File No. S360-29
GY28-2051-0

Program Logic

IBM System/360 Time Sharing System
PL/I Compiler

This publication describes the internal logic of the IBM System/360 Time Sharing System PL/I Compiler.

Program Logic Manuals are intended for use by IBM customer engineers involved in altering program design. It can be used to locate specific areas of the program, and it enables the reader to relate these areas to the corresponding program listings. Program logic information is not necessary for program operation and use.

PREFACE

This publication provides customer engineers and other technical personnel with information describing the internal organization and logic of the TSS/360 PL/I compiler. The material is divided into four sections and nine appendixes.

Section 1 describes the overall organization of the compiler and the relationship between the compiler and the time sharing system.

Section 2 contains a general description of each logical phase of the compiler, followed by descriptions of the physical phases contained within each logical phase. Descriptions of the control modules and of the interfaces between the compiler and the time sharing system are also included.

Section 3 consists of flowcharts, tables and routine directories. The flowcharts show the relationship between the routines of each phase, while the tables and directories list the routines and their functions.

Section 4 contains the layouts of tables used by the compiler, as well as formats of text and dictionary entries.

The appendixes contain supplementary material for references purposes.

PREREQUISITE PUBLICATIONS

Effective use of this manual requires knowledge of the information contained in the following manuals:

IBM System/360 Time Sharing System:

- Concepts and Facilities, Order No. GC28-2003
- PL/I Language Reference Manual, Order No. GC28-2045
- System Logic Summary PLM, Order No. GY28-2009

In addition, the following publications are recommended as supplemental reading:

IBM System/360 Time Sharing System:

- PL/I Programmer's Guide, Order No. GC28-2049
- PL/I Subroutine Library PLM, Order No. GY28-2052
- Dynamic Loader PLM, Order No. GY28-2031

First Edition (June, 1970)

This edition is current with Version 7, Modification 0, and remains in effect for all subsequent versions or modifications of IBM System/360 Time Sharing System unless otherwise indicated. Significant changes or additions to this publication will be provided in new editions or in Technical Newsletters.

Before using this publication in connection with the operation of IBM systems, refer to the latest edition of IBM System/360 Time Sharing System: Addendum, Order No. GC28-2043, for the editions of publications that are applicable and current.

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

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Corporation, Time Sharing System/360 Programming Publications, Department 643, Neighborhood Road, Kingston, New York. 12401

Contents

Figures

PURPOSE OF THE COMPILER

The TSS/360 PL/I compiler analyzes and processes source programs that are written in PL/I and translates them into object data sets. These object data sets contain code that is not suitable for execution by TSS/360. Therefore an additional processor, the object data set converter (ODC), converts these object data sets to TSS/360-executable code.

Usual output from the compiler consists of a load data set and a list data set, when these options have been specified by the user. A macro data set will also be produced when preprocessing is indicated (see "Preprocessing" in Section 2).

THE COMPILER IN THE TSS/360 SYSTEM ENVIRONMENT

The compiler consists of a series of logical phases that are under the supervision of compiler control routines; subroutines within these control routines provide whatever services the compiler requires during compilation. Communication between the compiler and TSS/360 is achieved through the program language controller (PLC), which is the interface with the system. PLC performs a series of functions for the compiler at various stages of compilation and, finally, calls the object data set converter (ODC) after compilation, to convert the object data set to TSS/360 code.

PLC -- Interface With the System

When the PL/I compiler is invoked, control is transferred to PLC. This module acts as a communications area for user-specified options, and controls the sequence of events during invocation of the PL/I compiler.

The PL/I compiler, unlike the TSS/360 Assembler and FORTRAN compiler, cannot function until the source data set has been fully entered. Therefore, when compilation is called for, PLC searches for an input data set. If the named data set does not exist, PLC invokes the text editor to create the PL/I source data set. When a source data set exists, control passes back to PLC, which then calls the PL/I compiler.

Depending upon user options specified, invocation of the PL/I compiler may cause PLC to act as interface for these functions:

- Creating a PL/I source data set (via the text editor).
- Converting separately created PL/I object data sets to TSS/360 code via ODC.
- Combining a list of PL/I object data sets for conversion to TSS/360-executable code.
- Performing multiple compilations within a single invocation of the PL/I compiler.
- Changing implicit calls to explicit calls via the name processor.
- Printing compiler-generated listings.

The program language controller (Figure 1) is a serially reentrant and sharable module containing recovery facilities used in case of interruptions. It can check, at any stage, the status of compilation; its recovery facilities permit compilation to proceed from the point of interruption or from the beginning.

PLC may be entered at five main entry points. Initial entry to PLC occurs when the PL/I compiler is invoked. Depending upon the options specified by the user, the text editor, compiler, ODC, and/or the name processor, may be called. PLC's additional entry points provide for entry to subroutines used to perform the specific functions for which PLC is responsible at various stages of compilation: entry from the text editor after creation of the source data set, entry after compilation is completed to build the MERGELST, an entry point for handling data management functions, and entry to the language processor early-end routine.

ODC -- Conversion of Object Code

The TSS/360 PL/I compiler produces code that is similar to OS/360 code. To transform the load data set, which is output from the compiler, into TSS/360 code, the object data set converter (ODC) resolves constants and reformats the module.

PLC invokes ODC after completion of the compilations specified with a given invocation of the PL/I compiler. ODC is called only once within an invocation, and then only if a merge list (MERGELST) or a merge data set (MERGEDS) has been specified in the options, or if the PL/I compiler has

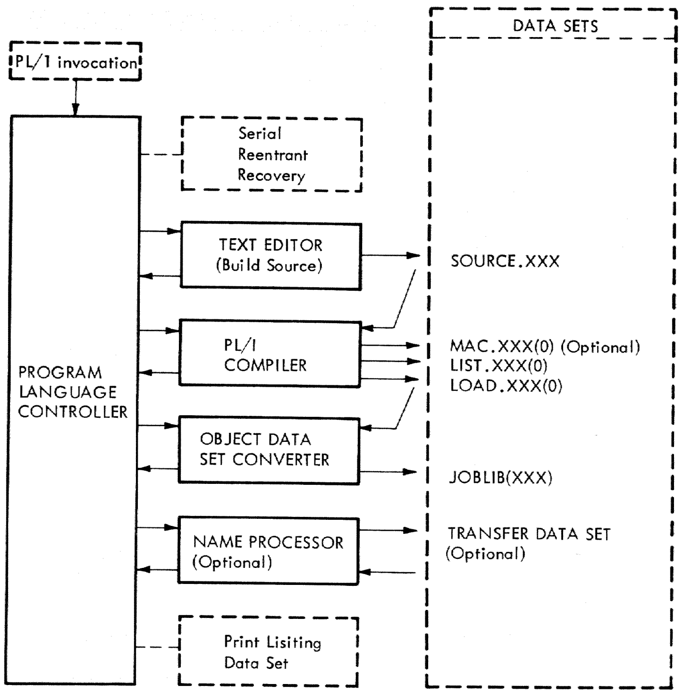


Figure 1. PLC - interface with TSS/360

created a merge list to accommodate a series of compilations. Input to ODC consists of the PL/I-compiled object data sets in card-image format (see Appendix J). Output consists of the executable program. ODC stores all of the TSS/360-executable programs from one invocation of the PL/I compiler as separate members in one job library, resolving standard QCONs (pseudo registers) and passing others to the dynamic loader. In addition, ODC packs CSECTs as specified by the default value PLIPACK.

Name Processor -- Conversion of Implicit Calls to Explicit Calls

The name processor is an optionally invoked routine that helps the user transform implicit calls to explicit calls. To do this, the name processor transforms external name references in the PMD of a module to new, unique names. In addition, for the new names to be connected with the subroutines that have the old names, the name processor optionally constructs or updates a line data set that is called a transfer data set and that has this format:

0-7	line number
8	X'00'
9-16	new name
17	blank
18-24	PLICALL
25-27	blank
28-35	old name

The user must supply his own PLICALL macro to perform the explicit calling or loading of the subroutines.

The name processor constructs or updates the transfer data set only if:

the user has read/write access to the transfer data set, and

the default value of UPDTXFER is set to Y, and

the EXPLICIT operand of the PLI command specifies names to be padded.

The new name is derived from the old name by adding a pad character ('@', or a different character that the user specifies with the default value PADCHAR) to the left of the old name.

PLC invokes the name processor after return of control by ODC, if the PLI command included an EXPLICIT or XFERDS operand. Input to the name processor includes the PL/I communications bucket (CHBPLI) and a table of converted modules to be checked for name transformation (CHBMGL).

ORGANIZATION OF THE COMPILER

The PL/I compiler comprises 12 logical phases, each of which consists of several physical phases, all under the control of, and serviced by, the compiler control routines. A compilation is initiated by loading the compiler control routines from SYSLIB. The control routines then carry out their own initialization and perform these functions:

Act as the interface between the compiler phases and TSS/360, controlling operations such as all input/output, storage allocation, program interruptions, and storage dumping.

Supervise the loading and linking of compiler phases in accordance with source program options.

Supervise all work space used by the compiler for information concerning the source program.

Provide a number of routines to assist in compiler debugging.

The entire PL/I compiler, including the control modules, is contained in six link-edited output modules (for contents of output modules, see Appendix K). When the user-specified compiler options are interpreted, it is determined which of these output modules is to be loaded. The addresses of the individual modules, in each of the loaded output modules, are then moved into a phase directory, and a request for the phases required is inserted in the status byte.

Data Sets Used by the PL/I Compiler

The source data set, which is input to the compiler, is given the name the user specifies, or SOURCE.XXX. The data sets that constitute possible output from the compiler are: a list data set, named LIST.XXX(0); a load data set, named LOAD. XXX(0); and a macro data set, named MAC. XXX(0). Table 1 contains the corresponding ddnames for each of the data sets used by the compiler. (Generally, ddnames will be used throughout this publication to refer to data sets used by the compiler).

The source program that is to be compiled appears as input to the compiler on the PLIINPUT data set. If one of the preprocessors is called prior to compilation, a macro data set is created with the ddname of PLIMAC. When preprocessing is completed, PLIMAC replaces PLIINPUT as input to the compiler. The PLILIST data set is opened by PLC unless the user specifies that a separate listing is unnecessary, in which case the listing is placed on SYSOUT and no record of it is retained in the system after printout. The PLILOAD data set, containing compiler output, and the PLIMAC data set, containing intermediate text, are optional and are opened by control routines in the compiler. The PLIINPUT data set is always used by the compiler, and is opened by PLC.

The data sets used in the compilation, and the overall data flow associated with a compilation, are illustrated in Figures 2 and 3.

Overview of Logical Phases

Control is passed between the phases of the compiler via the control routines. After each phase has been executed, a branch is executed to the control module, which selects (from its phase directory) the next phase to be executed. The compiler phases and their corresponding functions are shown in Figure 4.

Communication between the phases is implemented by the following:

1. The text string. At the start of the compilation, the text string is input text that is converted by the compile-time processor, if necessary, into a string that is PL/I source text. The characters in this string are translated into a code that is internal to the compiler. The phases of the compiler gradually process the text until it is in the final form of the object data set, which consist of machine instructions. The text-code bytes used for the compiler and the formats of statements at different stages of the compilation are in Section 4, under "Internal Formats of Text."

   The text is broken down into blocks; each block has a symbolic name that is independent of the physical location of the block in storage. Thus, the text blocks may be moved around in virtual storage under the supervision of the compiler control routines.

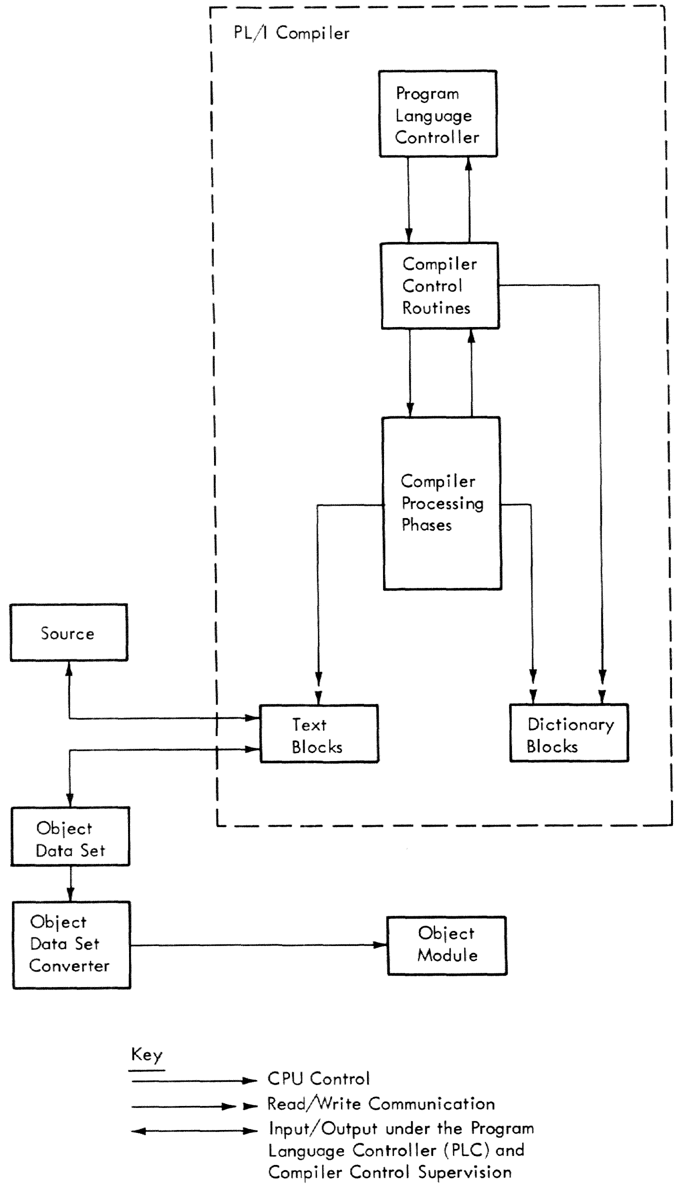


Figure 2. Compiler Organization and Control

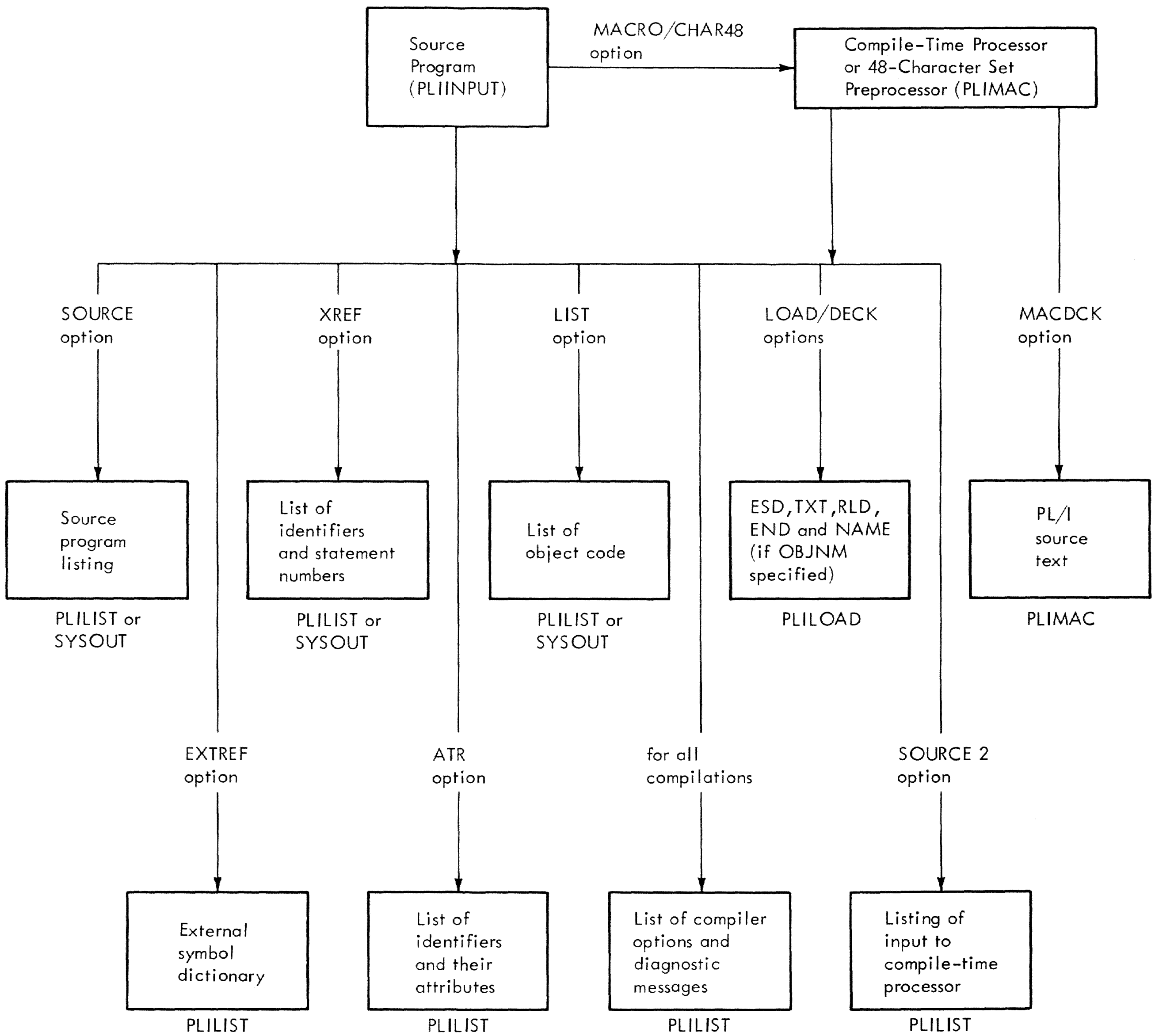


Figure 3. Compiler Data Flow and Data Sets Used

2. The dictionary. The dictionary consists of blocks, each of which has a symbolic name. Part of the first dictionary block is used as a communications region between phases (see Appendix B). The communications region contains such information as the addresses of the heads of chains and the symbolic start of text. The remainder of the dictionary contains all information relating to identifiers appearing in the program, such as temporary storage areas required. The format of all dictionary entries for the compiler are in Section 4, under "Internal Formats of Dictionary Entries."

Table 1. Data Sets Used by PL/I Compiler

DDNAME	DSNAME	Access Method	Comment
PLIINPUT or user-supplied $$$nnnn*	SOURCE.XXX or user-supplied	VISAM	Source input to compiler -- user-supplied or created by text editor before compilation is initiated
PLILIST	LIST.XXX(0)	VSAM	List data set -- built unless user options indicate none is necessary
PLILOAD	LOAD.XXX(0)	VSAM	Load data set -- output from compiler and input to ODC
PLIMAC	MAC.XXX(0) or user-supplied	VISAM	Intermediate source text -- created whenever preprocessing is specified
SYSULIB or user-supplied	USERLIB or user-supplied	VPAM/ VISAM	Member of library used by macro-phase %INCLUDE verb

*Name is generated by the system to be unique. The first three characters are "$$$" followed by a unique five-digit number.

Logical Phase	Function
Compile-time Processor	Reads input text, executes any compile-time statements in it, modifies text as directed, and produces modified text for further processing.
Read-In	Checks source-program syntax and removes from the test string all superfluous characters, such as comments and non-significant blanks
Dictionary	Removes all BCD identifiers and attribute declarations from the source string and replaces them with symbolic references to dictionary entries; entries contain all consistent declared attributes and all the attributes specified in language in default of source-program specifications; error messages are generated for all inconsistent attributes
Pretranslator	Processes features of language that are more easily processed in original PL/I form than when original syntactic form has been lost in later phases; carries out modifications that include rearranging of order of certain I/O statements, creation of temporary variables for procedure arguments that are expressions, conversion of array and structure assignments to a series of DO-loops surrounding scalar assignments, and removal of iSUB expressions
Translator	Converts original PL/I syntactic form to internal syntactic form ("triples"); triples consist of original source-program operators and operands, rearranged so that operations specified in source string may be carried out in proper order
Aggregates	Carries out all structure and array mapping, so that elements are aligned on correct virtual storage boundaries; when it is not possible to map at compilation time (such as when aggregates contain string lengths or array bounds that are specified by expressions) object code is produced to map at object time; also checks that items defined on arrays and structures can be mapped consistently

Figure 4. Compiler Logical Phases (Part 1 of 2)

Logical Phase	Function
Optimization	If requested, these phases attempt to reorder triples for subscript address calculations and generate efficient pseudo-code for DO-loop control; this enables some PL/I programs to compile into faster object code at cost of extra compile time
Pseudo-Code	Converts triples to form closely resembling machine instructions, in which registers are represented symbolically, and storage locations are represented by dictionary references with offsets; final version of text also contains special pseudo-code items for guidance of later phases
Storage Allocation	Searches dictionary for entries requiring storage, and allocates offsets to each, within its AUTOMATIC block or within STATIC storage area; code is compiled to set up dope vectors and pointers at object time for allocations of controlled variables and temporaries, storage for which must be obtained during execution of object program; prologue code is generated for each block of object program
Register Allocation	Allocates physical registers to symbolic registers that have been requested by earlier phases and ensures that all storage-location offsets allocated in previous phases can be addressed by insertion of necessary additional instructions
Final Assembly	Completes translation to machine-code instructions, by calculating branch-destination addresses inserted symbolically by earlier phases; loader text is produced for machine instructions, constants, INITIAL values in STATIC storage, and all constant data required for block initialization; external symbol dictionary (ESD) and relocation dictionary (RLD) are produced to enable object program to be converted by object data set converter (ODC); also produces listing of object code
Error Editor	Entered at end of every compilation; dictionary is examined to determine if diagnostic messages are to be printed out; if no, compilation is terminated by compiler control; if yes, error dictionary entries are processed and messages are printed; texts of all diagnostic messages are held in modules XG-YY.

Figure 4. Compiler Logical Phases (Part 2 of 2)

LOGIC OF THE COMPILER

The compiler modules are link edited into six output modules, which are broken down by function:

Control Output Module (CFBAC) - contains all the control modules, except those responsible for initialization. The code in this output module is reusable; it remains resident during multiple compilations.

