested unconditionally or conditionally.

When unconditional syntax-checking is requested, the compiler scans the source text for syntax errors, and generates the appropriate error messages, but does not generate object text.

When conditional syntax-checking is requested, the compiler scans the source text for syntax errors, and generates the appropriate error messages. If no message exceeds the W or C level, a full compilation is produced. If one or more E-level or D-level messages are produced, the compiler generates the messages, but does not generate object text.

(A few syntax errors may not be detected when a syntax-checking compilation is requested. When the compiler is released, a list of such errors will be made available.)

Syntax-only compilation has the capability of considerably reducing compile time. Unconditional syntax checking can reduce compilation time more than conditional syntax checking.

When unconditional syntax checking is specified, all of the following compile time options, if specified, are suppressed:

LOAD	NOSUPMAP	FLOW
XREF	PMAP	STATE
SXREF	DECK	NAME
CLIST	TRUNC	

If optimized object code is requested, and unconditional syntax checking is also requested, the object code is not produced. If symbolic debugging is requested and unconditional syntax checking is also requested, the symbolic debugging option is suppressed.

When conditional syntax checking is requested, the preceding options are suppressed only if one or more E-level or D-level messages are generated.

Unconditional syntax checking is assumed if all of the following compile-time options are specified:

NOLOAD	NOCLIST	SUPMAP
NOXREF/NOSXREF	NOPMAP	NODECK

If neither unconditional nor conditional syntax checking is specified, or if unconditional syntax checking is not assumed, a full compilation -- including error messages, object text, and all other specified (or default) options -- is produced.

APPENDIX A: VERSION 4 OBJECT-TIME SUBROUTINE LIBRARY

The Version 4 Object-time Subroutine Library is a partitioned data set residing on a direct-access device, and contains the COBOL library subroutines in load module form. The Version 4 Object-time Subroutine Library is designed for use under the Operating System with object modules produced by the Version 4 Compiler. The Version 4 Object-time Subroutine Library is also being made available as a separate Program Product.

COBOL library subroutines perform execution-time operations requiring either repetitive or extensive coding. It is inefficient to place such coding inline in the object module each time it is needed. Instead, library subroutines are used to reduce the size of the object module. Any library subroutines required to execute the problem program are either combined with the object module at link-edit time or are dynamically fetched during program execution.

To save even more main storage space, the Version 4 COBOL Library Management Facility allows a single copy of such COBOL object-time subroutines to be shared by problem programs in different partitions or regions. This is controlled by the user through a compile-time option. See the chapter on the COBOL Library Management Facility.

There are several major categories into which the object-time subroutine library can be classified:

- Input/Output routines
- Conversion routines
- Arithmetic verb routines
- Sort feature interface routines
- Checkpoint/Restart routines
- Segmentation feature routines
- Teleprocessing routines
- Debugging routines
- Other verb routines

The Version 4 Object-time Subroutine Library contains all subroutines needed to support the new features of the Version 4 Compiler.

APPENDIX B: VERSION 4 CHANGES IN THE COBOL RESERVED WORD LIST

For Version 4, the following entries in the reserved word list must be changed to appear as shown; the keys preceding the entries, and their meanings, are:

(xa) before a word means that the word is an extension to American National Standard COBOL.

(ca) before a word means that the word is a CODASYL COBOL reserved word not incorporated in American National Standard COBOL or in IBM American National Standard COBOL.

(xac) before a word means that the word is an IBM extension to both American National Standard COBOL and CODASYL COBOL.

(xa)	CANCEL	(xa)	QUEUE
(xa)	CD	(xa)	RECEIVE
(xa)	COUNT	(xac)	RELOAD
(xa)	DATE	(xa)	SEGMENT
(xa)	DELIMITED	(xa)	SEND
(xa)	DELIMITER	(xac)	SERVICE
(xa)	DEPTH		STATUS
(xa)	DESTINATION	(xa)	STRING
(ca)	DISABLE	(xa)	SUB-QUEUE-1
(xa)	EMI	(xa)	SUB-QUEUE-2
(ca)	ENABLE	(xa)	SUB-QUEUE-3
(xa)	ESI	(xa)	SYMBOLIC
(xa)	ETI	(ca)	TABLE
(ca)	INITIAL	(ca)	TERMINAL
(xa)	LENGTH	(xa)	TEXT
(xa)	MESSAGE	(xa)	TIME
(xa)	OVERFLOW	(xa)	UNSTRING
(xa)	POINTER		