Main Output Module (CFBAD) - contains the modules responsible for initialization, together with all the logical phases, except those responsible for preprocessing, optimization (option OPT=2), and interphase dumping and tracing.

First Preprocessing Output Module (CFBAE) - contains the modules required for macro and/or 48-character set preprocessing, with the exclusion of modules that are reused in the processing of the macro option.

Second Preprocessing Output Module (CFBAF) - contains those modules of the macro preprocessor that may be reused in the processing of the macro option.

Optimization Output Module (CFBAG) - contains those modules which are required when OPT=2 is specified by the user.

Interphase Dumping and Tracing Output Module (CFBAH) - contains all the modules required for interphase dumping and tracing.

Each of these output modules, with the exception of the control output module, contains a control CSECT made up of VCONs for each of the link-edited modules within it. The initialization and loading of the output modules is explained below. (For a list of the modules contained within each output module, see Appendix K.)

Compiler Control

The control-phase modules, which are resident in virtual memory throughout compilation, control these functions:

- Initialization and loading
- Character translation
- Communication between phases
- Scratch-storage control
- Text and dictionary block control
- Phase linkage
- Diagnostic-message control
- Input/output control
- Program-check handling
- Job termination

Initialization and Loading: The PL/I compiler is invoked by PLC via a CALL macro instruction issued to control module AA. This has the effect of loading the control output module (CFBAC). At the top of AA, a test is made to determine if this is a clean entry. If it is not, but is a reinvocation of the compiler, a cleanup routine is entered to ensure that all other output modules are deleted, that all open data sets are closed, and that any modified code in the control output module is initialized.

The main output module (CFBAD) is then loaded via the issuance of a LOAD macro instruction. Module AB of CFBAD is responsible for the detailed initialization of the compiler. A CSECT, AU, within module CFBAD contains a VCON for each of the modules within the main output module. AB is responsible for transferring these VCONS from AU to a list, called the phase directory, in module AA of the control output module (CFBAC). This list consists of 8-byte entries containing the addresses of modules in the compiler. Thus the phase directory, after initialization, will indicate the location in virtual memory of the individual compiler modules. If the user requires the interphase dumping and/or tracing routines, AB will load the output module containing these routines as part of its initialization responsibility.

When the detailed initialization of the compiler is complete, AB returns control to AA, where the linkage routines, using the phase directory, initiate execution of AM, the marking phase. Before marking the modules in the phase directory as wanted or not wanted, AM examines user-specified options and:

1. If MACRO and/or CHAR48 is specified, it loads the first preprocessor output module (CFBAE). Located in this output module is a CSECT, AW, which contains a VCON for each of the modules in that output module. These VCONS are then transferred to the relevant slot in the phase directory.

2. If MACRO has been specified, the second preprocessor output module (CFBAF) is loaded, whose VCON CSECT, AX, is used to fill the relevant slots in the phase directory.

3. If the user has specified OPT=2 as the level of optimization, then the optimization output module (CFBAG) is brought in and the phase directory filled from its VCON CSECT, AY.

Having completed initialization, AM passes control to the first logical phase. During compilation, additional modules may be marked as wanted or not wanted depending upon the nature of the source statements.

<u>Character Translation Tables</u>: The character translation tables (see "Internal Formats of Text" in Section 4) provide the facility for converting external code to compiler internal code and for converting the internal code back to the external form. These tables prevent the compiler from becoming character code dependent and enable the scanning routines to process the input source statements more efficiently. Note that the contents of these tables are different during compile-time processing from the contents during compilation.

<u>Communication Between Phases</u>: The communications region is an area, specified by the control routines, and used to communicate necessary information between two phases of the compiler. The communications region is resident in the first dictionary block throughout the compilation.

Entry to the various compiler control routines is via a transfer vector. Details of the transfer vector and the organization of the communications region are in Appendix B. (Note: The use of the communications region during compile-time processing is described in Appendix F.)

<u>Scratch-Storage Control</u>: Scratch storage of 4096 bytes is guaranteed to all phases. The control routines split the 4096-byte block into discrete sections, and allocate them as required. The sections are in multiples of 512 bytes. Additional scratch storage is obtained as required.

<u>Text and Dictionary Block Control</u>: During compilation, at least four text blocks and four dictionary blocks are available. The dictionary- and text-block size is four pages. Block control is achieved by a system of text and dictionary references.

<u>Phase Linkage</u>: The phase directory, in module AA, is constructed so that it may contain the location in virtual memory of each module required for compilation. These modules are then marked during initialization, by AM, or during compilation, as "wanted" or "not wanted" for that compilation. The phase-linkage routines, also in AA, are then used to access the phase directory, where they pick up the address of the next required module. This may be specified explicitly, or it may be the next phase after the current one that is marked "wanted". Having picked up the address from the directory, the linkage routines may either return the address to the caller in a communications area or they may branch directly to the address, to commence execution of a new module. Which of the above operations takes place is dependent upon the entry point used to enter the linkage routine.

Where preprocessing is requested, the modules in the second preprocessor output module (CFBAF) may be required for reuse. Since these modules are not serially reusable, the output module must be deleted and reloaded each time it is required. This service is performed by the linkage routines.

<u>Diagnostic-Message Control</u>: Diagnostic message-control routines cause diagnostic messages to be placed in a chain in the dictionary. When conversational diagnostics are specified, these will also be produced by these routines.

<u>Input/Output Control</u>: The I/O control routines involved act as interfaces between the compiler phases and the PLIINPUT, PLIMAC, PLILIST, and PLILOAD data sets (see Figure 5).

<u>Program-Check Handling</u>: The compiler handles all program checks; control can be passed to a phase to enable it to deal with the check.

<u>Job Termination</u>: The compiler completion code is picked up and control is returned to PLC.

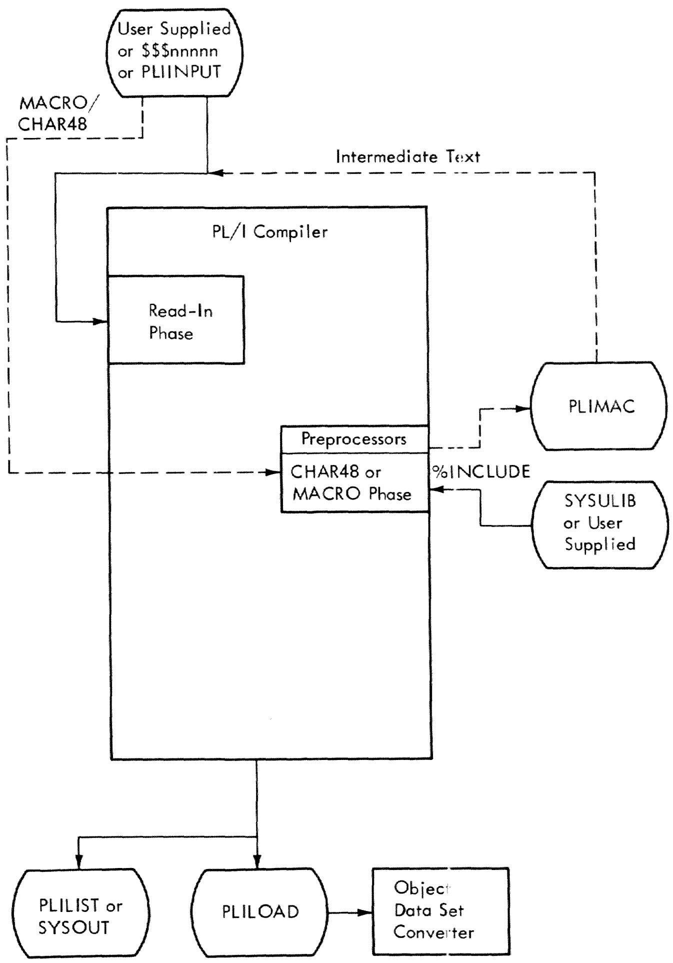


Figure 5. Input and Output Data Sets

The compiler completion codes are:

Code	Meaning
0	No diagnostic messages issued; compilation completed with no errors; successful execution expected.
4	Warning messages only issued; program compiled, successful execution is probable.
8	Error messages issued; compilation completed, but with errors; execution may fail.
12	Severe error messages issued; compilation may be completed but with errors; successful execution improbable. If a severe error occurs during compile-time processing, the compilation will be terminated and, if the SOURCE option has been specified, a listing of the PL/I program text produced by the compile-time processor will be printed.
16	Terminal error messages issued; compilation terminated abnormally; successful execution impossible.

Preprocessing

The PL/I compiler has two preprocessors, the 48-character set preprocessor and the compile-time processor. One of these preprocessors may be used prior to compilation, depending upon user-specified options. However, both of them would never be used for a single compilation.

1. The 48-character set preprocessor is called when input to the compiler is in the 48-character set, requiring translation to 60-character symbols before compilation. The user indicates this by specifying the CHAR48 option.

2. The compile-time processor is called when the source text contains preprocessor statements; this is indicated by specifying the MACRO option. The compile-time processor includes a facility for translating statements written in the 48-character set into the 60-character set. Thus, if both MACRO and CHAR48 are specified, only the compile-time processor will be called.

If neither of these options is specified, both preprocessors are bypassed and compilation is begun, using the PLIINPUT data set as input to the compiler. When either preprocessor is executed, it places the translated source text into the PLIMAC data set, which then serves as source input to the read-in phase. Figure 5 illustrates interaction between the compiler and input/output data sets.

Compilation

The compiler comprises a series of phases that are called and executed in turn under the supervision of the control modules. Each phase performs a single function or set of functions, and is entered only if the services it provides are required for a particular compilation. Control module AM marks the appropriate phases, placing the names in a phase directory in accordance with the content of the source program and the optional compiler facilities selected. Figure 6 illustrates the overall flow of the compiler.

The data that is processed by the compiler is known as text throughout all stages of the translation process. Initially, the text comprises the PL/I source statements submitted by the programmer; at the end of compilation, it comprises the machine

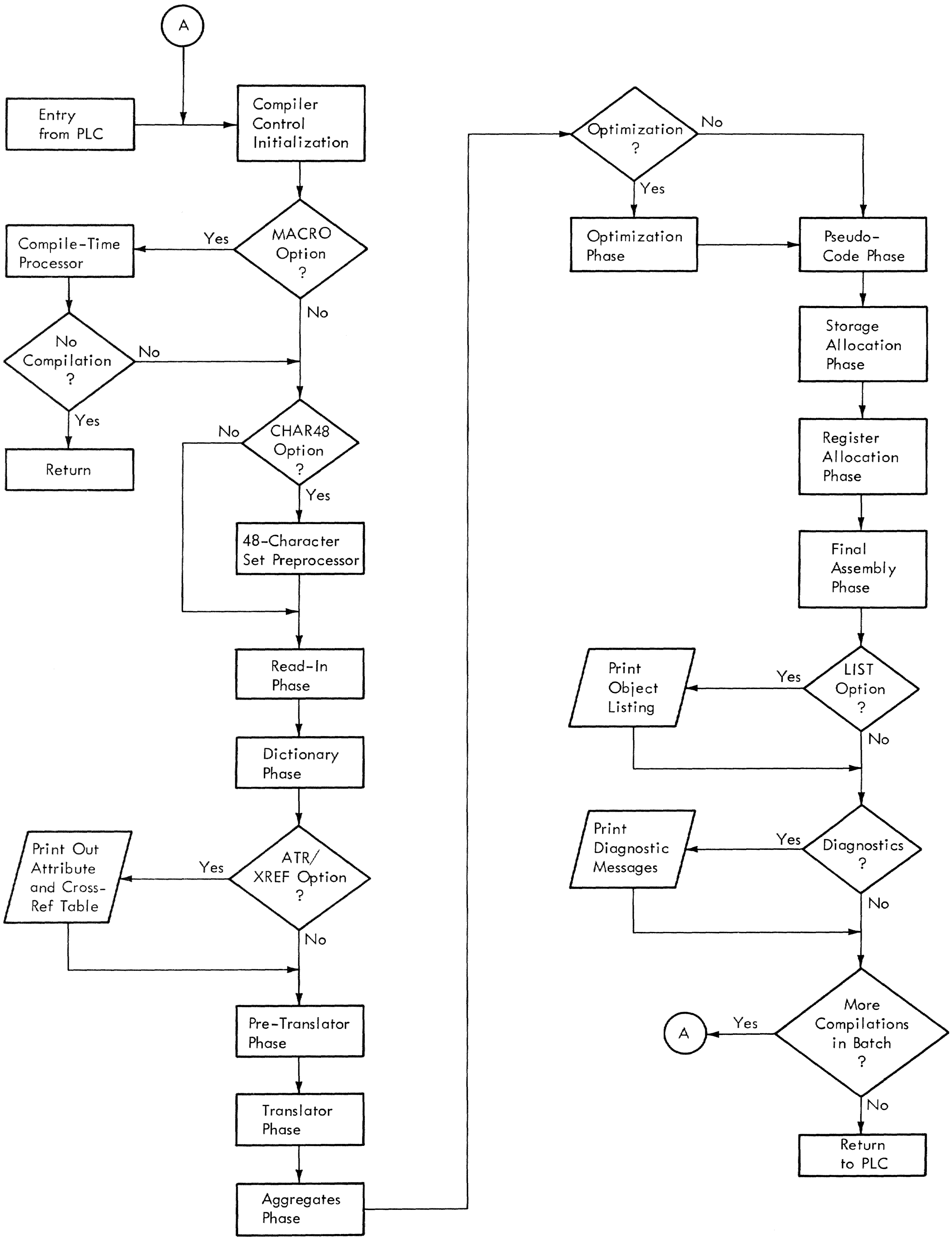


Figure 6. Overall Flow of Compiler

instructions that the compiler has substituted for the source statements, to which is added some reference information for use by ODC.

The read-in phase takes its input either from the PLIINPUT data set or, if preprocessing has preceded it, from the PLIMAC data set. This phase checks the syntax of the source statements and removes any comments and nonsignificant blank characters.

After read-in, the dictionary phase of the compiler creates a dictionary that contains entries for all the identifiers in the source text. The compiler uses the dictionary to communicate descriptions of the elements of the source program from one phase to another. The dictionary phase of the compiler replaces all identifiers and attribute declarations in the source text with references to dictionary entries.

Translation of the source text into machine instructions involves several compiler phases with this sequence of events:

1. Rearrangement of the source text to facilitate translation (for example, by replacing array of structure assignments with DO loops that contain element assignments).

2. Conversion of the text from the PL/I syntactic form to an internal syntactic form.

3. Mapping of arrays and structures to ensure correct boundary alignment.

4. Translation of text into a form similar to machine instructions; this text form is termed pseudo-code.

5. The compiler makes provision for storage allocation for STATIC variables and generates code to allow AUTOMATIC storage to be allocated during execution of the object program. (The PL/I library subroutines handle the allocation of storage during execution of the object program.)

The final-assembly phase translates the pseudo-code into machine instructions, and then creates the external symbol dictionary (ESD) and relocation dictionary (RLD) required by the conversion program. The external symbol dictionary is a list that includes the names of all subroutines that are referred to in the object module but are not part of the module; these names, which are external references, include the names of all the PL/I library subroutines that will be required when the object program is executed. The relocation dictionary contains information that enables virtual storage addresses to be assigned to locations within the object module when it is loaded for execution.

Throughout compilation, subroutines in control modules are referenced to provide whatever services are required by the compiler phases. When compilation is completed, control passes back to PLC, which determines, on the basis of user options, whether ODC or the name processor must be called.

COMPILER INTERFACES WITH THE SYSTEM

PROGRAM LANGUAGE CONTROLLER (PLC) - CFBAA

This routine is the interface for accomplishing any or all of these functions:

1. Compile a prestored data set

2. Create a line data set and compile it

3. Process the compiled data set to make it executable in TSS/360

4. Process references to external subroutines so that the subroutines can be called explicitly rather than duplicitly.

5. Print the compiler-created listing data set.

Entry Points:

CFBAA contains five entry points -
CFBAA1 - Entry from Command System Analyzer
GRI points to first word in BPKD list
PARAM dsect
CFBAA2 - End entry point from text editor
No parameter
CFBAA3 - Entry to MERGELST block build routine
GRI points to module name padded to right with blanks
CFBAA4 - Entry to compiler data management routine
GRI contains pointer to request code byte
PARAM dsect
CFBAA5 - Entry to language processor early end routine
No parameters

Input: PL/I command parameters pointed to by CFBAA8 BPKD
CHBTDT - Task data definition table
CHBTCM - Task common

Output: CHBPLI - PL/I communication area
CHBMGL - merge list of module names
Messages: All CFBAA messages listed in System Messages.

Routines Called: External

CZABD (PRINT)	CZASW4 (LPCEDIT)
CZAEA3 (DDEF)	CZATJ1 (PRMPT)
CZAEC1 (FINDDS)	CFBAB1 (ODC)
CZAE12 (CATALOG)	CFBAK (NAME PROCESSOR)
CZAEJ7 (ERASE)	CZCLA (COMMON OPEN)
CZAFJ2 (RELEASE)	CZCLB (COMMON CLOSE)
CZASC7 (SYSIN)	CZCOJ (FIND)
CZASDX (GDV)	CZCOK (STOW)
CZASW1 (LPCINIT)	IEMTAA (PL/I COMPILER)

Exits: All exits are made by means of the RETURN macro. There are no SYSERR or ABEND exits.

OPERATION: CFBAA is divided into four sub-modules:

1. Mainline processing (CFBAA1)
2. MERGELST block build routine (CFBAA3)
3. Compiler data management routine (CFBAA4)
4. Language processor early end routine (CFBAA5)

Mainline Processing: In order to be reentrant if interrupted during a previous compilation, PLC checks its footprint flag found in the communication bucket (CHAPLI) for the last operation completed before the interruption took place. From the value, it determines what end processing must take place before a new compilation may be begun.

After cleanup, the communication area is initialized to zeros and default values. BPKD pointers are then inserted in their designated slots in the communication area. The merge list is built, including each name specified in the MERGELST parameter, using the subroutine CFBAA3. PLC options are scanned and appropriate values are filled into the bucket.

If the user request explicit-call processing only (PLCOPT=NOCONV) and gave valid EXPLICIT and XFERDS operands, PLC skips compilation and conversion and calls the name processor (CFBAK) to change implicit calls to explicit calls.

If NONCONV was specified, and if a module name was given as input, the module name is validated, with prompting for a new name if it is invalid. In a non-conversational task the compilation is bypassed if the module name is invalid. If a module name was not input, SOURCEDS (if valid) may be used for the name. If neither NAME nor SOURCEDS were input, PLC will skip compilation and call ODC (CFBAB) to convert input from MERGELST or MERGEDS parameters.

When a source data set name was given as input, FINDDS is called to validate the name and locate a JFCB for it. If there is no JFCB, FINDDS is called again to create one. If it is unable to DDEF the data set, the CKNAM subroutine is used to determine data set attributes from the name. DDEF is called with the appropriate DSORG; then the data set is opened for the update option. The text editor is then invoked to create the data set.

If the data set exists, a check is made to see if it is shared with read-only access. For that case the data set is opened for input; all other data sets are opened for update. After the SOURCEDS is fully open, the data set is checked in the DCB to ensure that it is a VISAM line data set (or member). If it is not, compilation is bypassed.

When the source data set has been validated or created, the compiler is called with the address of a two-word parameter list in register 1. The first word contains the address of the PLIOPT string, whose length is in the byte preceding the string, and the second word contains the address of the communication bucket (CHAPLI).

When the count of modules in the MERGELST is greater than one, or a single module was compiled without terminal errors, or there is a pointer to a MERGEDS parameter, ODC is called with no parameters needed. Following conversion, if there has been a compilation with a separate listing data set, and a print request was an input parameter, the PRINT macro is issued with appropriate parameters.

The source data set is closed (stowed if necessary) and the JFCB released if PLC defined the data set. The continuation bit is checked for further compilation and, if off, a normal return is made to the user.

When the continuation bit is on, a SYSIN macro is issued to obtain the next input, with prompting "PLI:". The parameter is moved to PLI BPKD, all unwanted BPKD parameters from the previous compilation are zeroed out, and processing continues at initialization.

MERGELST Block Build Routine: The address of the name to be added to MERGELST is contained in register 1 on entry. The routine checks for the last block available by following forward chain pointers, then checks to see if this block is full (MGLCNT=15). If it is full, a GETMAIN is issued for another block of 128 bytes, and a pointer to this block is inserted in MGLPTR in the last block obtained. The name pointed at by register 1 is inserted in the first available slot and the MGLCNT is updated by a count of 1. A normal return is made to the calling routine with no return codes.

Compiler Data Management Routine: Register 1 contains a pointer to a code specifying what type of data set and what data management function are required.

Code values for functions are:

	DDEF	ERASE	RELEASE
LIST DS	14	18	1C
LOAD DS	24	28	2C
MACRO DS	44	48	4C

For a DDEF request, a CATALOG macro establishing a GDG index is issued for all cases except that where the user has specified his own macro data set name. Then the appropriate DDEF is issued for the requested data set. An immediate return is made to the calling routine.

For an ERASE request, either the system name or the user supplied macro data set name is used with the ERASE macro, followed by a return to the calling routine.

A RELEASE request will use the system supplied ddname for the data set in question followed by a return to the calling routine.

Language Processor Early End Routine: This is a stand-alone routine invoked only by the user control routine (LPCINIT) function under certain conditions:

1. PLC has been interrupted while creating a new source data set.

2. A new language processing request has been made for text-editor services.

The routine enables PLC to close out the data set being created, refresh the source DCB, and reset the footprint to zero. In order to prevent the routine from taking effect when PLC is the language processor, and thus reinvoking the text editor, PLC sets a switch so that all processing is bypassed. PLC does its cleanup at initialization time.

OBJECT DATA SET CONVERTER (ODC) - CFBAB

ODC converts compiler-formatted object modules into TSS-formatted object modules and resolves the library-known pseudo registers (PRVs), other pseudo registers are passed on to the dynamic loader.

Entry Points:
- CFBAB1 - Entry from PLC to mainline processing
- CFBAB2 - Entry from PLC to task cleanup

Input: The ODC routine will be passed the following input data by PLC and the compiler modules:

Via the communications bucket (CHBPLI) -
Merge list pointer, if any.
Pointer to the merge data set name, if any.