The IBM 3505 card reader and the 3525 card punch are 80-column devices that offer more flexible processing capabilities than former card devices. The 3505 card reader can be used for sequential reading; it can also be used for Optical Mark Read (OMR) processing. Both the 3505 and the 3525 support Read Column Eliminate (RCE) processing. The 3525 card punch, when equipped with appropriate special features, can be used separately as a card reader, as a card punch, as an interpreting card punch, and as a printer (either 2-line or multiline printing is available); in addition, the read, punch, and print functions (any two or all three) can be combined, so that those functions specified are all performed during one pass of a card through the device.

Note: The interpreting card punch is considered one function. It cannot be combined with the other functions, but is specified through the DD statement for the data set.

The processing functions are all specified through new parameters of the DD statement. For OMR and RCE processing, format descriptor card(s) must also be included as the first card(s) of the data set. (For OMR processing, the format descriptor specifies those columns that are optically marked; for RCE processing, the format descriptor specifies those columns that are to be ignored.) Detailed information on these considerations is given in the publication IBM System/360 Planning Guide for IBM 3505 Card Reader and IBM 3525 Card Punch On System/370, Order No. GC21-5027.

The following paragraphs describe the special COBOL programming considerations when these devices are used.

3505 OMR PROCESSING

Function-names S01 and S02 in the SPECIAL-NAMES Paragraph may be used to select logical stacker 1 or logical stacker 2 under program control.

When stacker selection is specified, RESERVE NO ALTERNATE AREAS must also be specified.

If the user wishes to inspect the substitution character (hexadecimal "3F") placed in column 80 by the system for a defective optically marked card, he must specify a record description of 80 characters. (Note that the "3F" is place in both card column 80 and the defective (unreadable) card column.

3505/3525 RCE PROCESSING

Function-names S01 and S02 in the SPECIAL-NAMES Paragraph may be used to select logical stacker 1 or logical stacker 2 under program control.

When stacker selection is specified, RESERVE NO ALTERNATE AREAS must also be specified.

When RCE processing is specified for input, the user must not refer to the ignored columns (as specified by the format descriptor) or results are unpredictable.

When RCE processing is specified for output, any data in the COBOL record that corresponds to the ignored columns (as specified by the format descriptor) is not punched or printed.

3525 COMBINED FUNCTION PROCESSING

COBOL handles each of the separate functions to be combined as a separate logical file. Each such logical file has its own file structure and procedural processing requirements. However, because such combined function files refer to one physical unit, the user must observe certain restrictions during processing. The following sections explain the programming requirements for combined function processing in OS American National Standard COBOL.

The COBOL language does not define the files as being combined function files; instead, the combined functions are specified through new parameters for the files' DD statements. (In this way, the user can, if he so desires, process the same COBOL files as completely separate read, punch, and print files.)

I -- ENVIRONMENT DIVISION CONSIDERATIONS

For each function, there must be a separate SELECT sentence written in the Environment Division. Each read function file and each punch function file must specify RESERVE NO ALTERNATE AREA(S).

SPECIAL-NAMES Paragraph

If stacker selection of punched output, or line control of printed output is desired, mnemonic-names for the purpose can be specified in the SPECIAL-NAMES Paragraph. The mnemonic-names may be equated with the following function-names:

Function-name	Meaning
S01	Stacker 1
S02	Stacker 2
C01	Line 1
C02	Line 3
C03	Line 5
.	.
.	.
.	.
C12	Line 23

II -- DATA DIVISION CONSIDERATIONS

For each logical file defined in the Environment Division for the combined function structure, there must be a corresponding FD entry and 01 record description entry in the File Section of the Data Division.