Via the task library chain -
The job library into which the output is to be stored and from which the data set names are determined, if the reprocess option is selected, as indicated by the fact that the merge data set name equals the job library.

Via the merge list (CHBMGL) -
A list of modules to be converted.

Via the merge data set -
Additional names of data sets to be processed.

This input can take three forms:

- The name of the most recently defined job library - reprocessing indicated.
- The name of a VPAM data set which is not the current job library. A copy of every member in this library is to be processed and placed in the current job library.
- The name of an independent data set made up of records, each containing up to 15 names of modules to be processed and stored in the current job library.

Output: This routine produces TSS/360-formatted object modules, which it stores in the appropriate job library. It optionally produces a data set that contains the offsets into the PRV.
Messages: All CFBAB messages are listed in System Messages.

Routines Called: External

CZCLA (COMMON OPEN)	CZAFJ (RELEASE)
CZCGA2 (GETMAIN)	CZCOJ (FIND)
CZAEC (FINDDS)	CZCGA3 (FREEMAIN)
CZATJ (PRMPT)	CZCLB (COMMON CLOSE)
CZCOK (STOW)	CZAEA (DDEF)

Exits: All exits are made by means of the RETURN macro. There are no SYSERR or ABEND exits.

OPERATION: CFBAB is divided into two submodules:

1. Mainline processing (CFBAB1)

2. Task cleanup (CFBAB2)

Mainline Processing: After standard initialization, ODC searches out the most recently defined job library and opens it for update. It then obtains virtual memory space in which to process changes for the pseudo register vector table. If the name of the merge data set and of the job library are the same, a merge list member is added for each member of the library.

When the merge data set name is the name of a VPAM data set which is not the job library, an entry is added to the merge list for each member of that data set. Subsequent processing proceeds as for a stand-alone merge list.

If the data set provided as the merge data set does not have VSAM, VISAM, or VPAM organization, a warning message is issued and the merge data set ignored. Processing otherwise proceeds as normal.

Data set names are processed from the merge list in the following order: first, the name has appended the prefix 'LOAD', and an attempt is made to find the input data set. If not extant, a message is issued and the next module is processed. If extant, a JFCB is created, if not previously defined.

A DCB is built for the input data set and the data set is opened. Storage is obtained in which to build a PMD and the text for the TSS/360-formatted module.

Records in card-image form from the input data set are processed according to type until either an END card or version ID card is encountered, or the data set is exhausted. If the version ID card shows that terminal errors were detected during compilation, conversion is terminated and the proper message is written. If no END card is found, a message is issued and conversion continues.

A default value, PLIPACK, is checked to determine whether the user wants his CSECTs packed on external storage. If PLIPACK=Y, all CSECTs are packed. If PLIPACK=P, non-common static external CSECTs smaller than 4096 bytes, text CSECTs, and static internal CSECTs are packed. If PLIPACK is any other value or no value, no CSECTs are packed. All packed CSECTs within a module are combined into a singel CSECT. Packed CSECT names are transformed into entry point names.

The compiler generates a single record following the end record. This record contains a time-date stamp for version ID and the maximum error level detected during compilation. These values are inserted in the PMD for use by the program control system and the dynamic loader.

The PMD header is then created. If any ESDID numbers are missing, a warning message is issued and processing resumed. A blank CSECT name likewise produces an error message, and the CSECT will be skipped.

A control section dictionary (CSD) is created for each valid CSECT. RLD entries are built for all external and internal references in the CSD. The CSD and PMD are then completed in preparation for stowing the module generated into the job library.

ODC then determines whether this module replaces another with the same name. If it does, the old version is deleted from the job library. A DELETE macro is issued to unload any old copy in virtual memory.

The new module is then stored in the job library. The working storage is released and the input data set closed. If the JFCB for the input data set was created by the processing of this routine, that JFCB is released. An appropriate message is issued to inform the user into which job library the module was placed, and whether it was a replacement. If the module is too large to convert within the virtual memory work space allocated for this purpose, an error message issued and the module is skipped.

The remaining modules, if any, are processed until the merge list has been exhausted. The pseudo register vector data set (if specified) is written after all modules have been processed.

Errors may be detected while storing away the newly processed module. They are handled as follows:

- If an error is detected while trying to determine the existence of a prior alias version in the job library, an error message is issued and the module is skipped.
- If an error is detected while trying to stow the new module, an appropriate error message is issued.
- If the error detected was that of duplicate entry point names, the user is offered the opportunity to have the duplicate names listed and to terminate or continue processing after skipping the present module.

Task Cleanup: This submodule closes data sets that ODC has left open and frees all working storage that ODC aquired.

NAME PROCESSOR - CFBAK

The name processor helps the user transform implicit calls to explicit calls

Entry Points:

CFBAK1 - Entry from PLC to mainline processing

CFBAK2 - Entry from PLC to task cleanup

Input: The name processor receives the following input:

Via the communications bucket (CHBPLI) -

Pointer to merge list, if any.

Pointer to EXPLICIT operand, if any.

Pointer to XFERDS operand, if any.

Via the task job library chain -

The last job library in the chain, in which ODC stowed the object module that contains the names to be processed.

Via the merge list (CHBMGL) -

A list of converted modules to be checked for name transformation.

Output: If the EXPLICIT or XFERDS operand was used, the name processor adds pad characters to the beginnings of selected external references. In addition, this routine optionally adds lines of the form

0-7	line number
8	X'00'
9-16	new name
17	blank
18-24	PLICALL
25-27	blank
28-35	old name

to a line data set named in the XFERDS operand; if the named data set does not exist, this routine creates it.

Messages: All CFBAK messages are listed in System Messages.

Routines Called: External

CZAEA4	(DDEF)	CZCLAO	(COMMON OPEN)
CZAEC1	(FINDDS)	CZCLBC	(COMMON CLOSE)
CZAFJ3	(RELEASE)	CZCOJ1	(FIND)
CZASDX	(GDV)	CZCOK1	(STOW)
CZATJ1	(PRMPT)	CZCOR1	(VS GET)
CZCGA2	(GETMAIN)	CZCOU1	(VS PUTX)
CZCGA3	(FREEMAIN)	CZCPA1	(VI PUT)
		CZCPB1	(VI GET)

Exits: All exits are made by a branch, on register 15, to PLC. There are no SYSERR or ABEND exits.

OPERATION: CFBAK is divided into two submodules:

1. Mainline processing (CFBAK1)
2. Task cleanup (CFBAK2)

Mainline Processing: After standard initialization, CFBAK obtains and validates default values for PADCHAR and UPDTXFER. If system default values are used, PADCHAR=@ and UPDTXFER=N.

CFBAK constructs three symbol tables to facilitate the search for symbols specified in the EXPLICIT and/or XFERDS operand. Each table entry contains sixteen bytes; the new name is in the first eight bytes, and the old name is in the second eight bytes.

Symbol table 3 is constructed first, from entries in the EXPLICIT operand. If EXPLICIT=*ALL, no table is built; a switch is merely set. If EXPLICIT=(MODA,MODB), MODA and MODB are entered. If EXPLICIT=*ALL(MODA,MODB), MODA and MODB are entered but a flag is set to indicate omissions.

Symbol table 1 is built next, from records in the transfer data set, if the transfer data set has been supplied. CFBAK dissects the records into label (new name) and operand (old name), ignoring records if they do not fit into the standard pattern, and inserts the names into the table.

The last job library is opened, and a FIND is issued for the first name in the merge list. If it is found, a GET is issued against the module to pick up the PMD. For each CSECT, the REF chain is checked REF by REF through tables 3 and 1 for a match. A match in table 3 plus appropriate flags tell whether the REF is to be changed or ignored. If it is to be changed, a check is made in table 1 to see if the name was changed in the transfer data set, and if it was, the label in the transfer data set is substituted for the REF in the PMD. If it is not in table 1 and is to be added to the transfer data set, the name is checked for valid characters, prefixed by the pad character, checked against both tables for possible conflict, and added to symbol table 2.

When all CSECTs and REFs have been checked, the contents of symbol table 2 are formed into PLICALL records for the transfer data set. Then the processed PMD is placed in the module by means of a PUTX macro instruction; the module is stored into the job library. The next module in the merge list is treated in the same way until no more remain. CFBAK then reports all names from the EXPLICIT operand that were not found.

Task Cleanup: This submodule frees working storage and closes data sets that were left open.

COMPILER CONTROL

The compiler control modules perform specific functions for the compiler; these modules and the subroutines they contain are referenced constantly throughout compilation. Two of the control modules, modules AA and AL, contain the service subroutines, and are responsible for performing most of the services required by the compiler. Tables of these subroutines and their functions are in Section 3.

When compilation is called for, PLC calls module AA, AA links to AB, and AB performs the initialization of the compiler. The addresses of the service routines contained in AL are placed by AB in a table in AA. From that point, modules AA and AL are referenced constantly throughout the compilation process.

Module AA - First-Half Service Routines

Module AA is the base module for the compiler. The transfer-vector table, containing the addresses of the entry points of service subroutines in both AA and AL, resides in AA. The transfer vector table consists of a series of ADCONS. The ADCONS for service routines in AA are resolved when AA is loaded. The addresses of service routines in AL are inserted into dummy ADCONS by AB. The offset of each ADCON in the table is fixed and is known by all compiler phases. If a compiler phase wants to call a compiler service routine, its link register is loaded with the ADCON from this offset and the branch executed. A second table in AA points to frequently referenced information in storage.

AA is responsible for phase linking. Facilities are provided for marking phases (as specified by the phase-marking module, AM), calling physical phases and then returning control to the caller, and passing control to a new phase.

Translate tables for converting external codes (EBCDIC, BCD) to internal code, and the reverse, are contained in AA. The specific table supplied for an operation will depend upon the option specified by the user. AA also contains the DCB for the load file.

Module AL - Second-Half Service Routines

Module AL contains a series of ADCONS for the service subroutines located in it. These ADCONS are resolved at load time and, by means of the initialization process performed by AB, inserted in the transfer-vector table in AA. There are a few infrequently used service routines in AL, whose addresses are maintained only in AL and are not transferred to AA. The remainder of module AL consists of service subroutines. These subroutines are described in Appendix H.

Module AB - Initialization

AB, the initialization routine of the compiler control phase, performs these functions:

- Opens the LOAD file (PLILOAD) if necessary,
- Constructs the phase directory (for details see "Resident Tables" in Section 4),
- Obtains space for text blocks and dictionary blocks,
- Sets up a communications region in the first dictionary block,
- Scans the user-supplied options list and picks up default values from the options table in module AF when necessary,
- Tests for CHAR48 and/or MACRO and then opens the macro data set (PLIMAC) and calls module AC, if necessary,
- Prints a list of options used in the current compilation,
- Tests for the BCD/EBCDIC option and moves the correct translate table from AA into the dictionary,
- Inserts error messages, which may have been generated when the LOAD file was opened, into the dictionary,
- Places the addresses of the compiler service routines in AL into the transfer-vector table in AA,
- Causes the first card to be read and stores it for use as a heading for the listing.

On completion, AB returns to AA with a completion code. If this code is satisfactory, the first logical phase (read-in) is invoked. If the code is unsatisfactory, the compilation is terminated.

Module AC - Intermediate File Control

This module controls writing operations of text, complete with VISAM line numbers, on PLIMAC, the intermediate text file. It is entered only if the CHAR48 or MACRO option is specified.

AC is also responsible for entering module AG at the end of the compile-time phase to close PLIMAC for output and open it for input. In other words, where MACRO and/or CHAR48 are specified by the user, PLIMAC rather than PLIINPUT acts as source input to the compiler.

Module AD - Interphase Dumping

Module AD is responsible for performing interphase dumping. All specified active storage is dumped at the end of the phases stated or implied in the DUMP option. If the DUMP option includes either I, for the annotated dictionary dump, or E, for the annotated text dump, or both, then module AD will load either module AH or modules AI and AJ, or all three, to produce the required output.

The DUMP option, which indicates where main storage is to be dumped, may be specified in one of these ways:

1. DUMP, means a dynamic dump is required (the dump routine will be called by a running phase),
2. DUMP=(area,x_1,x_2,x_3,....x_n) means a dump of the storage after the named phase, where x is the name of a phase.

Area is any combination of TDSCIE:

T text blocks
D dictionary blocks
S scratch storage
C control phase
I annotated dictionary blocks
E annotated text blocks

The general syntax is:

DUMP[=([area],{x|(y,z)},....)] where x, y, and z are phase numbers.

A single phase name indicates dumping of storage after this single phase. A pair of phase names indicates a continuous group of phases, after each of which dumping of storage is to occur. The dump will appear on PLILIST or SYSOUT, depending upon user option, inserted into the normal compiler output.

If area is omitted, the default taken is DTS. If a program check occurs, and DUMP has been specified, then area will be given the default DTSC.

Note: The operations of module AD are very closely linked to those of module AT (TRACE Option) in the performance of interphase dumping; module AT is, therefore, documented immediately following.

Module AT - TRACE Option

Module AT provides the debugging facility known as TRACE, which makes it possible to obtain a printed list of all instructions executed (TRACE) or of all branches taken (FLOW) during execution of a specified segment of a compilation. Use of the TRACE facility requires the inclusion of the following input:

- "DDEF TRACEOUT, VS, dsname", which defines the PRINT file that will carry the TRACE output. It should be printed after compilation with the EDIT option off.
- The option "T" in the PLIOPT parameter of the PLI command.
- *TRACE or *FLOW records immediately before the first PL/I source record. A maximum of 10 *TRACE and/or *FLOW records are permitted.

The format for a *TRACE or *FLOW record is as follows:

- * in column 1
- The keyword TRACE or FLOW
- The two-character name of the PL/I module in which the trace is to start
- A four-digit offset (with leading zeros, if necessary) within the module in which the trace is to start
- The two-character name of the PL/I module in which the trace is to end
- A four-digit offset (with leading zeros, if necessary) within the module in which the trace is to end
- A five-digit statement number (with leading zeros, if necessary) designating the statement for which the option is to be applied. If no statement number is specified, the trace will occur for every executable statement in the program.

Blanks between the * and the keyword are optional. One or more blanks are required between other fields.

An example of a valid *TRACE record is:

```
*TRACE  CI  002E  CO 0f3c  00024
```

Modules AI and AJ - Text Dump

Modules AI and AJ are called, if E is specified in the area field of the dump option, to provide an 'easy-to-read' text printing, in which the triples and pseudo-code items are printed separately. This option is available between phases IA and OE inclusive.

Module AK - Compiler Closing

Module AK, the closing routine of the compiler, releases main storage and scratch storage used for dictionary and text blocks and unloads all output modules except the control output module.

The only data set AK is responsible for closing is PLILOAD (PLIMAC, if used, was closed by AE; PLIINPUT and PLILIST will be closed by PLC). AK closes PLILOAD after each compilation, whether or not batch compilation was specified. A new load data set is opened by AB for each compilation in a batch. Figure 7 shows what action is taken on each of the data sets by the various modules.

For each load data set produced, the error level and date/time must be preserved together with the data set. This

MODULES	DATA SETS				
	PLIINPUT	PLILIST	PLILOAD	PLIMAC	INCLUDE LIBRARY*
PLC	OPEN/ CLOSE	OPEN/ CLOSE			
AA			WRITE		
AB		WRITE	OPEN	OPEN	
AC				WRITE	
AE				CLOSE	
AG				CLOSE/ OPEN	
AK			CLOSE		CLOSE
AL	READ	WRITE			
AS					READ
BG					OPEN

* Either SYSULIB or user supplied name.

NOTE: The module name refers to the module containing the I/O subroutine and does not indicate the module requesting the I/O operation.

Figure 7. Input/Output Usage Table

information is obtained by AK from an 80-byte record added to the load data set file immediately prior to closing the file; then an entry is made to control subroutine ZULF.

If a batch compilation is specified, a check is made to determine whether any source programs are still to be compiled. When one or more programs remain to be compiled, the batch delimiter card is scanned for syntax errors, and control is returned to module AA.

Module AM - Phase Marking

Module AM marks phases as either wanted or not wanted, depending upon the compiler invocation options. Phases that are always called are marked wanted. AM is entered after completion of AB. It tests the relevant bits in the Control Code Word (CCCODE), loads the required output modules, and updates the phase directory. It then marks modules as wanted or not wanted in the phase directory.

Module XZ - Conversational Diagnostic Messages

This module is responsible for building conversational diagnostic messages. In addition to the conventional method of printing diagnostic messages with the listing, the user has the option of having them printed out at the terminal as errors are detected. XZ is called by the ZUERR subroutine in module AL whenever this option is specified.

On entry, XZ prepares a buffer area for constructing the message text. The severity code is examined and inserted in the buffer area. The statement number is used to examine the statement-line table to obtain the corresponding line number. Both of these are then inserted in the buffer area.

The BREVITY option is examined to determine if the message text must be located and a full message constructed in the output buffer. The buffer is then directed to SYSOUT by GATWR macro and XZ returns control to ZUERR.

PREPROCESSING PHASES

48-CHARACTER SET PREPROCESSOR

Phase BX is the 48-character set preprocessor. It is called on programmer option and receives, as input, source text in the 48-character syntax.

The preprocessor scans the input text for occurrences of characters peculiar to the 48-character set, and converts these to the corresponding 60-character symbols. It then puts out the adjusted text onto auxiliary storage ready for Phase CI, the first pass of the Read-In Phase.

The text is read in record by record. It is then scanned for alphabetic characters which may be the initial letters of operator keywords, for periods, and for commas. Items within comments or character strings are ignored.

When a possible initial letter is discovered, tests are made to determine whether or not one of the reserved operator keywords has been found. If one has been found, it is replaced by its 60-character set equivalent. Similarly, appearances of two periods are replaced by a colon, and a comma-period pair is replaced by a semicolon if the comma-period pair is not immediately followed by a numeric character.

Allowance is made for the possibility that a concatenation of characters which is meaningful in the 48-character set may be split between two records.

The output from the preprocessor is the transformed 60-character set text only; the 48-character set text is not preserved. The read-in phase processes the transformed text, and only the 60-character set text is printed.

The 48-character set preprocessor uses Compiler Control routine ZURD to obtain input, and routine ZUBW to place its output onto auxiliary storage.

Note: If the MACRO option is specified, all the processing described above is done by the compile-time processor, and phase BX is bypassed.

COMPILE-TIME PROCESSOR LOGICAL PHASE

The compile-time processor consists of six physical phases. Each of these phases is executed once, unless an INCLUDE data set is encountered. In this case certain phases will be re-executed.

The compile-time processor moves source text that does not contain compile-time statements directly into text blocks. During this process invalid characters are replaced by blanks, and line numbers are encoded and inserted into the text. Compile-time statements are decoded and translated into an internal form and then placed directly into text blocks. An entry is made into the dictionary for each compile-time variable, procedure, label, or INCLUDE identifier.

A second pass is then taken over these text blocks, during which compile-time statements are executed and the PL/I source program text is scanned and replacements are made. The output from this pass is a PL/I source program contained on PLIMAC.

If during the second pass, an INCLUDE data set is processed, the entire procedure indicated above is executed recursively to process this text.

Text and dictionary formats used by the compile-time processor are contained in Appendix F.

Line Numbering

As the input is being processed a unique line number is assigned to every logical record processed. If a listing of the input is requested, these line numbers are written out beside the appropriate line. The line numbers are also encoded and inserted into the text so that diagnostics can be keyed to them. These line numbers are also output on PLIMAC, to aid the user in determining from which input line a particular line of output came.

Phase AS

This phase, consisting of one physical module, is loaded if the MACRO option is specified. It is resident throughout compile-time processing until the cleanup phase (BW) is invoked.

This phase controls the loading of the subsequent compile-time processor phases. The initialization phase (AV) is loaded only once. The two processing phases (BC and BG) are loaded and executed once unless an INCLUDE data set is processed. In this case phase AS reloads the processing phases to process this data set.

In addition, phase AS contains a set of service routines used by both processing phases. Access to these routines is via a transfer vector located at the beginning of phase AS.

Phase AV

This phase consists of one physical module. Its purpose is to initialize certain cells in the communications region for the compile-time processor phases.

Phase BC (BE, BF)

Phase BC consists of three physical modules, BC, BE, and BF. Module BE contains the control routine.

Phase BC accepts input text, moving it into text blocks until a compile-time statement is found. (For a description of the use and layout of text and dictionary blocks, see Appendix F.) When a compile-time statement is encountered, it is encoded into a set of interpretive instructions and, except for compile-time procedures, added to the current text block. Compile-time procedures are similarly encoded, but are placed in separate text blocks.

As compile-time statements are encoded, all non-keyword identifiers encountered are entered into the dictionary, together with any attributes that are known. Entries are also made in the dictionary for constants and iterative DO-loops.

During phase BC, invalid characters occurring outside of strings and comments cause a diagnostic to be printed. They are converted to blanks. Invalid characters can thus be used for markers of various sorts in text blocks. Diagnostics are given for syntax errors in compile-time statements. Line numbers are encoded and inserted into the text for the use of the phase BG scan. All input characters are converted to their EBCDIC representation before they are processed.

Phase BG (BI, BJ)

Phase BG consists of three physical modules: BG, BI, and BJ, which contain the control routine, the macro-code interpreter, and the built-in function handler, respectively.

In general, the input to phase BG is the set of chained text blocks and dictionary blocks created by phase BC. The phase BG execution is essentially that of the compile-time processor described in the external specifications. That is, its basic action is to move through text blocks looking for instances of compile-time variables or compile-time statements, which it uses to produce the output text. As line numbers are encountered in the text, they are placed into a location containing the current line number. This is used both for phase BG diagnostics and by the output editor.

If a compile-time variable or procedure reference is found, the scan cursor is positioned to scan its value. When the scan of the value is completed, the cursor is properly positioned back into the text. If a compile-time variable or procedure reference is found in this value scan, the process repeats itself. Such nesting can occur to a depth of 100.

If the scan encounters an encoded compile-time statement (built by phase BC), control is passed to an interpreter. This interpreter executes the statement -- possibly repositioning the scan cursor -- and returns to the scan.