III -- PROCEDURE DIVISION CONSIDERATIONS

Input/output operations must proceed in a specified order in the Procedure Division. In the 3525 device, the card passes first through the reading station, next through the punching station, and last through the printing station. Therefore, the following combined functions may be specified, but only in the order shown:

Functions to be Combined	Order of Operations	Associated COBOL Statement
read/punch/print	read	READ ... AT END
	punch	WRITE ... ADVANCING/POSITIONING
	[print]	WRITE ... ADVANCING/POSITIONING
read/punch	read	READ ... AT END
	punch	WRITE ... ADVANCING/POSITIONING
read/print	read	READ ... AT END
	[print]	WRITE ... ADVANCING/POSITIONING
punch/print	punch	WRITE ... ADVANCING/POSITIONING
	[print]	WRITE ... ADVANCING/POSITIONING

All required operations on one card must be completed before the next card is obtained, or there is an abnormal termination of the job.

The following Procedure Division considerations in the COBOL source program apply:

OPEN Statement

For any specified function, an OPEN statement must be issued before the input/output operation for that function is attempted. The following additional considerations apply:

- For the read function, the file must be opened INPUT.
- For the punch function and print function, the file must be opened OUTPUT.

WRITE Statement -- Punch Function Files

If the user wishes to punch additional data into some of the cards and not into others, he must issue a dummy WRITE statement for the null cards, first filling the output area with SPACEs.

If stacker selection for the punch function file is desired, the user can specify S01 (for stacker one) and S02 (for stacker two) as function-names in the SPECIAL-NAMES Paragraph. He can then issue WRITE ADVANCING statements using the associated mnemonic-names. Stacker selection may be specified only for the punch function file.

WRITE Statement -- Print Function Files

If the user wishes to print additional data on some of the data cards and not on others, he may omit the WRITE statement for the null cards.

Depending on the capabilities of the specific model in use, the print file may be either a 2-line print file or a multiline print file. Up to 64 characters may be printed on each line.

For a 2-line print file, the lines are printed on line 1 (top edge of card) and line 3 (between rows 11 and 12).

For a multiline print file up to 25 lines of characters may be printed.

If line control is not specified, then automatic single spacing is provided. Any attempt to write beyond the limits of the card results in abnormal termination of the job.

Line control is specified by issuing WRITE BEFORE/AFTER ADVANCING statements, or WRITE AFTER POSITIONING statements for the print function file. If line control is used for one such statement, it must be used for all other WRITE statements issued to the file. The maximum number of printable characters, including any SPACE characters, is 64. The first character of the record defined must be reserved by the programmer for the line control character; therefore, the record may be defined as containing up to 65 characters.

Identifier and integer have the same meanings they have for other WRITE ADVANCING or WRITE POSITIONING statements. However, such WRITE statements must not increase the line position on the card beyond the card limits, or abnormal termination results.

The mnemonic-name of the WRITE ADVANCING or WRITE POSITIONING statement may also be specified. In the SPECIAL-NAMES Paragraph, the following function-names may be associated with the mnemonic-names:

Function-name	Meaning
C01	Line 1
C02	Line 3
C03	Line 5
.	.
.	.
.	.
C12	Line 23

(Note that for a 2-line print file, only C01 and C02 are valid as function-names.)

CLOSE Statement

When processing is completed, a CLOSE statement must be issued for each of the combined function files. After a CLOSE statement has been issued for any one of the functions, an attempt to perform processing for any of the functions results in abnormal termination.

For Version 4, the following additions to the glossary will be made.

Communication Description: An implicitly defined fixed-format storage area that serves as the interface between the COBOL object program and the Message Control Program (MCP). It is specified in the Communication Section.

Communication Description Entry: An entry in the Communication Section of the Data Division that is composed of the level indicator CD, followed by a cd-name, and then optionally followed by a set of independent clauses. It describes the interface between the MCP and the COBOL object program.

Communication Section: The section in the Data Division that describes the interface area between the MCP and the COBOL program. It is composed of one or more CD description entries that define the fields in the interface area.

Communications Device: a mechanism (hardware or hardware-software) capable of sending data to a queue and/or receiving data from a queue. This mechanism may be a computer or a peripheral device. One or more programs containing Communication Description entries and residing within the same computer define one or more of these mechanisms.