The output of this phase is a PL/I source program contained on PLIMAC.

Phase BM (BO)

Phase BM examines the heads of the error chains in the first dictionary block, and programmer options which specify the severity level of messages required. If there are no messages, it passes control to the clean-up phase (BW). If diagnostic messages are required, the phase loads BN to process them after scanning the chains and indicating where the text is to be found, from the message directory block, module BO.

Module BN (BP, BV)

The text of all compile-time processor error messages is kept in modules BP through BV. The messages are ordered by severity, within these modules. BM will have listed those modules which contain messages required for a particular pass. Module BN loads and releases these modules, one at a time and extracts the required messages. When all compile-time error messages have been processed, module BN returns control to BM.

Phase BW

The purpose of this phase to set all tables and communication region cells to the values required by the compiler proper. In addition it will release all text and dictionary blocks used by the compile-time processor phases and then pass control to the next required phase of the compiler.

If a severe or terminal diagnostic has been produced by the Compile-time processor a listing of the contents of PLIMAC will be printed (provided that the SOURCE option applies), and compilation will be bypassed.

COMPILER LOGICAL PHASES

READ-IN LOGICAL PHASE

The read-in logical phase consists of five discrete physical phases, each of which processes a particular group of statement types. The phase obtains the input text in the externally coded form by a call to the compiler read routine, and converts it to internal code by means of a translation table provided by compiler control.

The source text is scanned for syntactical errors. During this time an output string is built up, which consists essentially of the input text with comments and insignificant blanks removed. The source text is scanned and statements are numbered, identified, and diagnosed. Any required substitutions are made, statement labels are inserted in the dictionary, and chains are formed (for example, BEGIN, PROCEDURE chains). If the SOURCE option applies, source statements, with their line number, and optionally, their block levels and DO-nest levels, are printed out immediately after they have been read.

When the input text provides an end-of-file indication, processing is terminated. In ERROR situations this may not occur when a valid external procedure has been completely processed. By keeping a count of PROCEDURE, BEGIN, DO, END, ON, and IF statements, the phase can detect when the logical end-of-program indication is found. If there are more records after the end of the external procedure, they are ignored.

If an end-of-file indication is encountered before the logical end of the program, diagnostic messages are issued and suitable END statements are inserted to allow compilation to continue.

The output of the Read-In Phase provides a syntactically correct output string; a table of entry and statement labels; chains of coded diagnostic messages; a set of switches specifying compilation content details; a set of chains linking statements of a particular type, to facilitate subsequent scanning; and optionally, a listing of the source text.

Statement Numbering

All statements are given a sequential number. A table is then built that associates each statement number with the VISAM line number of the statement. This includes each compound statement, each statement contained in a compound statement, block and group delimiting statements, and null statements. The statement and line numbers are indicated on source listing and diagnostic message printouts.

Statement and Entry Labels

Statement and entry labels appearing in the source text are removed and added to a label table, which is built up in the region intended for the dictionary. This region may be extended by further blocks as required. The label table entry is an embryo dictionary entry, with blank regions to be filled later by the Dictionary Phase EG.

When a label declaration is found, an entry is made in the label table with a statement label code, the current (updated) sequential number, and the current block level and block count.

Statements having multiple labels give rise to multiple label table entries. These entries are identical except for the BCD name.

If the statement following a label is subsequently identified as a PROCEDURE or ENTRY statement, the label table is reaccessed, and the entries associated with the statement are modified (see "Dictionary Entries for Entry Points" in Section 4).

Chains Constructed by Read-In

To provide rapid scanning in the dictionary phases, the following chains are constructed by the Read-In Phase:

The CALL chain

The PROCEDURE-ENTRY-BEGIN chain

The DECLARE chain

The ALLOCATE chain

Errors and Diagnostic Messages

As the source text is scanned it is syntactically analyzed. Keywords are identified and passed as valid only if they may legally appear within the type of statement being diagnosed. However, consistency of attributes and options within a statement are not normally analyzed. This is left for Phase EK.

When a syntactical error is detected, an attempt is made to correct it and an appropriate diagnostic message is generated. The main aim of the Read-In Phase is to present syntactically correct text to subsequent compiler phases. Certain corrections are performed without prejudicing the complete compilation.

Detected errors cause a diagnostic message to be added to a diagnostic message chain in the dictionary area. Each message is in a coded form with parameters (textual matter, statement and line numbers, and so on). The message is decoded and printed out by the error editor.

Where an error makes it impossible for the scan of a statement to continue, the statement is terminated correctly at such a point as to leave the statement syntactically correct. The text between that point and the next semi-colon (not in a comment or character string) is skipped. The diagnostic messages produced in these circumstances will include at most the first ten characters of the text that is skipped.

The Output String

The output string is so arranged that a complete statement never spans storage blocks. One of the conditions of a successful compilation is that the output resulting from any statement must not exceed the block. This restriction, however, does not apply to DECLARE statements. Formats of the statements appearing in the output string are given in Section 4 under "Text Formats After the Read-In Phase."

All constants and operators, and all identifiers which are not recognized as keywords in the source text, appear in the output string.

Initial Labels

Subscripted label variables which are initialized by attachment to statements are placed in pseudo-assignment statements in text, and then handled as if they were normal labels.

STRUCTURE OF THE READ-IN LOGICAL PHASE

The read-in phase can occupy 16K bytes of storage for any one pass. A storage map for this phase is shown in Figure 8.

```
Bytes
  0    ┌───────────────────────────────────┐
       │                CA                 │
 4K    ├───────────────────────────────────┤
       │                CC                 │
 7K    ├──────┬──────┬──────┬──────┬───────┤
       │  CE  │  CK  │  CN  │  CR  │  CR   │
 8K    ├──────┼──────┼──────┼──────┼───────┤
       │  CG  │  CL  │  CO  │  CS  │  CV   │
12K    ├──────┼──────┼──────┼──────┼───────┤
       │  CI  │  CM  │  CP  │  CT  │  CW   │
16K    └──────┴──────┴──────┴──────┴───────┘
        PASS 1 PASS 2 PASS 3 PASS 4 PASS 5
```

Figure 8. Storage Map for the Read-In Phase

The read-in phase consists of five phases or passes, each containing at most five modules. Modules CA and CC consist of common routines which are invoked throughout the phase by each of the passes, in turn. Modules CE, CK, CN, and CR contain separate keyword tables. Details of the organization of these tables are given in Section 4 under "Resident Tables." Control for each pass resides in modules CI, CL, CO, CS, and CV respectively. The following description refers to the phases by these names.

Phase CI

During phase CI (the first physical phase of the Read-In Phase) the source text is read into storage, and character codes are converted to an internal form. Statement types are identified, labels are inserted into the dictionary, and statement identifiers are replaced by single-byte codes (see "Text Code Byte after Read-In Phase" in Section 4).

A record is kept of block nesting levels and counts to enable a check to be made for the logical end-of-program indication. In order to do this, certain statements have to be either partially or completely analyzed in this pass.

These statements are:

PROCEDURE-END
BEGIN-END
DO-END
IF-THEN-ELSE
ON

CI calls a subroutine in AL to issue a GETMAIN for 16K bytes of storage in which a statement-line number table is created. Each statement number is associated with its corresponding VISAM line number.

If the SOURCE option has been requested, a listing of the source program, with the statement and line numbers, is printed out onto the specified output medium.

Phase CL

The output from phase CI is processed and the statement types listed below are analyzed in greater detail:

ENTRY	FREE
PROCEDURE	WAIT
DO	READ
Iterative DO	WRITE
RETURN	DELETE
GO TO	UNLOCK
DELAY	LOCATE
DISPLAY	REWRITE

If any errors are detected during this pass, diagnostic messages are inserted into chains in the dictionary as required.

Phase CO

The output from phase CL is processed. In particular, the DECLARE, ALLOCATE, and CALL statements are analyzed in greater detail. The syntax of attributes is checked, but their consistency is analyzed during phase EK. If the source program does not contain any of these three statements, this pass is not invoked.

If any errors are detected during this pass, diagnostic messages are inserted into chains in the dictionary.

Phase CS

The output from phase CL or CO is processed. In particular, the syntax of input/output statements is analyzed, together with the FORMAT statement. If the source program contains no input/output statements, this pass is not invoked.

Phase CV

This phase processes the output from earlier phases. In order to assist subsequent processing, chains are constructed for PROCEDURE, ENTRY, BEGIN, CALL, ALLOCATE, and DECLARE statements.

DICTIONARY LOGICAL PHASE

The dictionary phase forms a dictionary of identifiers, by first analyzing PROCEDURE, BEGIN, DECLARE, and ENTRY statements. The text is then scanned for contextual use of identifiers, constants, and pictures. Finally, every identifier and constant in the source text is replaced by a reference to its respective dictionary entry. Dictionary entries are made during this phase for all implicitly defined identifiers. The formats of dictionary entries appear in Section 4.

Constructing and Accessing the Dictionary

The dictionary, during the construction stage, comprises two parts, the hash table and the dictionary proper.

To facilitate a search through the dictionary for an entry with a particular BCD, a method is used of dividing the dictionary into areas. Each area is characterized by a property of the BCD of each entry in it. In practice, these areas are not contiguous but are chained lists, each item in the list being one dictionary entry long.

The start of each list is in a table, known as the hash table. The association of a particular identifier with a list, i.e., the characterization of an area, is achieved by deriving from a given BCD an address in the hash table.

"Hashing" is a process of reducing the length of the internal representation of the BCD to one word. This is done by adding successive four-byte lengths of the BCD into one four-byte register. This is then divided by 211, and the remainder is doubled to give the hash table address associated with the particular BCD. All identifiers which hash to the same address are placed in a chain; in particular, all dictionary entries with the same BCD will be in the same hash chain.

If TOM, DICK, and HARRY occur in the same DECLARE statement in that order, and they all hash to the same address in the hash table, the address in the hash table will point to HARRY's entry, which contains the address of DICK, which, in turn, contains the address of TOM.

When no further BCD entries are to be made in the dictionary, and all BCD identifiers in the source text have been replaced by dictionary references, the hash table is deleted.

Testing for Consistent Attributes

A test is made at the start of each list of attributes, to ensure that any list of attributes at one level of factoring in a DECLARE statement is consistent.

Compiler Pseudo-Variables and Functions

Expressions specified for array bounds, string lengths, and initial value iteration factors must be evaluated at object time, or at allocation time if the variable is controlled. The expressions are placed temporarily at the end of the text, and are later moved by Phase FV and placed immediately following the BEGIN, PROCEDURE or ALLOCATE statement to which the declared variable belongs. The expression results are assigned to pseudo-variables generated by the compiler. These serve two purposes: first, the assignment statement appears as a normal PL/I statement and need not be treated as a special case; secondly, the pseudo-variable contains the dictionary reference of the variable and information concerning the destination of the expression. Compiler functions with a format similar to the pseudo-variables are also created. The function result is the specified array bound, or string length. Compiler functions are created for two purposes: first, to set bounds for base elements of structures when the structure bound is an expression, or to set the bounds of temporary arrays; and secondly, to set the storage address of a dynamically defined item immediately before its use. The formats of all the compiler pseudo-variables and functions appear in Section 4 under "Second File Statements."

Dictionary Entries for Entry Points

A PROCEDURE or ENTRY statement may have more than one label. Each label must have a data description to indicate the type of data returned when the label is invoked as a function, and also the type of data to which the expression in a RETURN (expression) must be converted. These need not be the same: there must therefore be provision for two data descriptions for each label. A PROCEDURE or ENTRY statement may specify parameters. The descriptions of these identifiers, obtained from DECLARE statements or default rules, are used for prologue construction, but not for parameter matching. Any data description given on these statements is to be used for conversion at a RETURN (expression), but not for determining the result returned by a function reference.

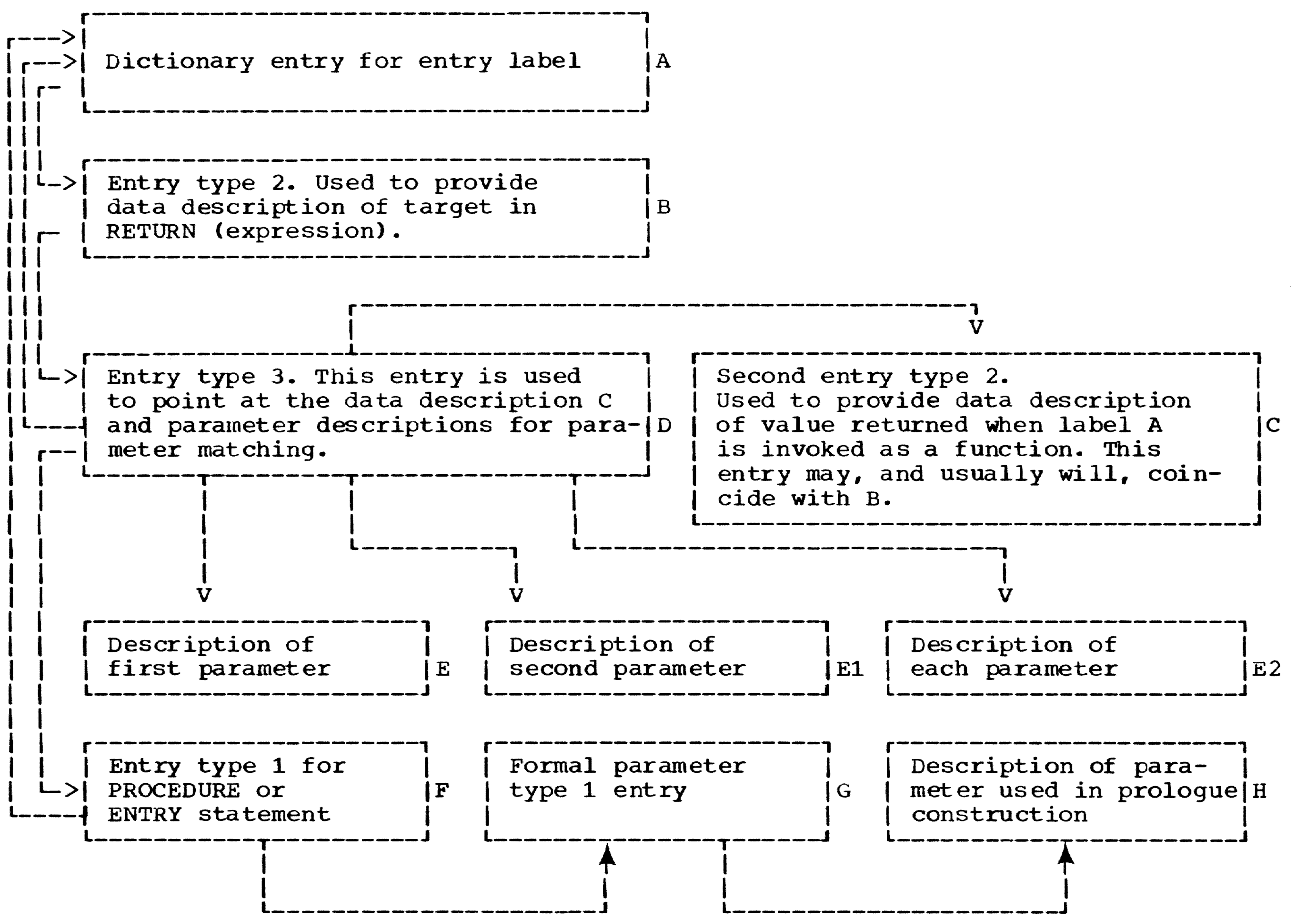
Parameter descriptions for use in parameter matching, and data descriptions used for determining the type of data returned by a function reference, may be specified by the source programmer in an ENTRY declaration. If these are not given, default and implicit rules must be used to build a data description, but no parameter description can be given.

Given the foregoing requirements, the dictionary entries describing an internal entry point are as given in Figure 9.

The set of dictionary entries A, B, C, D, E is repeated for each label associated with the PROCEDURE or ENTRY statement. The entry F will point to entry A for the first label only. D will point at the label with which it is associated. It should be noted that B and C may coincide.

The entries type 1 for PROCEDURE, ENTRY, and BEGIN statements are chained amongst themselves in the following way. Each entry type 1 belonging to a PROCEDURE or BEGIN statement contains the dictionary reference of the entry type 1, of the next PROCEDURE or BEGIN statement in the source program, and also of the entry type 1 of the immediately containing block.

The entries type 1 of PROCEDURE and ENTRY statements belonging to a single procedure are chained together in a circular manner. If there are no ENTRY statements the entry type 1 of the PROCEDURE statement points at itself.



Note: There is an entry E for each parameter described in D.

Figure 9. Dictionary Entries for an Internal Entry Point

External entry points are described by dictionary entries termed entry type 4. They contain data descriptions of the value returned when referenced as a function, and may contain descriptions of parameters.

Formal parameters which are entry points are termed entry type 5, and parameter descriptions which are entry points and are pointed at by types 3, 4, or 5 are termed entry type 6.

Phase ED

Phase ED contains a set of subroutines, for processing certain of the tasking and list processing attributes, and tables of generic and non-generic built-in functions. The phase obtains 1K of scratch storage, into which it moves the routines and tables, setting a slot in the communications region to point at them. This address is later picked up and used by phase EL.

Phase EG(EF)

Phase EG has two main functions. The first is to set up a hash table, and to insert the label entries left in the dictionary by the Read-In Phase into hash chains. The second function of the phase is to create dictionary entries for PROCEDURE, BEGIN, and ENTRY statements, and to construct chains linking entries of particular types.

For PROCEDURE-BEGIN statements, entry type 1 dictionary entries are created (see "Dictionary Entries for Entry Points" in Section 4), and block header chains are set up to link these entries sequentially. A containing block chain is also set up to link each entry with that of its containing block.

BEGIN statements are scanned for the ORDER/REORDER option, and the optimization byte is created in the entry type 1 (see "Dictionary Entries for Entry Points" in Section 4).

On the appearance of PROCEDURE statements, circular PROCEDURE-ENTRY chains are initialized to link the entry type 1 dictionary entries of the PROCEDURE and ENTRY statements of the same block. The formal parameter list is scanned, and formal parameter type 1 entries are created and inserted into the hash chain. Details of the PROCEDURE-ENTRY chains appear in Section 4.

The attribute list and the options are scanned and an options code byte and optimization byte are created in the entry type 1 (see Section 4). A check is then made for invalid and inconsistent attributes. CHARACTER and BIT attributes are processed, and second file statements (see Section 4) are created if necessary. Precision data are converted to binary, and dictionary entries are created for pictures (see Section 4).

Statement labels are scanned and their entry type 2 dictionary entries are created. The relevant data bytes in the dictionary are completed by default rules (see Section 4).

For ENTRY statements, entry type 1 dictionary entries are created (see Section 4), and the circular PROCEDURE-ENTRY chain is extended. Formal parameters, attributes, and labels are processed in a similar manner to those for PROCEDURE statements, except that the options code byte is not created.

Phase EI (EH, EJ)

Phase EI scans the chain of DECLARE statements set up by the Read-In Phase, and modifies the statements to assist Phase EK as follows:

Structure Level Numbers: these are converted to binary.

Factored Attributes: parentheses enclosing factored attributes are replaced by special code bytes, so that Phase EK can distinguish them easily. A factored attribute table is set up. It consists of slots corresponding to each factored level. Each slot contains the address of the attribute list associated with that level, and the address of the slot for the containing level.

The following attributes are processed:

DIMENSION: dimension table entries (see Section 4) are created in the dictionary and the source text is replaced by a pointer to the entry. Fixed bounds are converted to binary and inserted in the table. A second file statement (see Section 4) is created at the end of the text, for adjustable bounds, and a pointer to the statement is inserted in the dimension table. Identifiers with identical array bounds share the same dimension table.

PRECISION: precision and scale constants are converted to binary.

INITIAL: dictionary entries are created for INITIAL attributes.

INITIAL CALL: second file statements are created for INITIAL CALL attributes.

CHARACTER and BIT: fixed length constants are converted to binary; a code byte marker is left for * lengths (see Section 4). Second file statements (see Section 4) are created for adjustable length constants, and the source text is replaced by pointers to the statements.

DEFINED: second file statements (see Section 4) are created and the source text is replaced by pointers to the statements.

POSITION: the position constant is converted to binary.

PICTURE: a picture table entry (see Section 4) is created and inserted into the picture chain; similar pictures share the same picture table. The source text is replaced by a pointer to each entry.

USES and SETS: USES and SETS attributes are moved into dictionary entries, and pointers to the entries replace the source text.

LIKE: BCD entries are created for identifiers with the LIKE attribute.

LABEL: if the LABEL attribute has a list of statement label constants attached, a single dictionary entry is created. The dictionary entry contains the dictionary references of the statement label constants in the list.

OFFSET and BASED: Second file statements are made and text references are inserted in the DECLARE statements for these attributes.

AREA: Fixed-length specifications are converted to binary; second file statements are made for expressions; a code byte, followed by the length of text reference, is inserted in the DECLARE statement text.

All other attributes, identifiers, or constants are skipped.

Phase EL (EK, EM)

Phase EL, consisting of modules EK, EL, and EM, scans the chain of DECLARE statements constructed by the Read-In Phase.

An area of storage known as the attribute collection area is reserved. This is used to store information about the identifiers, and has entries of a similar format to that for dictionary entries.

Complete dictionary entries are constructed for every identifier found in a DECLARE statement. These identifiers can be one of the following types:

1. Data Items (see Section 4)
2. Structures (in this case, the 'true' level number is calculated) (see Section 4)
3. Label Variables (see Section 4)
4. Files (see Section 4)
5. Entry Points (see Section 4)
6. Parameters (see Section 4)
7. Event Variables
8. Task Variables.

Identifiers appearing as multiple declarations are rejected and a diagnostic message is given.

The attributes to be associated with each identifier are picked up in three ways.

First, the attributes immediately following the identifier are stored in the attribute collection area.

Secondly, any factored attributes and structure level numbers are examined. These are found by using the list of addresses placed in scratch storage by Phase EI. Each applicable attribute is marked in the attribute collection area, and any other information, e.g., dimension table address, or picture table address, is moved into a standard location in the attribute collection area. All conflicting attributes are rejected and diagnostic messages are given.

Finally, any attributes which are required by the identifier, and which have not been declared, are obtained from the default rules.

After the dictionary entry has been made, further processing (e.g., linking of chains, etc.) must be done in the following cases:

1. DEFINED data
2. Data with the LIKE attribute
3. Files
4. Strings with adjustable lengths
5. Arrays having adjustable bounds
6. GENERIC identifiers
7. Structure members
8. Identifiers with INITIAL CALL
9. Identifiers with the INITIAL attribute

After the declaration list has been fully scanned and processed, it is erased.

Phase EP

Phase EP first conditionally marks later phases as 'wanted' or 'not wanted,' according to how certain flags in the dictionary are set on or off. This assists in the load-ahead technique.

The entry type 1 chain in the dictionary is then scanned. For each PROCEDURE entry in the chain, each entry label is examined for a completed declaration of the type of data the entry point will return when invoked as a function. If this has previously been given in a DECLARE statement nothing further is done, otherwise entry type 2 and 3 dictionary entries are constructed from default rules (see Section 4). If this default data description does not agree with the description derived from the PROCEDURE or ENTRY statement, a warning message is generated.

At each PROCEDURE entry, the chain to the ENTRY statement entry type 1 is followed. Each statement is treated in a similar manner to that for a PROCEDURE entry type 1.

The CALL chain is then scanned and, at each point in the chain, the dictionary is searched for the identifier being called. If the correct one is not found, a dictionary entry for an EXTERNAL procedure is made (see Section 4), using default rules for data description. Before making the entry, the identifier is checked for agreement with any of the built-in function names. If there is agreement, a diagnostic message is generated, and a dummy dictionary reference is inserted.

If an identifier is found, it is examined to see if it is an undefined formal parameter. If it is, the formal parameter is made into an entry point, again using default rules for data description. If it is not, or if the declaration of the formal parameter is complete, the type of entry is checked for the legality of the call. A diagnostic message is generated if the item may not be called. In all cases, the item called is marked IRREDUCIBLE if it has not previously been declared REDUCIBLE.

Phase EW (EV)

Phase EW is an optional phase, loaded only if any LIKE attributes appear in the source program.

This phase scans the LIKE chain which has been constructed by Phase EK, and completes the dictionary entry for any structure containing a LIKE reference. When a structure in the LIKE chain is found, its validity is checked, and dimension data and inherited information are saved. The dictionary is scanned for the reference of the "likened" structure and the entry is checked for validity.

This dictionary entry (see Section 4) is copied into the dictionary, with alterations if there is a difference between the original structure and this structure with regard to dimensioned data. If both structures have dimensions a straight copy is made; if the structure with the LIKE attribute has dimensions and the likened structure has not, the dimension information is added to the copy; if the structure with the LIKE attribute is not dimensioned and the likened structure is, then the dimension data is deleted from the copy. Inherited data is added to the copy. If an error is found, the structure with the LIKE attribute is deleted and a base element copy of the master structure is inserted instead. Where copies of entries occur which refer to dimension tables with variable dimensions, the dimension table entry is copied, and new second file dictionary entries and statements are created. Similar entries must be made if the structure item has been declared to be an adjustable length string, or has been declared with the INITIAL attribute.

Finally, the newly completed structure is scanned by the ALIGN routine in phase EV, to provide correct explicit/inherited/default alignment attributes for its base elements.

Phase EY

Phase EY is an optional phase which processes all ALLOCATE statements.

The second file is scanned first and all pointers to the dictionary are reversed. All ALLOCATE statements using the DECLARE chain are then scanned, and the dictionary references of allocated items are obtained by hashing the respective BCD of each item. The attributes given on the ALLOCATE statement for an item are collected together.

A copy of the dictionary entry of the allocated item is then made (see Section 4), and the ALLOCATE statement is set to point to it. The dictionary entry is completed by including any attributes given on the ALLOCATE statement, and copying any second file statements from the DECLARE chain which are not overriden by the ALLOCATE statement.

In the case of an ALLOCATE statement in which a based variable is declared, no copy of the original dictionary entry is required. The BCD is replaced by the original dictionary reference.

All pointer qualified references in the text are checked to determine that the qualified variable is based. For every occurrence of a variable with a different pointer a new dictionary entry is made. If the variable is a structure the entire structure is copied. A PEXP second file statement is made for the pointer and the 'defined' slot in the new dictionary entry is set to point to it instead of to the declared pointer.

The BCD of the pointer and the based variable in the text are replaced by the new dictionary reference followed by padding of blanks which will be removed by phase FA.

The based variable can be the qualified name of a structure member. If this is so, the name is checked for validity. Only the first part or lowest level of the qualified name in the text is replaced by the dictionary reference of the member. It is preceded by a special marker to tell phase FA that a partially replaced name follows.

Phase FA

Phase FA scans the text sequentially. If, during the scan, qualified names are found with subscripts attached, they are reordered so that a single subscript list appears after the base element name. The dictionary is scanned and references obtained for any identifiers which are contextually, file, event, pointer variables, or programmer-named ON conditions. If no reference is available, a new dictionary entry is made. The identifier is then replaced in the text by the dictionary reference.

If a constant marker is found, the dictionary is scanned to check if the constant is present. If it is not, a new dictionary entry is made (see Section 4) and the resulting reference replaces the constant in the text.

If a P FORMAT marker is found, the dictionary is scanned for a picture entry in agreement. If there is no agreeing entry, a new dictionary entry is made (see Section 4) and the picture chain is updated. The dictionary reference replaces the format marker in the text.

The CALL chain is removed from CALL statements. The appearance of PROCEDURE, BEGIN, END, and DO statements results in adjustments to the level and count stacks. If statement introduction code bytes appear (such as SN, SL, CL, and SN2), the current statement number is updated. All data items associated with the PROCEDURE, BEGIN, ENTRY, and DECLARE statements are removed, leaving only the statement identification and the keyword.

Phase FE

When an identifier is found, the hash chain is used to scan the dictionary for a valid entry. If one is found, its dictionary reference replaces the identifier in the output text. If no valid entry is found, and the BCD does not agree with any entry in the tables of BCDs of PL/I built-in functions, then a dictionary entry is made as if the identifier was declared in the outermost procedure. However, if the BCD agrees with a function name, and it is not in a SETS position, a function entry is made in the dictionary, and its reference is used to replace the identifier.

If a left parenthesis is found, the previous dictionary entry is checked for an array, function, or pseudo-variable. If it is one of these, the relevant marker is inserted in the text before the parenthesis (see Section 4).

Checks are also made for the positions of function references in assignment statements. Any dictionary references encountered in the input file are moved directly to the output file.

PROCEDURE, BEGIN, DO, and END statements cause the current level count to be updated.

Phase FI

Phase FI scans the text and checks, where possible, the validity of dictionary references found. References in a GOTO statement are checked that they refer to labels or label variables and that the subsequent branch is valid. The code byte for GOTO is changed to GOOB (see Section 4) if the branch is to a label constant outside the current PROC or BEGIN block. If the branch is to a label variable, GOOB is set up unless a label value list was given at the declaration, and all members of the list lie within the current block.

List processing based variables in ALLOCATE, FREE, READ, WRITE, and LOCATE statements are marked as requiring a Record Dope Vector (RDV). Variables in TASK and EVENT options on CALL statements are checked for validity.

References are checked if they appear where a file is expected. Items in data lists are checked for validity, and Data Element Descriptors (DEDs) and symbol bits are set on for all variables found in the lists.

Any errors which are found cause diagnostic messages to be generated and dummy references to be placed in the text in place of erroneous references.

Phase FK

Phase FK scans the attribute collection area for entries with the SETS attribute. The SETS lists in the dictionary entries are scanned, and their syntax checked. Identifiers are counted and replaced by their dictionary references. Constants are counted, converted to binary, and arranged in ascending order in the dictionary entry.

Phase FO

Phase FO makes a dictionary entry for each ON condition mentioned inside a block. For ON CHECK conditions multiple dictionary entries are made (see Section 4), one for each BCD. If a similar condition is mentioned more than once in a block, only one dictionary entry is made for that condition, except for file conditions, ON CONDITION, and ON CHECK, when separate dictionary entries are made for each different BCD name.

SIGNAL and REVERT statements are treated in a similar manner to ON statements.

The dictionary entries for each BCD name associated with file or CONDITION conditions are checked and, if in error, the ON, SIGNAL, or REVERT statement is replaced by an error statement. A diagnostic message is generated.

The BCD name of each file entry referred to in ON, SIGNAL, and REVERT statements is examined. If the BCD is PLIINPUT or PLILOAD, the dictionary reference of the file entry is placed in a slot in the communications region.

A check is made to ensure that formal parameters do not appear in CHECK and NOCHECK lists. A single dictionary entry is created for each CHECK and NOCHECK list and a pointer to the entry is placed in the relevant entry type 1.

When dictionary entries are made for CHECK lists, one of three different check codes is used depending on whether the BCD is an ENTRY LABEL, a LABEL CONSTANT, or a variable.

List Processing POINTER and OFFSET variables in CHECK lists are treated as data variables. BASED variables may not appear in CHECK lists.

A dictionary entry is made for the list processing AREA condition. This condition is always enabled and may not appear in a condition prefix.

Dictionary entries are also created for each ON condition which is disabled for a particular PROCEDURE or BEGIN block, and for each ON condition whose status is changed within the block. Pointers to these dictionary entries are placed in the relevant entry type 1.

All dictionary entries for ON conditions are placed in the AUTOMATIC chain for the relevant PROCEDURE or BEGIN block.

A further, quite distinct, function of this phase is to substitute error statements for all statements containing dummy dictionary references (which have been inserted by previous phases on detecting a severe error). If a dummy reference is found in the second file, the compilation is aborted.

Wherever an element of a label array is initialized by appearing as a statement label, an assignment to a compiler label has been inserted by the Read-In phase. Phase FO checks the validity of each such assignment; for each array with this type of initialization, a second file dictionary entry is made, and all assignments to the array are chained.

Phase FQ

Phase FQ checks the validity of each item in the PICTURE chain in the dictionary (see Section 4).

The precision for each correct picture is calculated, together with its apparent length, and stored in its dictionary entry. A data byte is created in the entry for use by Phase FT.

Invalid pictures cause appropriate diagnostic messages to be generated.

Phase FT

Phase FT performs certain housekeeping tasks. These are as follows:

1. The second file entries are scanned and pointers to each entry are inserted in the associated dictionary entry (see Section 4).
2. Each item which has a storage class is inserted into the appropriate chain for that class (see Section 4).
3. Constants are placed in the constants chain and their apparent precision is calculated. Sterling constants are converted to pence.

4. Dimension tables are separated for items which are not in structures, but which are arrays having similar bounds, but with different element lengths.

5. Items which are members of structures and which have "inherited" dimensions, i.e. are contained in a structure which itself is dimensioned, are made to inherit their dimensions. If a base element of a structure inherits dimensions which are not constant, second file statements (see Section 4) are set up to initialize the bounds in the object time dope vector.

6. Items which have expressions to be evaluated at prologue time, e.g., parameter descriptions for entry points and defined items, are placed in the AUTOMATIC chain for the appropriate block.

7. The dictionary entry for any item described by a picture is expanded by the precision and scale or string length, extracted from the picture table entry. Identifiers of different modes sharing the same picture table are now placed in separate tables.

8. The 'dope vector required' bit (see Section 4) is set on where necessary.

9. When a label array is found which has initial label statements for any of its elements, the chained statements are moved into the second file. The original statement is left in the text, to be removed by Phase FV.

10. Dictionary entries similar to label BCD entries are made for all TASK variables.

Phase FV

Phase FV scans the second file and reverses the pointers to the dictionary.

Dictionary entries for DEFINED data are completed (see Section 4). Overlay and correspondence defining are differentiated between, as are static and dynamic defining. A preliminary check of the validity of defining is also carried out.

When PROCEDURE and BEGIN statements are encountered, any second file statements associated with data in the AUTOMATIC chain for that block are inserted in the text following such statements.

When ALLOCATE statements are found, any second file statements associated with the item being allocated are inserted in the text following the statement.

When a reference to dynamically defined data is found, the base reference is inserted into the text following the defined reference.

When an initial label statement is encountered in the main text, it is not copied into the output string.

The dictionary reference of a POINTER in a PEXP (pointer expression) second file statement is inserted into the defined slot of the associated based variable. If the based variable is a structure this reference is propagated throughout the structure. The PEXP statement is then deleted.

A similar procedure is performed for BVEXP (based variable expression) second file statements whereby the dictionary reference of the AREA is inserted into the dictionary entry of the associated OFFSET variable.

ADV second file statements referring to a BASED variable are checked for compliance with the compiler implementation rules. If the rules are obeyed, the dictionary entry of the 'bound' variable is inserted in the appropriate slot in the multiple table entry.

If an MTF statement refers to a based variable the appropriate bound slot is copied from one multiple table entry to the other.

Phase FX

Phase FX is an optional phase entered only if the ATR (attribute list) or XREF (cross-reference list) option is specified. It scans the STATIC, AUTOMATIC, and CONTROLLED chains, and the formal parameter lists.

For each identifier it creates an entry in text scratch storage of the form:

2 bytes	3 bytes	3 bytes
Dictionary reference	Text reference to this item	Text chain

This entry is inserted into a chain of similar entries in the alphabetical order of the BCD of the identifier.

If the XREF option is specified, the text is scanned for dictionary references. When the dictionary reference of an identifier is found in the text, an entry is created in a chain of entries from the dictionary entry of the identifier. If the identifier is that of a BASED item, an entry is also created in a chain of entries from the dictionary entry of the associated pointer.

Each chain member thus represents a text reference to an identifier and has the form:

2 bytes	3 bytes
Statement number	Text chain

Each reference chain for an identifier is in text scratch storage.

The sorted chain of identifiers is then scanned, and for each entry in the chain the following actions take place:

1. The statement number of the DECLARE statement, if any, in which the identifier was declared is printed

2. The BCD of the identifier is printed.

3. If the ATR option is specified, the dictionary entry of the identifier is analyzed and its attributes are printed. For variables having constant dimensions and/or constant string lengths, these dimensions and lengths are printed.

   Except for file attributes, the attributes printed will be those obtaining after conflicts have been resolved and defaults applied. Since the file attribute analysis does not take place until after the attribute list has been prepared (see Phase GA), file attributes in the list are those supplied by the programmer, regardless of conflicts.

4. If the XREF option is specified, the reference chain for the identifier is scanned, and the statement number contained in each entry is printed

Finally, all scratch storage is released and control is passed to the syntax check option phase.

Phase F1

Phase F1 is entered at the end of the dictionary phase. It tests the syntax check option flag which was set by module AB. If the flag indicates that the option is in effect at the "TERMINAL" error level, control is passed immediately to the next phase. If the option is in effect at the "SEVERE" or "ERROR" level, F1 checks to see if any such errors were found during the read-in and/or dictionary phases. If they were, F1 either terminates the compilation (nonconversational tasks) or issues a message to the terminal (conversational tasks) asking the user if he wishes to terminate or continue compilation. If no errors of the specified or greater level were found, control is passed to the next phase. F1 issues diagnostic messages describing the action taken.

PRETRANSLATOR LOGICAL PHASE

The purpose of the Pretranslator Phase is to expand those statements in the language that can be broken down into simpler statements, and to insert explicitly generated statements in place of implied ones.

Second level markers (see Section 4) are removed from internal compiler codes, and some of the I/O statements are changed into a form more suitable for the pseudo-code phase.

Argument lists are examined and the matching of arguments with parameter descriptions takes place, with temporary variables being created where necessary, e.g., where data conversions are required.

If the compilation contains ON CHECK conditions the appropriate calls to the library routine are provided.

Any structure assignments containing the BY NAME option are processed.

If any structure assignment statements or structures in I/O lists are detected in the program, they are expanded into scalar assignments and DO groups.

If the program contains any array assignments, or array expressions in I/O lists, these are expanded into DO loops and scalar assignments or expressions.

If the program contains iSUB references, the subscripts are computed for the base array corresponding to the subscripts given for the defined array.

Additions to the Text

In addition to changing the content of the text, the Pretranslator introduces some new symbols and grammatical forms into the source text. These are as follows:

The Umbrella Symbol: this is designated by the symbol code X'5E', which is used to introduce a literal as an operand. It is used only as a bound of a DO loop, or in a call of the dope vector pseudo-variable.

Statements within statements: a list of statements may be introduced within another statement. In this case the inserted list is enclosed in parentheses. Statements in the list are given no statement number field, but they have semi-colons at the end.

I/O statements: the form of I/O statements is changed considerably during the pretranslator phases, as explained in the description of Phase GB.

BUY and SELL statements: special statements are introduced for manipulating temporary storage at object time; they have a form similar to ALLOCATE and FREE statements.

Temporary Storage: Pretranslator phases create temporary variables for function and procedure calls where the arguments do not match the final parameters, where expressions appear as arguments, for control variables for DO loops in array and structure assignments, and for iSUB defined subscript lists. The Pretranslator has no

mechanism for evaluating expressions. Therefore, temporaries which have no data type are created for expression arguments with no parameter description. Such temporaries are known as 'chameleon' temporaries. The data type of these chameleon temporaries is completed by the Translator generic phase when the resultant data type of the expression has been determined.

When the Pretranslator creates a temporary from an argument which contains any array with adjustable bounds or adjustable string length, compiler functions (see Section 4) are generated in-line, to set up the adjustable quantities at object time, to enable storage of the correct size to be acquired by means of the BUY statement.

The temporary variables created by the Pretranslator have dictionary entries similar to variables declared in the source program, except that the temporaries do not have BCD names.

Phase GA

Phase GA is an optional phase which scans the STATIC chain for file constants and OPEN control block entries.

For file constants a DECLARE control block is constructed from the file name and attributes, while checking the attributes for consistency. For file constants with the ENVIRONMENT option a dictionary entry is constructed, chained from the file constant, containing the storage image of the 56-byte DECLARE control block.

For OPEN control block entries an OPEN control block is constructed from the attributes in the entry, a check is made for consistency, and another dictionary entry, chained from the OPEN control block entry, is constructed. This new entry contains the 8-byte storage image of the OPEN control block.

When the COBOL option is encountered in the ENVIRONMENT string of a FILE statement, phase GA sets the low-order bit in the fifteenth byte of the FILE dictionary entry. Although this action overwrites the dictionary reference of the ENVIRONMENT string, it is permissible since GA is the only phase which processes this string.

The EXCLUSIVE second level marker is recognised in the file attribute dictionary entry during the diagnostic check and construction of the DCLCB or the OCB.

Phase GB (GC)

Phase GB, containing Modules GB and GC, processes I/O statements. GB removes all second level markers from internal character codes (see Section 4). It then reorders the options so that either EDIT, DATA, or LIST options appear last.

In data lists the DO specification is moved so that it precedes the relevant list, and the END statement is added.

In format lists iteration factors are expanded.

RECORD I/O statements for which the COBOL file option is recognized are examined for validity by GC. Diagnostics are put out for LOCATE and READ SET statements for which COBOL files are used. A temporary variable is created to assist such data transfers as occur when a COBOL record is read into or written from a structure which does not consist entirely of one of the following:

- doubleword data
- fullword data
- halfword binary data
- character string data
- aligned bit string data
- a mixture of character string and aligned bit string data

I/O activity found within a PROCEDURE or BEGIN block causes the bit X'10' to be set to one in the optimization byte of its entry type 1.

Phase GK

Phase GK scans the source text for function references. If it finds one, it inserts a special marker byte before the argument list, followed by:

1. Two code bytes giving information about the type of function, and whether it was called with the TASK option
2. The current statement number
3. The current block level and count

This phase also inserts a special argument marker before each argument in the list, followed by the reference of the corresponding parameter and a code byte to show whether or not the argument is specified in a SETS list. The number of arguments present is checked against the number given as required by the corresponding dictionary entry.

NULL, NULLO, and EMPTY built-in functions are recognized and converted to constants.

Phase GO

This phase acts as a pre-processor for phase GP.

Phase GP

Phase GP scans the text for procedure and function calls with arguments. These are detected by the special markers inserted by Phase GK.

Temporaries (see Section 4) are created for any arguments which are expressions. (An expression is defined as being any sequence of variables and operators, other than single variables followed only by a subscript list, or only by a defined subscript list and then a subscript list). If a parameter description has been declared in an entry declaration, the temporary which is created is of the same type as the parameter description. Otherwise, a 'chameleon' temporary of unspecified data type is created, its type being subsequently completed when the expression type has been determined by the translator generic phase.

Expressions are scanned for arrays (including partially subscripted arrays), structures, or the end of the expression, in order to determine the highest form of aggregate in the expression, so that the correct type of temporary may be created.

Where the expression contains a partially subscripted array, a temporary is created with a dimensionality equal to the number of cross sections specified in the subscript list.

When single arguments are specified together with parameter descriptions, the arguments are compared with the parameter description. If there is a lack of match, action may be taken in one of two ways.

1. If the data types are compatible, a warning message is printed, and a temporary is created

2. If the data types are incompatible, an error message is printed, and the parameter description is ignored

When the argument is a single partially subscripted array which matches the parameter, a special temporary is created which has the same dimensionality as the number of cross sections in the subscript list, and it appears to be defined upon the original argument. Code is then generated to initialize the temporaries, multipliers, and virtual origin from the dope vector of the original argument and the subscript list.

Whenever a temporary is created, a BUY statement contained in nested statement brackets is inserted in the output text, followed by the assignment of the expression or non-matching argument to the temporary. After the end of the PROCEDURE or function call, all the temporaries generated in the call are released by means of a SELL statement in nested statement brackets.

In all argument temporaries created by phase GP, other than those created for constants, a special flag bit is set on (see Section 4), but in the case of temporaries created for arguments to built-in functions, this bit is turned off by phase IM. This bit is used in phase QU when halfword instructions replace fullword instructions in the manipulation of halfword binary operands which are temporary arguments.

Temporaries are created for constants which are specified as arguments to functions defined by the programmer.

If a TASK, EVENT, or PRIORITY option is present in a CALL statement, then any temporaries which are created are of the 'not sold' type.

If GENERIC entry labels are specified as arguments to procedures, a special dictionary entry is made which contains the argument and parameter description dictionary references, to enable the Translator generic phase to select the correct generic member.