Delimiter: A character or sequence of contiguous characters that identify the end of a string of characters and that separate the string of characters from the following string of characters. A delimiter is not part of the string of characters that it delimits.

Destination: The symbolic identification of the receiver of a transmission (i.e., a message) from a queue.

Destination Queue: An MCP storage queue for one or more messages from one or more remote stations or to one or more remote stations. Destination queues serve as buffers between a COBOL Teleprocessing program and the remote stations.

Input Queue: An MCP destination queue from which the COBOL Teleprocessing program accepts messages from the remote stations.

Line-Control Cards: A set of control cards that request a symbolically formatted dump just before execution of user-selected COBOL statements. The format of the dump, and the number of times it is to be created can also be controlled by the user.

Message: A string of characters associated with an end-of-message indicator or end-of-transmission indicator. A message may consist of one or more related message segments.

Message Control Program (MCP): A TCAM communications control program that supports the processing of messages.

Message Indicators: Three message indicators are allowed in a COBOL program. Each signals that some specific condition exists:

ETI	Indicates logical end-of-transmission of a group of messages
EMI	Indicates end-of-message
ESI	Indicates end-of-segment

The hierarchy of message indicators is in the order of the preceding list. Within this hierarchy an ETI is conceptually equivalent to an ETI, EMI, and ESI; an EMI is conceptually equivalent to an EMI and an ESI. Thus, a segment may be terminated by an ESI, EMI, or ETI, and a message may be terminated by an EMI or ETI.

Message Segment: A string of characters that forms a logical subdivision of a message, and is normally associated with an end-of-segment indicator. A message segment is the equivalent of a TCAM record. (See "Message Indicators.")

Nonswitched Line: A line that is a continuous link between a remote station and the computer. It may connect the central computer with either a single station or more than one station.

Output Queue: An MCP destination queue into which the COBOL Teleprocessing program places messages for one or more remote stations.

Overflow Condition: In string manipulation, a condition that occurs when the sending area(s) contain untransferred characters after the receiving area(s) have been filled.

Program-Control Cards: A set of control cards that at object time requests a symbolically formatted abnormal termination dump, and optionally outlines the scope of operations of the symbolic debugging feature.

Queue: A logical collection of messages awaiting transmission or processing.

Queue Blocks: Blocks containing status and control information pertaining to the message being processed and to each active queue. Created when a queue is first accessed by a COBOL Teleprocessing run unit, all queue blocks in one region/partition are chained to each other.

Queue Name: A symbolic name that indicates to the MCP the logical path by which a message, or portion of a completed message, may be accessible in a queue. (The first eight characters must match the DDname of the DD statement that specifies the queue.)

Remote Station: A control unit and one or more input/output devices connected to the central computer through common carrier facilities. A remote station may be a terminal device or it may be another computer.

Source: The symbolic identification of the originator of a transmission to a queue.

Switched Line: In Teleprocessing, a communication line for which no single continuous path between the central computer and the remote station exists. Several alternative paths are available for transmission; the common carrier switching equipment selects the path. The remote station is continuously connected to the switching center by an access line associated with a specific telephone number.

(Where more than one page reference is given, the major reference is first.)

READER'S COMMENTS

TITLE: IBM OS Full American National Standard COBOL Compiler and Library, Version 4, Planning Guide

ORDER NO. GC28-6431-0

Your comments assist us in improving the usefulness of our publications; they are an important part of the input used in preparing updates to the publications. All comments and suggestions become the property of IBM.

Please do not use this form for technical questions about the system or for requests for additional publications; this only delays the response. Instead, direct your inquiries or requests to your IBM representative or to the IBM Branch Office serving your locality.

Corrections or clarifications needed:

Page *Comment*

Please include your name and address in the space below if you wish a reply.

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

GC28-6431-0

cut along this line

fold fold

FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION
1271 Avenue of the Americas
New York, New York 10020

Attention: PUBLICATIONS

fold fold

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

GC28-6431-0

IBM OS Full ANS COBOL V4 Pl. Guide Printed in U.S.A. GC28-6431-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

GC28-6431-0

IBM OS Full ANS COBOL V4 Pl. Guide Printed in U.S.A. GC28-6431-0

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)