A warning message is printed whenever a temporary is created for an item declared in a SETS list.

When subscript lists for the number of cross sections are being checked, a severe error message is printed if a subscript list contains too many subscripts, and the statement is deleted.

Phase GU

Phase GU scans the source text for PROCEDURE, BEGIN, and END statements, and for statements that may raise a possible CHECK condition.

A list of all items currently checked is extracted from the CHECK and NOCHECK lists present in PROCEDURE and BEGIN statements.

Items contained in statements that may raise a CHECK condition are examined and compared with the list of currently checked items. If the item appears in the list, a SIGNAL CHECK statement is created for it, either before the statement concerned (for labels and entry names) or after it (for variables).

Phase HF

The purpose of phase HF is to detect structure assignment statements, possible structure expressions in data lists in GET and PUT statements, and nested statements, in particular nested structure assignments.

The leftmost structure in an expression or assignment is used as a basis for comparison, and if similar structuring is not found throughout the expression or assignment, diagnostic messages are issued. Any expression containing no structures is left unchanged.

The base elements of the structures are found, and if the referenced structures are dimensioned, a temporary is created for each dimension. It is then added to the AUTOMATIC chain for the appropriate block. Iterative DO loops are constructed, with the temporaries iterating between the upper and lower bounds of that particular dimension. Base elements are assigned, with the temporaries as subscripts, and with scalars remaining unchanged. END statements are created for the DO loops, and SELL statements for the temporaries. The statements which have been created are nested within the original statement.

Phase HK

The purpose of Phase HK is to detect array or scalar assignments, possible array expressions in I/O lists in GET and PUT statements, and nested statements, in particular nested assignment statements.

The leftmost array in an expression, or the leftmost array or scalar in an assignment is used as a basis for comparison, and if similar dimensions or bounds are not found in the array references, diagnostic messages are issued. Any expression containing only scalars is left unchanged.

For unsubscripted arrays which are equally spaced in storage only one temporary is bought. For all other arrays a temporary is bought for each dimension, except in the case of certain partially subscripted arrays where the number may be minimized. Each temporary will be added to the AUTOMATIC chain for the appropriate block. If the ON-condition name SUBSCRIPTRANGE is enabled for any statement, a temporary will be bought for each dimension in all cases. Iterative DO loops are constructed: for an unsubscripted array expression of dimensionality N, the temporary will iterate between the lower bound of the Nth dimension and an evaluated product so that all elements of the array are processed; while for other arrays the temporaries will iterate between the lower and upper bound of the particular dimension of the array. The assignment statement is added to the output string with additional subscripts where necessary. End statements are created for the DO loops, and SELL statements for the temporaries. The statements which have been created are nested within the original statement.

The syntax of pseudo-variables is also checked.

Phase HP

Phase HP scans the source text for references to items defined using iSUBs. For each reference found, the subscripts are computed for the base array corresponding to the subscripts given for the defined array.

The subscripts of the defined array are assigned to temporaries specially created for this purpose, which are then used to replace the iSUBs in the defining subscript list. The base array, with the subscript list so formed, replaces the defined array in the text.

TRANSLATOR LOGICAL PHASE

The Translator phase consists of two physical phases, the stacker phase and the generic phase. The purpose of the translator is to convert the output from the Pretranslator into a series of "triples" (see Section 4). A "triple" is in the form of an operator followed normally by two operands.

The translation is achieved by using a double stack, with one part for operators, and the other part for operands, and assigning two weights to each operator. One weight (the stack weight) applies to the operator while it is in the stack, and the other weight (the compare weight) applies when the operator is obtained from the input string.

When an operator is obtained from the input string it is compared with the top stack operator. Depending on the result of the comparison, one or other of the two operators is switched on to determine what action is next to be performed. Apart from some special cases, this action is usually either to continue to fill the stack, or to generate a triple. The special cases lead to various manipulations of the stack items, after which the translation process continues.

For the purposes of translation, the input text to the translator is considered to consist of operators and operands only. This means that I/O options, etc., are regarded as operators.

After translation, the text string consists of operands and operators. All statements start with an operator to indicate a statement number or label, followed by the statement type, which may be a single operator, as in the case of RETURN or STOP, or which may be an operator such as a function or subscript marker, followed by a list of arguments. This list may also include compiler generated statements, e.g., DO loops for I/O lists. All I/O options are regarded as operators and require no markers before them. The end of the source text will be marked by a special operator, and compiler generated code, which may follow this end-of-program marker, will appear between the marker and the special second-end-of-program marker. The end of a block of text will be marked by an EOB operator. The program is now assumed to be syntactically correct.

Phase IA

Phase IA rearranges the source text into a prefix form, in which parentheses and statement delimiters have been removed, and the operations within a statement have been so arranged that those with the highest priority appear first.

As operators and operands are encountered, they are stored in stacks. Tables give the priority of each operator as it appears in the input text and in its stack.

When an operator is found during the scan of the source text, its compare weight (see Section 4) is tested against the stack weight of the top operator in the stack. If the compare weight is the lesser of the two, then action is taken according to the compare operator. This is referred to as the compare action. Similarly, if the compare weight for the current operator found in the scan is greater than or equal to the stack weight of the top stack operator, action is taken according to the top stack operator. This is referred to as the stack action. Normally, the compare action is to place the compare operator in the stack, and to continue the scan, placing any subsequent operand in the stack until another operator is found. The normal stack action is to generate a triple, consisting of the top operator in the stack and the top two operands, eliminating the items from the stack, and inserting a special flag as the operand of the triple which is now at the top of the stack. The source (compare) item is then compared with the new top stack item.

The output text of the stacking phase is in the form of a series of triples, i.e., statement types with no operands, and operators with one or two operands. If the result of a triple operation is to be used in a later triple, the appropriate result is flagged accordingly.

Certain phases are marked wanted or not wanted at this stage. If the source text contains an invocation by CALL or function reference, Phases IL and IM are marked wanted. If it does not, Phases IL, IM, IN, IO, IP, IQ, MG, MH, MI, MJ, MK, MM, MN, and MO are marked not wanted. Phases MB and MC are marked wanted when the source text contains pseudo-variables or multiple assignments; otherwise, they are marked not wanted. The DO loop processing phases (LG and LH) are marked in co-operation with the dynamic initialization phases (LB and LC). If LB and LC are requested, the marking of LG and LH is left until that stage of compilation; otherwise, LG and LH are marked by Phase IA independently.

When ALLOCATE and FREE statements occur, phase NG is marked wanted. When LOCATE statements occur, phase NJ is marked wanted.

Phase IG

Phase IG is an optional phase which is loaded to process array and structure arguments to built-in functions. When aggregate arguments are given for built-in functions they are expanded by the structure and array assignment phases so that the built-in functions appear as base elements, subscripted where necessary.

Phase GP examines these arguments, and ascertains whether it is necessary to create a dummy. If it is necessary, a scalar dummy is created, but the assignment of the argument expression is not inserted in the text, as this would be an invalid aggregate assignment.

Phase IG examines the text for a BUY statement for a dummy for an aggregate argument to a built-in function, and then inserts an assignment triple in the correct place in the text.

Phase IK

This phase immediately precedes the phase IL and shares with it the initialization processes required by the main generic phase IM. It obtains text block storage and moves into it routines and a table that will be used later by the main generic phase. Part of the storage is reserved for use by the main generic phase as a nested function stack area. Control is passed to phase IL.

Phase IL

This phase immediately precedes the main generic phase IM and completes the initialization process begun by phase IK. It obtains 4K bytes of scratch storage and places in it the entire built-in function table and a list of constants used by the main generic phase. Registers are set to point to the built-in function table, to the list of constants, and to the nested function stack area reserved by phase IK. Further text block storage is obtained for use by the main generic phase and a register is set to point to it. Control is passed to phase IM.

Phase IM

This phase is the main generic processor. It scans the source text for procedure invocations by a CALL statement, procedure or library invocations by a function reference, and assignments to "chameleon" dummy arguments (see Phase GP).

Any procedure which is generic and is invoked by a CALL statement or function reference is replaced by the appropriate family member. If the invoked procedure is non-generic, it is ignored. A generic library routine invoked by a function reference is also replaced by the appropriate family member.

The arguments passed to library routines are checked for number and type, and a conversion inserted where necessary and possible.

The type and location of the result of all function invocations is placed in the text which follows the end of the text which invoked the function. The resulting type of an expression assigned to a "chameleon" dummy is determined and set in the dictionary entry which relates to the dummy.

The argument bit, set on for all argument temporaries created by phase GP, is turned off for arguments of built-in functions.

Phase IT

Phase IT scans the source text for function triples and, in particular, the built-in functions for which code will be generated in-line. Further tests are made to detect the functions which, according to the method used to generate in-line code, are optimizable. This applies only to the SUBSTR, UNSPEC, and INDEX functions. All references to 'chameleon' temporary assignments within the scope of these functions are removed subject to certain restrictions imposed by the function nesting situation.

Phase IX

Phase IX checks that POINTER and AREA references are used as specified by the language. This phase is loaded only if POINTER or AREA references are found, declared either explicitly or contextually. Error messages are produced if errors are found and the statement in error is erased.

Data type triples in the text are scanned and a stack of temporary results is created containing the values:

X'40' for POINTER
X'02' for AREA
X'00' for any other data type

The maximum permitted number of temporaries at any one point in a program is 200. The compilation is terminated if this figure is exceeded.

Phase JD

Phase JD scans the text for concatenation and unary prefixed triples with constant operands. These are evaluated and the results are placed in new dictionary entries. The references are passed through a stack into the corresponding result slots in the text.

AGGREGATES LOGICAL PHASE

The aggregates phase consists of three physical phases, the preprocessor (phase JI), the structure processor (phase JK) and the DEFINED chain check (phase JP).

The structure processor phase carries out the mapping of structures and arrays in order to align elements on their correct storage boundaries.

The DEFINED chain check ensures that items DEFINED on arrays and structures can be mapped consistently.

Phase JI

The first function of phase JI is to obtain scratch storage in which the text skeletons contained in phase JJ are to be held. Phase JJ is then loaded, and its contents are moved to the scratch storage for subsequent use by phases JI and JK. Phase JJ is then released and control is returned to phase JI.

The main function of phase JI is to expedite data interchange activities. A scan of static, automatic, and controlled chains is performed. The chains are reordered so that all data variables appear before non-data items. Adjustable PL/I structures and arrays are detected. Each entry in the COBOL chain is mapped as far as possible at compile-time, removed from the chain, and placed in the appropriate AUTOMATIC chain.

Phase JK

This phase scans the AUTOMATIC, STATIC, and CONTROLLED chains for arrays, structures (including COBOL structures), adjustable length strings, DEFINED items, AREA, and POINTER arrays and structures, TASK and EVENT arrays, and TASK and EVENT arrays in structures.

For the base elements of structures without adjustable bounds or string lengths, the following calculations are made:

- The offset from the start of the major structure
- The padding required to align the elements on the correct boundary
- All multipliers of arrays of structures.

For all minor structures and major structures the following calculations are made:

- Size
- The offset from the preceding alignment boundary with the same value as the maximum appearing in the structure

Where a structure contains adjustable bounds or string lengths, code is generated to call the Library at object time.

For arrays, the multipliers are calculated, unless the array contains adjustable items, in which case the Library performs the calculations.

For adjustable structures, arrays, or strings, code is generated to add a symbolic accumulator register into the virtual origin slot of the dope vector, and the accumulator register is incremented by the size of the item.

Calculations are made in a similar fashion for arrays of strings (in structures or otherwise) with the VARYING attribute. In addition, code is generated to set up an array of string dope vectors which refer to the individual strings in the array using the dope vector. Code is also generated to convert the original dope vector to refer to the array of string dope vectors, instead of to the storage for the array.

The routine which generates code for arrays of VARYING strings is also used to generate code for the initialization of arrays of TASK, EVENT, and AREA variables.

DEFINED items are processed in the following way:

- Code is generated to set the multipliers and virtual origin address of correspondence defined arrays without iSUBs in the dope vector of the DEFINED items from the defining base dope vector.
- Code is generated for overlay DEFINED items if they do not fall into the class which is to be addressed directly. The code first maps the DEFINED item, if necessary, calculates the address of the start of the storage to be used by the DEFINED item, and finally, relocates the DEFINED item using this address.

Dope vector descriptor dictionary entries and record dope vector dictionary entries are made for items which need to be mapped at object time, or which appear in RECORD-oriented input/output statements.

Phase JP

Phase JP scans the DEFINED chain, and differentiates between the following:

1. Correspondence defining
2. Scalar overlay defining
3. Undimensioned structure overlay defining
4. Mixed scalar-array-structure-string class overlay defining

In correspondence defining, this phase differentiates between arrays of scalars and arrays of structures. It also checks that the elements of the defined item which may validly overlay the elements of the base belong to the same defining class, and that the base is contiguous.

In scalar overlay defining, this phase checks that the defined item may validly overlay the base.

For undimensioned structure overlay defining, this phase checks that the elements of the defined item may validly overlay the elements of the base.

For mixed scalar-array-structure-string class overlay defining, this phase checks that all elements of the defined item and all elements of the base belong to the same defining class (bit or character), and that the base is contiguous.

Phase JZ

Phase JZ examines the CCCode to determine if the compiler is attempting to abort: if it is, control is passed to XA, in order that error messages may be processed by the diagnostic editor; if not, control is passed to the next logical phase.

OPTIMIZATION LOGICAL PHASE

The optimization logical phase consists of several physical phases and is loaded if OPT=2 is specified in the PLIOPT field of the PLI command.

The work done during the optimization phase can be split into two parts. The first consists of testing the text and dictionary to see if optimization is permissible. As a result of these tests, tables are built pointing to optimizable text. The second part consists of code generation and modification requiring scanning of the tables built in the first part, and direct references to the text and dictionary.

All code generation resulting in text expansion is placed in a patch file, and the point of insertion in the text is overwritten with a PTCH triple pointing to the patch. The last physical phase merges the patch text into the main program text.

Optimized code is produced for subscript address calculations and iterative DO-loop control. In the case of subscripts most of the optimized code consists of reordered triples, but optimized loop control code is generated as pseudo-code using BXLE, and BXH instructions.

Only simple loops and subscript lists are optimized, and the variables involved must be real, fixed binary, scalar integers and the constants must be decimal integers.

The two main problems in deciding whether it is permissible to optimize code are:

1. Aliasing of variables
2. The action of the program for exceptional conditions

Optimization is inhibited where it is difficult, or impossible, to decide that optimization will produce an object program which will execute according to the rules of PL/I. The keyword REORDER, indicates to the Optimization Phase, that ON-units for exceptional computational conditions may be ignored. This enables more cases to be optimized than for the default setting of ORDER.

Three types of subscript optimization are performed:

1. Transformation - Where possible, a control variable used as a subscript is transformed such that, instead of a 'subscript * multiplier + virtual origin' address calculation, each iteration produces a simple increment of a register to access the next element.
2. Invariance - Where possible, an invariant subscript calculation inside a DO-loop is moved outside.
3. Commoning - Where possible, a common subscript expression is only calculated once and this value is placed in a register to be used at later occurrences.

For array expressions an attempt is made to combine the incrementing of a transformed control variable with the BXLE or BXH of the optimized loop control code.

The text is optimized starting from the innermost of a nest of iterative DO-loops and working outwards. This enables patch code, which moves out of a DO-loop, to be included in the processing of the enclosing DO-loop, hence moving out code as far as possible in a nest of loops.

Phase KA (KB)

Contains utility routines and common data space used by the later optimization phases. Details of the utilities are given in Appendix G.

The utilities enable the optimization phases to build and process tables in text blocks without concern for physical block boundaries, status of text blocks, or maintaining pointers to first, last, and current table entries.

The facilities provided:

1. Define a table using a table control block area.
2. Add new entries to the end of a table. Table entries may be of fixed or varying length and a table can contain more than one type.
3. Scan a table forwards or backwards.
4. Make direct reference to table elements.
5. Delete a table.
6. Specify locking of entries.
7. Remove all locks on table entries.

Phase KC

Phase KC scans the text for DO-loop specifications. If the loop is potentially optimizable, then any expressions in the initial, the TO, or the BY specifications are assigned to temporary variables. The expression and the assignment are moved outside the loop and are replaced in the specification by a simple reference to the temporary variable.

Text is also scanned for ON-units. The occurrence of each type of ON-unit is recorded by the appropriate bit in the mask used by Phase KG.

Phase KE

Phase KE performs a scan of the dictionary and a scan of the text. The purpose of these scans is to mark variables 'unsafe' if they can possibly be affected by changes to other variables (i.e., aliases). Variables are marked unsafe if they are EXTERNAL, DEFINED, defined upon, BASED, or PARAMETERS, or if they are (or might be through being arguments of procedure calls) arguments of the ADDR built-in function.

In addition, during the text scan, the DO MAP table is created. This table contains an entry for each DO-loop and procedure in the source text. Each entry contains information describing the loop or procedure and giving its location in the text. A chain is constructed through these entries giving the order in which they are to be processed by subsequent K phases.

Phase KG

Phase KG scans the text corresponding to each DO MAP entry in turn and builds up two lists which are chained off the DO MAP entry. The USE list is a list of all the real, fixed binary, scalar integer variables which are used within the loop. A flag byte indicates whether the variable is assigned to or is invariant in the loop.

The SUBS/REGION list consists of two types of entry:

1. A SUBS entry which contains the text reference of a SUBSCRIPT triple referring to an array for which SUBSCRIPTRANGE is not enabled.
2. A REGION entry which contains the text reference of a triple which results in an assignment to one or more variables. There are four types of REGION boundaries:
   a. A GLOBAL region boundary which contains the text reference of a point where the value of any variable could be changed.
   b. A PARTIAL SAFE boundary which contains the text reference of a point where an assignment is made to a variable which is a SAFE real fixed binary scalar integer, followed by the dictionary reference of this variable.
   c. A PARTIAL UNSAFE region boundary which contains the text reference of a point where an assignment is made to an UNSAFE variable (not just a scalar). The dictionary reference is not inserted in this case.
   d. An ITDO region boundary which contains the text reference of an ITDO triple corresponding to an enclosed loop.

Phase KJ

Phase KJ creates the SUBS TABLE from the SUBS/REGION list produced by phase KG. The DO MAP created by KE provides the order of processing and further information.

The Region entries from the SUBS/REGION list are copied directly into SUBS TABLE whenever they occur. The SUBS entries from the list are expanded to contain information on the type of expression involved at this point. The USE list created by KG provides information during this analysis. The SUBS/REGION list is deleted by this phase.

The iterative specification triples of each DO-loop are inspected, and the spare operands used to set flags to indicate whether this loop is optimizable for BXLE or BXH loop control code.

Phase KN

Phase KN provides initialization of the scratch storage area used by phase KO.

An initial text scan is made in DO MAP sequence, to remove offsets from optimizable subscript lists and produce hash totals for optimizable subscript expressions. The hash totals are placed in the SUBS/REGION table and are used in phase KO to speed up the matching process.

Phase KO (KP,KQ)

Phase KO processes text in the order specified in the DO MAP, i.e., working through a nest of iterative DO-loops and procedures from innermost outwards.

The three types of subscript optimization: transformation of the control variable; invariance; and commoning; are performed and optimized code is generated and inserted in a patch file. The code to be replaced in the original text is overwritten with NOP's and a PTCH triple points to the patch text.

All three types of subscript optimization require searches for multiple occurrences of the same expression in the text. This is done by scanning the SUBS TABLE for matching triple expressions in optimizable subscript lists. When a match is found a chain is constructed in the SUBS TABLE between the matched elements. The code is generated for one chain at a time.

Code generated for optimized subscripts may be inserted:

1. Before the ITDO triple, i.e., where an invariant subscript calculation is moved out of a loop or where the initial setting of a transformed control variable is required.

2. Before the ITD' triple, i.e., for the incrementing code of a transformed control variable.

3. After the ITD' triple, i.e., the DROP's for symbolic registers used in the optimized code.

4. At the point of use in the subscript list.

For array expressions the incrementing code for a transformed control variable will be deleted if a BXLE or BXH can be generated which will increment the transformed control variable and control the number of iterations of the loop.

USSL declarations may be inserted in the optimized code to indicate that registers have priority and need not be saved and restored at branch points. The register allocator phase gives these registers priority over normal symbolic registers.

Phase KT

Phase KT is a renamed replacement of phase LA which is now obsolete. It is always loaded. This phase is a utility phase which remains in storage throughout the remainder of the Optimization Phase and the whole of the Pseudo-Code Phase. It provides the main scanning routines to handle input and output of text containing triples and pseudo-code.

The routine/subroutine directories in Section 3 give a complete list of the routines provided, together with brief descriptions of their functions.

Phase KU(KV)

Phase KU has three main functions performed during a single text scan.

The first function is DO-loop control optimization. Each ITDO triple encountered during the text scan is checked to determine whether or not it has been flagged as being optimizable by a previous phase. If not flagged the scan is continued. All DO-loop control specifications headed by an ITDO triple flagged as optimizable are replaced in text by an optimized pseudo-code group using the BXH and BXLE instructions. There are three basic forms to this optimized pseudo-code control specification, the particular one used for any loop depending on the type of step.

The second function is to detect each of the PTCH triples inserted into text by a previous phase. The corresponding patches are obtained from patch file text blocks and are processed as necessary before being inserted into text in place of the PTCH triple.

The last function is that of the subscript list processing. Each innermost subscript list encountered, as indicated by the presence of a SUBS triple in the main text, is checked for the occurrence of COMA or COMR triples within it. The SUBS triple is then altered as may be necessary.

PSEUDO-CODE LOGICAL PHASE

The pseudo-code phase accepts the output of the translator phase , and converts the triples into a series of machine-like instructions. The transformation into pseudo-code is achieved by a series of passes through the text; each pass removes certain triples and replaces them by pseudo-code, until the entire text is in pseudo-code form. On completion of this phase, control is handed to the storage allocation phase.

Pseudo-Code Design

Pseudo-code is essentially a one-for-one symbolic representation of machine code, designed so that it can be transformed directly into executable machine code by an assembly process.

Pseudo-code is constructed in basic units, the majority of which have a standard size of three or five bytes. A variable sized unit, however, is also available to allow flexibility, its length being specified by a length code within the unit.

The formats of pseudo-code instructions are shown in Section 4.

A unit consists of a one-byte operation code followed by normally, a two- or four-byte field, or on the other occasions by a variable length field. The bit pattern of the operation code indicates the type of unit which it heads.

Pseudo-Code Items

In addition to there being one pseudo-code item for each machine instruction which could be generated, there are also pseudo-code items which are produced to convey information from one phase of the compiler to another.

These items of information have the same format as a pseudo-code item, so that the handling and scanning of the source text is standardized. They do not, however, appear in the final object code.

Register Description

In all cases where a general purpose register appears in pseudo-code, it will be described symbolically. When conventional registers are required in, for example, calling sequences, the registers will be referred to physically, as they will be in all cases of floating-point register usage.

The Use of Symbolic Unassigned Registers

Whenever a new register is required while pseudo-code is being generated, a symbolic register counter is incremented by one and, subject to this new value not being greater than 16,383, it is used as the symbolic name of the required register. When this register is no longer required a DROP pseudo-code item is inserted into the text to indicate to the Register Allocation Phase that the physical register allocated to this symbolic register may be reassigned.

The Use of Physical Registers

Physical general purpose registers will be used either as arithmetic registers or as parameter registers.

With arithmetic registers, it is the responsibility of the pseudo-code generation phases to save and restore the registers as necessary. This will apply both to the general purpose arithmetic registers (namely 14 and 15) and to the four floating-point registers. Although this is of primary interest to the expression evaluation phases, it should be realised that all phases which generate calling sequences must be aware of the current status of arithmetic registers, and generate code to save and restore them as necessary.

In the case of parameter registers, however, the Register Allocation Phase will be able to save and restore them as required.

Temporary Descriptors

As expressions are evaluated, a series of intermediate temporary results are obtained. These results, or their addresses, may be contained in symbolic or assigned registers, in a dictionary reference, with or without an index register, or in workspace. Temporary descriptor triples (TMPD) are inserted in the text to enable the correct pseudo-code instructions to be generated from the triples. The format of TMPD triples is described in Section 4.

Temporary Workspace

A block of temporary workspace is used to store intermediate results obtained in evaluating expressions at object time. Pseudo-code phases allocate the next available workspace location within the block, and then update the location pointer, whenever the necessity to save an intermediate result arises. The location of the intermediate result is then described for later phases by a TMPD in the text. Intermediate results are only required during the execution of single PL/I statements; they are never preserved from one statement to another.

At the end of the pseudo-code phases the maximum size of the temporary storage required in each PL/I program block is placed in a dictionary entry. The required amount of workspace is then allocated in each Dynamic Storage Area (DSA) by Phase PT.

Phase LB

Phase LB scans through the text for PROCEDURE, BEGIN, and ALLOCATE statement triples.

Whenever one of these is found, a scan is made through the immediately succeeding second file statements; this is for any IDV (initial dope vector) statement referring to a variable replication factor in the array initial string. Processing of these statements and of the corresponding array initial strings is then carried out.

On completion of this secondary scan, the action taken depends on which triple was originally found:

1. For PROCEDURE or BEGIN triples, a scan is then made of the AUTOMATIC chain in the dictionary. For any scalar variables that have been declared INITIAL, a set of triples is created and inserted into the text. For any array declared INITIAL, the initial string is scanned, and a mixture of triples and pseudo-code is generated.

2. For ALLOCATE triples, if the item has been declared INITIAL, the initial string is scanned, and a mixture of triples and pseudo-code is generated.

Phase LB also marks Phase LG (DO-groups) as wanted or not wanted; this is done in cooperation with Phase IA.

Phase LD

Phase LD scans the STATIC chain for any variables which have been declared INITIAL.

When a scalar variable is found, the phase constructs two dictionary entries: one for the constant, and one for the converted constant.

For arrays, the phase scans the initial value string, creating an initialization table in the dictionary. Replication factors are converted and inserted into the table; treatment of the constants is then as described for scalar variables.

Phase OS converts the constants to their specified internal form.

Phase LG

Phase LG scans the text for DO loops. A stack is maintained with each entry containing a description of a DO group. The stacking reflects the nesting of the DO groups. For each DO or iterative DO triple a new entry is made at the top of the stack.

DO specification triples are analyzed and expressions are assigned to temporaries; subscripts in the control variable are assigned to binary integer temporaries if they are themselves variable. At the end of each specification, pseudo-code and triples are generated to control the loop.

Triple operators (see Section 4) peculiar to the specification of DO loops are removed from the text.

For control variables, other than simple scalars, text is placed in the DO stack and used at every appearance of the control variable in the generated text. During this time, a scan is also made for pseudo-variables, subscripts, functions, and argument markers.

Phase LR

The purpose of Phase LR is to save space during the expression evaluation phase, LS. It provides the initialization for Phase LS by obtaining 4,096 bytes of scratch storage and setting stack pointers. The scan phase is initialized and Phase MP is marked.

The translate table for scanning triples, and the constants for expression evaluation are included in this phase and are moved to the first 1K area of scratch storage. Subroutines required by phase LS are also moved into scratch storage at this time. Finally, control is passed to Phase LS.

Phase LS

Phase LS scans the source text to convert expression triples to pseudo-code. If a triple produces a result, it is added to the temporary work stack.

For the arithmetic triples +,-,*,/,**, prefix +, and prefix -, the operands are combined to give the base, scale, mode, and precision of the result. If conversion is necessary, an assignment triple, with the target and source types as operands, is inserted in the text. In-line pseudo-code is generated for all operators except ** and some complex type * and / operators. In these cases, library calling sequences are generated. An intermediate result is always produced and the triple is removed from the text.

The operands of comparison triples GT, GE, equals, NE, LE, and LT are combined and converted as for the arithmetic triples. In-line pseudo-code is generated and the triple is removed from the text, unless both operands are string type, in which case a temporary is created. If the next triple is a conditional branch, a mask for branch-on-false is inserted. Otherwise, the result is a length 1 bit string.

For the string triples CAT, AND, OR, NOT, and string comparisons, if an operand is zero, TMPD triples, containing the intermediate result from the top of the stack, are inserted in the text after the triple. The result is a CHARACTER or BIT string or a compare operator.

When subscript triples appear, a symbolic register number is inserted in the triple. The result contains the dictionary reference of the array and the symbolic register.

For function triples, a description of the workspace for the function result is inserted in the TMPD triples which follow the function triples. The function result is added to the intermediate stack.

For add, multiply, and divide functions, the function and argument triples are removed from the text. Arithmetic type in-line pseudo-code is generated, with modifications for the precision and scale factor, and the result is added to the intermediate stack.

With pseudo-variable triples, a special marker is added to the intermediate result stack.

Other triples which may use an intermediate result, are examined. If an operand is zero, two or three TMPD triples, containing the intermediate result from the top of the stack, are inserted in the text after the triple. If both operands are zero, the TMPDs for the second operand precede those for the first operand.

Phase LV

Phase LV provides string handling facilities for the pseudo-code phases.

It converts any type of data item to a CHARACTER or BIT string, and an assignment triple, with the target and source types used as the operands, is inserted in the text.

A string dope vector description is produced from a standard string description.

Phase LX (LW, LY)

Phase LX consists of three modules, LW, LX, and LY. Module LW acts as a pre-processor for LX and LY, moving constants into scratch storage prior to loading the string-handling modules.

Phase LX scans the source text to convert string triples to pseudo-code. If a result is produced it is added to a stack of intermediate string results.

For the comparison triples GT, GE, equals, NE, LE, AND LT, both operands are already string type. If one operand is zero, the operand is obtained from the associated TMPD triples. In-line pseudo-code is generated if the operands are aligned and are of known lengths less than or equal to 255 bytes; otherwise, library calling sequences are generated. The triple and any TMPD triples are removed from the text.

In the case of the string triples CAT, AND, OR, and NOT, the operands are converted to string type by phase LV. Zero operands are obtained from associated TMPD triples. In-line pseudo-code is generated when operands are aligned and are of known lengths less than or equal to 255 bytes. For the CAT operator, the first operand must be a multiple of 8 bits unless the strings involved are less than or equal to 32 bits in length. In-line code is also generated for the following cases involving non-adjustable varying strings:

1. Character string concatenation of varying strings with lengths less than 256 bytes.
2. Bit string operations for AND, OR, NOT, concatenation, and comparison where the strings are aligned and are less than 33 bits in length.

Otherwise, library calling sequences are generated. The triple and any TMPD triples are removed from the text, and the string result is added to the intermediate result stack.

For TMPD triples, if the intermediate result described by the TMPD triples is a string, a complete string description is moved from the top of the intermediate stack to the TMPD triples. If the TMPD triples do not describe a string, they are ignored.

In-line code is generated for the BOOL functions AND, OR, and EXCLUSIVE OR, when the third argument is a character or bit string constant and the first and second arguments are aligned and of known lengths less than or equal to 255 bytes. Otherwise library calling sequences are generated. Subscript and function triples may produce intermediate string results.

Phase MA

Phase MA generates pseudo-code for both the in-line invocations of TRANSLATE and VERIFY and for the invocations which call a library routine. It is optional depending on the presence of the TRANSLATE or VERIFY function in the source program.

Three kinds of tables are handled:

1. Compile-time created (up to three)
2. Floating, initialized by in-line code
3. Floating, initialized by library subroutine

When three constant tables have been created at compile-time, any further occurrence of this case, will cause the constants of both the second and third arguments to be handled via the library.

Blocks which have RECURSIVE, TASK, or REENTRANT attributes will have their own table, otherwise one table will be used for many blocks.

Phase MB

Phase MB scans the text for pseudo-variable markers and multiple assignment markers. A stack of pseudo-variable descriptions is maintained, together with the left hand side descriptions of multiple assignments when they occur. Pseudo-code and triples are generated for pseudo-variables and the left hand side descriptions of multiple assignments are put out in the correct sequence.

Phase MD

Phase MD uses the SCAN routine to scan the text for ADDR and STRING built-in functions for which it generates in-line code. It appears before the normal function processor phase and removes all trace of the in-line function. The general SCAN routine passes control when these functions are found.

For all cases of ADDR the generated code establishes the start address of the argument. If structure name arguments are present the structure chain is hashed for the first base-element. For array names the address of the first element is calculated.

If the argument to the STRING function is contiguous in main storage, and its length is known at compile-time, an adjustable string assignment is generated. Otherwise the library routines IHESTGA and IHESTGB are called to produce the concatenated length and to concatenate the elements of the array or structure argument.

Phase ME

Phase ME identifies all invocations of the SUBSTR function and pseudo-variable, all UNSPEC, STATUS, and COMPLETION functions, and those invocations of the INDEX function which can be implemented in-line; and generates pseudo-code to perform these functions at object time. The scan of the text is conducted by the general SCAN routine, and all trace of the invocations of these functions is removed before the normal function processor phase is loaded. When the end-of-program marker is encountered the terminating routine is entered.

Phase MG

Phase MG identifies functions which are to be coded in-line, and generates, in their place, the pseudo-code to perform the relevant function. This phase appears before the normal function processor phase and removes all trace of the in-line function.

The scan of the text is conducted by the general SCAN routine, and control is handed to the present phase when one of the following functions is found:

ALLOCATION	FLOOR	BINARY
BIT	IMAG	DECIMAL
CEIL	REAL	FIXED
CHAR	TRUNC	FLOAT
COMPLEX		PRECISION
CONJG		

Control is also passed to this phase if ABS is found with real arguments. The arguments are collected, and the appropriate routine is entered to generate the pseudo-code. When the end-of-program marker is encountered the terminating routines are entered.

Phase MI

Phase MI identifies functions which are to be coded in-line, and generates, in their place, pseudo-code to perform the relevant function. This phase appears before the normal function processor phase and removes all trace of the in-line function.

The scan of the text is conducted by the general SCAN routine and control is handed to the present phase when one of the following functions is found:

MAX	MOD
MIN	ROUND

If the number of arguments to the MAX or MIN functions is greater than three, a library call is generated.

Phase MK

Phase MK identifies functions which are to be coded in-line, and generates, in their place, pseudo-code to perform the relevant function. This phase appears before the normal function processor phase and removes all trace of the in-line function.

The scan of the text is conducted by the general SCAN routine, and control is passed to the present phase when one of the following functions is found:

DIM	HBOUND
LBOUND	SIGN
LENGTH	FREE

Phase ML

Phase ML scans the source text for generic entry name arguments to procedure invocations.

Such entry names may be floating arithmetic built-in functions or programmer-supplied procedures with the GENERIC attribute. When one is found, the correct generic family member to be passed is selected by this phase, depending on the entry description of the invoked procedure.

Phase MM

Phase MM scans through the source text for procedure invocations by a CALL statement, or for procedure or library routine invocations by a function reference.

Procedure invocations are replaced by an external standard calling sequence, and library routine invocations are replaced by an external or internal standard calling sequence as appropriate (see Section 4).

If a CALL is accompanied by a TASK, EVENT, or PRIORITY option, library module IHETSA is loaded rather than IHESA, and the parameter list is modified to include the addresses of the TASK and EVENT variables and the relative PRIORITY.

Phase MP

Phase MP reorders the BUY and SELL statements involved in obtaining Variable Data Areas (VDAs) for adjustable length strings or temporaries, which were created by Phase GK. On entering this phase, the BUY triples precede the code compiled to evaluate the length of storage required for the VDA. This evaluation code is included between further BUYS and BUY triples, which themselves are between the BUY triple being considered and its associated SELL triple. Phase MP extracts these sections of code and places them before the BUY triple of the adjustable string temporary. Since such BUY triples may be nested, the phase maintains a count to record the nesting status.

Phase MS

Phase MS scans the source text for references to subscripted array elements.

If references are found, pseudo-code is generated to calculate the offset of the subscripted element in relation to the origin of the array. If necessary, further pseudo-code is generated to check the subscript range.

Optimization of constant subscript evaluation is carried out on arrays having subscripts which are integer constants, and for which the corresponding dope vector multipliers are constant. This applies to arrays with fixed-length elements.

Phase NA

Phase NA generates pseudo-code for the following triples:

For PROCEDURE' and BEGIN' triples a Library call is generated to the FREEDSA routine.

For RETURN triples a library call is generated, unless a value is to be returned as the result of a function invocation, in which case code is first generated to assign the result to the target field, and then the library call is made. If the function may return the result as more than one data type, a switch would have been set at the entry point to the function, and the RETURN statement would test the switch value, so that the data type appropriate to the entry point is returned.

GOTO triples either will be invalid branches detected by Phase FI, in which case they will be deleted, or they will be branches to statement label constants in the same PROCEDURE or BEGIN block. In this case, they will be compiled as one-instruction branches.

GOLN triples are compiled into one-instruction branches to the compiler label number in operand 2 of the triple.

A GOOB (Go Out Of Block) triple is a branch to a label variable, possibly subscripted, or to a label in a higher block than the current one (a branch to a lower block is invalid). A call is generated to a library epilogue routine, pointing at a double-word slot containing the address of the label and the Pseudo-Register Vector (PRV) offset (for a label constant), or the invocation count (for a label variable).

STOP and EXIT statements are implemented simply by invocation of the appropriate library routine.

For IF triples, if the second operand is an identifier, or the result of an expression which is not a comparison, code is generated to convert it to a BIT string, if necessary. This BIT string is compared to zero, either in-line, or by a call to the library.

The second operand may be a mask which will have been inserted by the expression evaluation phase as a result of the comparison specified in the IF statement. This mask is put into a generated instruction to branch if the condition is not satisfied, i.e., either to the ELSE clause or to the next statement.

For ON triples, code is generated to set flag bits and update the ON-unit address in the double-word ON slot in the DSA.

For SIGNAL arithmetic condition triples, in-line code is generated to simulate the condition. For all other conditions, a library error routine is called.

REVERT triples generate code to set flag bits in the double-word ON slot in the DSA.

Phase NG

Phase NG generates the calling sequences to the library for DELAY and DISPLAY and WAIT statements.

It generates code to call the library routines which handle ALLOCATE and FREE statements whose arguments are BASED variables.

For DELAY statements, the argument has to be a fixed binary integer, and, if necessary, code is generated for conversion.

For DISPLAY statements, the message must be a CHARACTER string, or, if necessary, converted to one. A parameter list is built up to pass to the library.

For WAIT statements, the parameter list is built up in workspace. It consists of the address of the scalar expression (converted to a fixed binary integer), followed by the addresses of the event-names that appear in each WAIT statement. If the scalar expression option does not appear, the address of the total number of event-names is used.

For the tasking option WAIT, whose argument is an EVENT array, the phase makes a 4-byte entry in the parameter list, containing the number of dimensions involved, and the address of the EVENT array dope vector. If the WAIT statement contains an EVENT array and no scalar expression, the first byte of the parameter list is set to X'FF'.

For ALLOCATE and FREE statements, with based variables as arguments, a parameter list is built in workspace before a call is made to one of the entry points to IHEWLSP. The parameter list is an 8-byte RDV followed by the address of the AREA variable from the IN option if present.

For ALLOCATE, the pointer-variable in the SET option is given the value returned by IHEWLSP.

Phase NJ

Phase NJ and its supporting block, NK, generate the calling sequences to the library module for the RECORD-oriented input/output statements: DELETE, LOCATE, READ, REWRITE, UNLOCK, and WRITE.

For each of these calls, the information contained in the options of the source statement is passed by a parameter list, constructed as follows:

```
DC  A(DCLCB)
DC  A(RDV|COUNT¹|PNTR²|SDV³)|0
DC  A(EVENT|LABEL⁴)|0
DC  A(SDV.KEYTO|SDV.KEYFROM|SDV.KEY)|0
DC  A(REQUEST_CODES)
```

[1] expr in IGNORE (expr)
[2] pntr in READ SET (pntr)
[3] SDV of varying string in READ INTO (varying string)
[4] Compiler label as result of LOCATE

REQUEST_CODES is a full-word containing four control bytes with the following meanings:

Byte 0	Operation code	
	00	READ
	04	WRITE
	08	REWRITE
	0C	DELETE
	10	LOCATE
	14	UNLOCK
Byte 1	Group 1 options code	
	00	SET
	04	IGNORE
	08	INTO|FROM
Byte 2	Group 2 options code	
	04	KEYTO
	08	NOLOCK
Byte 3	Group 3 options code	
	04	VARY INTO
	08	VARY KEYTO
	0C	BOTH

Note that null arguments in the parameter list or REQUEST_CODES are indicated by zeros.

Both the parameter list and the REQUEST_CODES word are constructed in STATIC storage. However, if the argument of any of the options refers to AUTOMATIC, CONTROLLED, or BASED storage, the parameter list is moved to the workspace storage for the statement; the argument is then provided just before the library call is made.

In the case of the LOCATE statement, the phase is responsible for generating code to set the pointer variable with the pointer value returned in the first word of the RDV by the library. If the BASED variable was a structure with a REFER option in an extent definition, it is also responsible for generating code to initialize the extent variable named in the REFER option.

The DCLCB parameter is taken from the FILE option of the statement; the FILE option must be either a file constant or file parameter.

The record dope vector (RDV) is assumed to have been constructed by earlier phases, except in the case of CONTROLLED or BASED variables or CONTROLLED or BASED aggregates, when the procedure is as follows:

1. For CONTROLLED or BASED aggregates, Phase NJ creates a library call to IHESTRA, passing the following arguments through registers:

   ```
   Register 1  A(D.V)
   Register 2  A(DVD)
   Register 3  A(RESULT.RDV.SLOT)
   ```

2. For CONTROLLED or BASED strings, the phase generates code to construct the RDV in the workspace storage of the statement, using the dope vector of the string.

The IGNORE expression is taken from the IGNORE option of the statement and if necessary, converted to an integer.

The EVENT scalar is taken from the EVENT option of the statement.

The KEYTO SDV is derived from the KEYTO option of a READ statement.

The KEY SDV and KEYFROM SDV are derived from their respective options. If necessary, they are converted to character strings.

The PNTR is taken from the SET triple of the statement or from the BASED variable of the LOCATE triple if no SET triple appears.

Phase NM

Phase NM generates the calling sequences to the library modules for OPEN, CLOSE, GET, and PUT statements.

For OPEN and CLOSE statements, a parameter list is constructed from the options given. The options are first checked for validity with respect to multiple specifications. The arguments on the options are checked and converted, if necessary, to the correct data type. If no file is specified in an OPEN or CLOSE statement, it is ignored. The parameter lists are as follows:

```
OPEN  DC  A(DCLCB)

      DC  A(OCB)

      DC  A(TITLE.SDV)

      DC  A(IDENT.SDV)

      DC  A(IDENT.DED)

      DC  A(KEYLENGTH)

      DC  A(LINESIZE)

      DC  A(PAGESIZE)

CLOSE DC  A(DCLCB)

      DC  A(IDENT.SDV)

      DC  A(IDENT.DED)
```

Null arguments are indicated by zero address constants.

For GET and PUT statements, the library call is in three parts. The initialization, data transmission (Phase NU), and the termination. The initialization call requires a parameter list to be constructed from the given options. The options are checked for legal combinations and the arguments examined.

The parameter list when a file is specified is :

```
DC  A(DCLCB)

DC  A(next statement)

DC  A(binary integer) if SKIP or
        LINE is given.
```

For GET and PUT STRING, the argument to STRING is checked, and the parameter list formed is:

```
DC  A(SDV of string argument)

DC  A(DED of string argument)
```

The termination library call has no parameters. As for the initialization, the routine used depends on the options given in the statement.

Phase NT

This phase, which is a preprocessor for Phase NU, has two functions:

1. Initialization of a block of scratch storage for use by Phase NU

2. Setting up of INCLUDE matrix and library routine entries for edit-

directed, STREAM-oriented I/O statements

The phase contains all pseudo-code skeletons used by Phase NU. 4096 bytes of scratch storage are obtained and the pseudo-code skeletons are copied into it. The address of the scratch area is then passed to Phase NU.

If a flag has been passed from Phase NM, indicating the presence of edit-directed I/O, a scan of the text is performed. Data and format list items encountered during the scan are associated as far as possible, and a sufficient set of library modules are identified for the edit-directed transmission specified in the program. The INCLUDE matrix is updated and dictionary entries are made for the required library format-director routines.

Phase NU

Data/format lists in I/O statements produce an internal library calling sequence (see Section 4) for each data item and format item pair, using registers to point at the data item, the data item DED, and the FED for the format item.

Iterations of data items, as in array input or output, and of format items, are achieved by making DO loops out of the iterations.

The data items are transmitted serially, with program flow going from an item in the data list, to the corresponding format item and then to the relevant library I/O module. On return from the library module, control goes to the code for the next data item or, in the case of repeated data items, to another iteration of the DO loop.

Remote format statements are executed in a similar way. After the R format item is met, control is passed directly from the data list to the format statement until the end of the format statement. Control then returns to the item in the in-line format code of the EDIT statement following the appropriate remote format item. However, if no format elements remain but some data list elements are still present, control is passed back to the beginning of the format statement.

An R format item referring to a label which is not attached to a format statement will cause an object time error condition to be raised, and the execution to terminate.

Phase OB

Phase OB scans through the text for compiler functions and compiler pseudo-variables (see Section 4). When a compiler function is found, pseudo-code is generated to access the operands of the compiler functions (e.g., string length, array bound), and to place the operand in the location specified by the TMPD following the function. Assignments to compiler pseudo-variables are treated in reverse; the result from the TMPD following the assignment is stored in the array bound or string dope vector slot specified in the compiler pseudo-variable.

Phase OB also scans the text for BUY, SELL, and BUY ASSIGN statements. The temporary operands of these statements are examined, and if they are CAD or short fixed-length strings, they are allocated the next available workspace offset, and the BUY and corresponding SELL statements are removed from the text.

Phase OD

This phase contains the translate and test table used by SCAN, and other tables and constants for phase OE. A block of scratch storage is obtained into which the tables, routines, and constants are moved. A pointer to the beginning of this area is passed to OE in a register.

Phase OE

Phase OE translates the following triples into pseudo-code:

- Assignment
- Multiple source assignment
- Multiple target assignment
- ALLOCATE, FREE, BUY, and SELL
- Special assignment

In-line code is generated for the following types of ASSIGNMENT triples:

1. Floating-point to floating-point
2. Fixed binary to fixed binary
3. Fixed decimal to fixed decimal
4. Numeric field to numeric field, if the pictures given for the operands are identical
5. CHARACTER string to CHARACTER string, if the operands are fixed length and not more than 256 characters
6. BIT string to BIT string, if the operands are aligned and not more than 2040 bits

7. Label to label

8. File constant to file parameter

9. POINTER/OFFSET to POINTER/OFFSET

10. FIXED CHARACTER string to VARYING CHARACTER string and VARYING CHARACTER string to VARYING CHARACTER string provided that:

    - The length of the source operand is not greater than 256 bytes

    - The length of the target string is not greater than 256 bytes, if the maximum length of the source string is not known.

    - For FIXED CHARACTER string to VARYING CHARACTER string the length of the FIXED string is not greater than 256 bytes.

Library calling sequences are compiled for those cases of CHARACTER string to CHARACTER string and BIT string to BIT string codes not compiled in-line.

After checking both AREA operands, AREA assignments are performed by the library.

All other assignment triples are translated into the CONV pseudo-code macro.

If the source operand is a constant, the type of the target operand is inserted in the constant dictionary entry, for processing by the constant conversion phase, and the assignment is translated assuming the target type.

MULTIPLE ASSIGNMENT triples produce the same code as for single assignment, except that the registers used by the operand concerned must not be changed or dropped.

Library calling sequences are generated for ALLOCATE, FREE, BUY, and SELL triples, and pseudo-code markers are left in the text for insertion of code by Phase QF.

With SPECIAL ASSIGNMENT triples, if the target is a varying or adjustable string, storage is obtained if the target is AUTOMATIC, or allocated if the target is CONTROLLED. The assignment is then translated.

Phase OG (OL)

Phase OG converts to pseudo-code all statement numbers, statement labels, PROCEDURE, BEGIN, PROCEDURE', BEGIN', and end-of-program triples.

The CONVERT pseudo-code macro is examined in conjunction with the OPTIMIZATION parameter and pseudo-code is generated in one of three forms:

1. Code to call the library conversion package

2. Code to perform the conversion "in-line"

3. A modified CONV macro which is passed to phase OM or OP for processing. In-line conversion phases which are not required (OM and/or OP) are marked unwanted.

IGN pseudo-code items and JMP triples are removed. The amount of temporary working space required by each block of program is calculated and placed in the workspace dictionary entry (see Section 4).

The format of the text is converted so that a pseudo-code item does not span blocks.

The INCLUDE card matrix is formed for all the conversion modules required.

Phase OM

Phase OM is called when either optimization levels 00 or 01 are specified. This phase scans the pseudo-code for the CNVC macros, which phase OG has placed into the text as 28-byte entries containing a transfer vector to select the appropriate conversion routine within OM, and replaces any such macros with in-line code.

The conversions inserted by phase OM are controlled by phase OG. When OPT=0, certain of the simpler FIXED DEC to PICTURE, PICTURE to FIXED DEC, and FIXED DEC to FIXED BIN conversions are passed to OM. When OPT=1, the remainder of the feasible FIXED DEC to or from PICTURE and FIXED DEC to FIXED BIN conversions are passed to OM together with FIXED DEC to CHAR conversions.

Certain FIXED DEC to PICTURE conversions, which phase OG cannot itself efficiently detect to be uneconomic when performed in-line, are recognized by phase OM, which inserts the calls to the appropriate library routines.

Phase OP

Phase OP generates in-line code to perform BINARY to BIT string, BIT string to BINARY, and FLOAT to FIXED BINARY conversions.

Phase OS

Phase OS scans through the constant chain in the dictionary and converts the constants to the required internal form.

These are then stored in a constants pool, and the offset of each constant from the start of the pool is saved in the dictionary entry for that constant.

To permit the correct alignment of the constant pool, three scans are made of the constant chain; first to convert all double word constants, secondly to convert all single word constants, and thirdly to convert all unaligned constants.

In the first two scans only one pool entry is made for constants having the same internal form and value.

A fourth scan is made of the constant chain and all constants required to initialize static are converted, but instead of inserting these constants in the constant pool, they are moved into special dictionary entries constructed by Phase LB.

STORAGE ALLOCATION LOGICAL PHASE

The storage allocation phase ensures that every item requiring storage in a PL/I object program obtains a unique location of the correct size, located on the correct boundary. Items requiring storage include PL/I source program variables, dope vectors, dope vector skeletons, temporary variables, work areas, data descriptors, symbol tables, addressing slots, register save areas, flag areas, etc. Storage locations are allocated to items in order of descending alignment requirement to avoid wasting storage by padding to the required alignment.

The storage allocation phase is also responsible for generating prologues. In generating the prologues, expressions which determine size of variables, code generated by the aggregates phase to initialize dope vectors, and code generated by the initial values phase, must be extracted and placed in the correct sequence in the text. Also, when a variable depends for its size or initial value upon another variable, the requests for dynamic storage must be arranged so that the dependant variable obtains its storage after the variables upon which it depends.

Since all AUTOMATIC and CONTROLLED storage is obtained dynamically at object time, the Storage Allocation Phase generates code to relocate dope vectors when the allocated storage address is known.

Phase PA

The purpose of phase PA is to determine the eligibility of the automatic chains of any block for STATIC DSAs. Any chain not so far found to be ineligible for a STATIC DSA is scanned to determine the DSA size. STATIC DSAs are generated for any chains of less than 512 bytes.

Dictionary entries are generated for STATIC DSAs. This phase also acts as a spill area for routines used in phases PD and PH.

Phase PD

Phase PD is the first STATIC storage allocation phase. It scans the text, and for every second file statement encountered sets up a pointer in the associated dictionary which points to the second file statement. It then sorts the STATIC chain so that the dictionary entries occur in the order in which the storage for their items will be allocated.

Storage is allocated for simple non-structured, non-external variables, RDVs, DEDs, SAVE/RESTORE entries, and the BCD of entry labels and label constants. Storage is also allocated for dope vectors for all items in the STATIC chain requiring them, with the exception of EXTERNAL items. A full word address slot is allocated in STATIC for each STATIC DSA.

The external section of the sorted STATIC chain is scanned and a 4-byte addressing slot is allocated for each entry label, label constant, external (entry type 4) entry, built-in function, or EXTERNAL item. For each EXTERNAL item the size of the external control section is calculated and stored in the dictionary entry.

The constants chain is scanned and the offsets of the storage and dope vectors for constants in the constants pool are relocated.

The current size of the STATIC INTERNAL control section is computed and the result is passed via the communications region to the next phase.

Phase PH

Phase PH is the second STATIC storage allocation phase. It scans the AUTOMATIC chain and CONTROLLED chain for all items requiring a dope vector.

For each such item a skeleton dope vector dictionary entry is generated in the STATIC chain (see Section 4). This dictionary entry contains a bit pattern equal in length to that of the dope vector and containing all those values which are known at compilation time. In particular, it contains as much of the relative virtual origin as is known at compilation time, the constant bounds and string lengths, and the constant multipliers.

Skeleton dope vectors are not put into the STATIC chain for AUTOMATIC variables in any block whose DSA is in STATIC, except when the variable dimensions bit is set to one.

If the item is dynamically DEFINED, then the dope vector is preceded by one extra four-byte slot. (In the case of structures there is one extra slot for each element of the structure.) If the item is a dynamic temporary (temporary type 2) or a CONTROLLED scalar string, the virtual origin slot is relocated by the length of the dope vector.

In all cases the skeleton dope vector dictionary entry is pointed at by the dictionary entry of the associated item.

The sorted STATIC chain is scanned from the first skeleton argument list entry. For each such entry, space is allocated in the STATIC INTERNAL control section according to the assembled length of the argument list. The offset of each skeleton argument list is stored in the OFFSET1 slot of the dictionary entry.

RDV and DVD entries are found on this same scan of the STATIC chain. RDV entries are allocated eight bytes; DVD entries are allocated the specified length.

A scan is made of the section of the STATIC chain containing STATIC INTERNAL arrays. Storage is allocated for each array according to its size (computed by Phase JK) and the offset of the relative virtual origin is relocated to the start of the STATIC INTERNAL control section. If the array is of the VARYING type and it needs a dope vector, then storage is allocated for the secondary dope vector. The number of elements is calculated for INITIAL arrays and stored in the associated INITIAL dictionary entry.

The section of the STATIC chain containing STATIC INTERNAL structures is scanned. Storage is allocated for each structure according to the size of the structure (computed by Phase JK), and this storage is placed on the correct boundary on information supplied by Phase JK. The structure member chain for each structure is scanned and the relative offset of each member is relocated to the start of the STATIC INTERNAL control section. Further, on the structure member scan, secondary dope vectors are allocated when required, and the number of elements is calculated for INITIAL arrays.

Phase PL

Phase PL scans the STATIC, AUTOMATIC, CONTROLLED, structure, and PROCEDURE block chains for variables which require storage for their symbol tables and/or data element descriptors.

When a variable is found which requires a symbol table, the variable is joined onto the chain of symbol variables for the particular block. A symbol table dictionary entry is created for the variable (see Section 4), and a chain is set up to and from the dictionary entry for the variable. The new dictionary entry is joined onto the STATIC chain.

The size of the symbol table is calculated, and its offset from the start of the STATIC control section is stored in the symbol table dictionary entry. Throughout the allocation of STATIC storage a location counter is maintained to contain the next free location in STATIC; this counter is increased appropriately.

All symbol variables require a DED and a branch is taken to the routine which allocates them.

When a variable is found which requires a DED, it is determined whether or not the DED describes a standard type; there are eight standard types, which consist of the different kinds of real coded arithmetic data that can be obtained by the combination of the attributes FIXED/FLOAT, BINARY/DECIMAL, LONG/SHORT (default precisions only).

If the DED is of a standard type, a check is made for an identical DED that may have already been encountered, so that there will be only one allocation of storage for any one type of standard DED. If the DED is not of a standard type, it is allocated storage of its own.

If the variable does not already have a symbol table dictionary entry (which contains space for DED information), a DED dictionary entry is constructed, and the offset of the DED in the STATIC control section is stored in it. A pointer in the new entry in the dictionary entry for the variable is also set up.

When all data element descriptors and symbol tables in the compilation have been processed, all STATIC storage has been allocated and the total size of the STATIC control section is placed in a slot in the communications region.

Phase PP (PO)

Phase PP extracts all ON condition entries and places them at the head of the AUTOMATIC chain. It then extracts all temporary variable dictionary entries from the AUTOMATIC chain and places them in the zone following the ON conditions in the chain.

All dictionary entries which are totally independent of any other variable are extracted, and also placed in the zone following the ON conditions.

The phase then extracts all dictionary entries which depend upon some other variable in containing blocks or in the zones already extracted, and places them in the next following zone. Dependency includes expressions for string lengths, expressions for array bounds, expressions for INITIAL iteration factors, and defined dependencies. This is repeated recursively until the end of the chain. If some variable depends upon itself, a warning message is issued.

A special zone delimiter dictionary entry is inserted between each zone in the AUTOMATIC chain (see Section 4). A codebyte is initialized in the delimiter to indicate to Phases PT and QF whether its following zone contains any variables which require storage (i.e., it does not consist entirely of DEFINED items, which do not require storage), and whether or not the following zone contains any arrays of VARYING strings.

Phase PT

Phase PT allocates AUTOMATIC storage, scans the CONTROLLED chain, and determines the size of the largest dope vector. It scans the entry type 1 chain, and for each PROCEDURE block or BEGIN block it allocates storage for a DSA and compiles code to initialize the DSA.

A two-word slot in the DSA is allocated for each ON condition in the block, and code is compiled to initialize the slot. Space for the addressing vector and workspace in the DSA is also allocated.

Two words are allowed for tasking information in the DSA if the TASK option is on the external PROCEDURE of the compilation.

The AUTOMATIC chain is scanned and dope vectors are allocated for the items requiring them. Code is compiled to copy the skeleton dope vector, and to relocate the address in the dope vector.

Where there is a block with its DSA in STATIC, dope vector initialization is not performed for the variables in the first region of the AUTOMATIC chain. Address slots in dope vectors for variables in the remainder of the chain are relocated.

Storage is allocated for addressing temporaries type 2 and for addressing controlled variables, and for the parameters chained to the entry type 1.

The first region of the AUTOMATIC chain is scanned and storage allocated for double precision variables, single precision variables, halfword binary variables, CHARACTER strings, and BIT strings, in that order.

The first region of the AUTOMATIC chain is scanned and storage allocated for arrays, relocating the virtual origin. For arrays of strings with the VARYING attribute, the secondary dope vector is also allocated and code is compiled to initialize the secondary dope vector. Correctly aligned storage is allocated for structures. If a structure contains any arrays of strings with the VARYING attribute, the storage for the secondary dope vector is allocated at the end of the structure.

A pointer is set up in the AUTOMATIC chain delimiter to the second file statement which has been created.

The remaining regions of the AUTOMATIC chain are scanned and code is compiled to obtain a Variable Data Area (VDA) for each region. Code is compiled to copy the skeletons into the dope vectors and to relocate the addresses in the dope vectors. During this pass, any DEFINED items which are to be addressed directly have the storage offset and the storage class copied from the data item specified as the base identifier.

Phase QF

Phase QF, which constructs prologues, scans that text which is in pseudo-code form at this time with end-of-text block markers inserted.

When a statement label pseudo-code item is found, it is analyzed and one of three things happens:

1. The item is saved if it relates to a PROCEDURE statement
2. The item is omitted if it relates to a BEGIN or ON block
3. The item is passed if it relates to neither of the first two conditions.

When a BEGIN statement is found, a standard prologue of simple form is generated, and code is inserted from second file statements (if there are any) to get the DSA, either dynamically, or in the case of eligible bottom-level blocks, by using the supplementary LWS made available at initialization time. Code is also inserted to initialize the DSA and to allocate and initialize any VDAs.

When a PROCEDURE statement is found, it is first determined whether it heads an ON block or a PROCEDURE block. If it is an ON block, a standard prologue (similar to that for a BEGIN block) is generated. If it is a PROCEDURE block, a specialized prologue is generated. This takes account of the manner of getting the DSA, the number of entry points, the number of entry labels on a given entry point, the number of parameters on each entry point, and whether the PROCEDURE is a function.

Prologue code is generated for AUTOMATIC scalar TASK, EVENT or AREA variables, in order to perform the initialization required when these variables are allocated.

The code generated by the prologue construction phase is partly in pseudo-code and partly in machine code. The machine code (which is delimited by special pseudocode items) has the same form as the code produced by the Register Allocation Phase (see Section 4).

DSA optimization is performed under certain conditions (see Appendix D).

At the end of the prologue, the statement label item saved earlier is inserted to mark the apparent entry point. Code is produced to effect linkage to BEGIN blocks in such a way that general register 15 contains the address of the entry point, and general register 14 contains the address of the byte beyond the BEGIN epilogue.

At the end of the text, any text blocks that are not needed are freed, and control is passed to the next phase.

Phase QJ

Phase QJ scans the text for ALLOCATE, FREE, and BUY statements.

On finding an ALLOCATE statement, a routine is called which does a 'look ahead' for initialization statements associated with the allocated variable, e.g., adjustable array bounds or adjustable string lengths, and places the text references of each statement in the dictionary entry associated with each statement.

If the allocated item has a dope vector, code is generated to move the skeleton dope vector generated by Phase PH into a block of workspace in the DSA of the current block.

Any adjustable bound expressions or string length expressions are then extracted from the text references, and the expressions are placed in-line in the text.

Any information required from previous allocations (specified by * in the ALLOCATE statement) is extracted from the previous allocation, and copied into the workspace.

Code generated by Phase JK to initialize multipliers, etc., is extracted and placed in-line, after first loading the variable storage accumulator with the dope vector size. Phase JK generates code to increment the accumulator register by the size of the item.

If the item has no adjustable parameters, code is generated to increment the accumulator by the size calculated at compilation time. If this size is greater than 4,096, Phase JK generates a constant dictionary entry, which is used in this code.

If the item has any arrays of varying strings, the size of the array string dope vector is added to a second accumulator register. Code is generated to add the two accumulators into the second one, which is a parameter to a library routine. A routine is then called which extracts the library call inserted by pseudo-code and places it in-line in the text.

Code is inserted after the library call to initialize the dope vector in workspace to point to the allocated storage. Code is generated to transfer the dope vector from the workspace to the allocated storage.

The code generated by phase JK to initialize arrays of varying strings, tasks, events, and areas is then inserted in the output stream.

Any initial value statements associated with the ALLOCATE statement are extracted and placed in-line. The initialization statements are then skipped, and the scan continues. The last two steps are also performed for LOCATE (based variable) and ALLOCATE (based variable) statements. Action for a BUY statement is similar to an ALLOCATE statement, with the following exceptions:

1. Bound and string length code is in-line, bracketed between BUYS and BUY statements - there is therefore no look ahead
2. There is no initial value code associated with temporaries

3. A slot in the DSA is updated with the pointer to the allocated storage for a temporary.

The action on encountering a FREE statement is to generate code to load a parameter register with the pointer to the allocated storage for the FREE VDA Library call inserted by the pseudo-code.

Phase QU

Phase QU scans the pseudo-code text in search of instructions which have misaligned operands. (A misaligned operand has the UNALIGNED attribute and is not aligned on the boundary appropriate to its data type). When such an instruction is found, QU inserts a move character (MVC) instruction in the pseudo-code text to move the operand to or from an aligned workspace area, and substitutes the address of this workspace for the operand address in the original instruction. If the address of a misaligned operand is loaded into a register, a note is made of that register. QU thereafter treats the instructions which refer to it as if they referred to the operand itself, by inserting a move character instruction, and substituting the workspace address for the reference in the instruction.

In handling misaligned operands, phase QU uses storage beginning at offset 32 from register 9 for its workspace.

Whenever a load address (LA) instruction is found which lies within the calling sequence of a library routine and which loads the address of a misaligned argument of that routine, an aligned workspace address is substituted in the instruction, and the requisite move character instruction is stacked. It is not inserted in the output text until the instruction is encountered that loads register 15 prior to the exit to the library routine, or in the case of EDIT-directed I/O routines, until the appropriate branch-and-link (BALR) instruction is encountered. The stacked move character instruction is inserted into the output before the exit to the routine if the argument in question is an input argument to the routine, and after the return from the routine if it is an output argument.

Whenever a fixed binary temporary of precision < 16 is encountered in the text, the dictionary is checked to see if this is a member of an argument list (phase GP will have set bit). If it is, the instructions referring to it are altered to halfword. The displacement in any Load Address referring to the temporary is incremented by 2.

References to halfword binary items are replaced by halfword instructions where PL/I permits. Where possible and desirable, fullword instructions are used to perform calculations, and only LH/STH instructions used to access storage.

Fullword conversion is inserted into the library calls marked by phases LS and NG.

In handling halfword binary items, phase QU uses 4 bytes, beginning at offset 0 from register 9, for workspace.

Phase QX

Phase QX is the 'AGGREGATE LENGTH TABLE' printing phase. It is entered only if the ATR (attribute list) or XREF (cross reference list) options are specified. It scans the STATIC, AUTOMATIC, CONTROLLED and COBOL chains, and, for each major structure or non-structured array that is found, an entry is printed in the AGGREGATE length table.

An AGGREGATE LENGTH TABLE entry consists of the source program DECLARE statement number, the identifier and the length (in bytes) of the aggregate. In the case of a CONTROLLED non-BASED aggregate no entry is printed for the DECLARE statement, but an entry is printed for each ALLOCATE for the aggregate, the source program ALLOCATE statement number being printed in the 'statement number' column.

Where the length of an aggregate is not known at compilation the word "ADJUSTABLE" is printed in the 'length in bytes' column of the entry for that aggregate. If an aggregate is dynamically defined, the word "DEFINED" appears in that column. An entry for a COBOL mapped structure (i.e., a structure which a COBOL record is read into or written from), has the word "(COBOL)" appended, but such an entry will appear only if the structure does not consist entirely of one of the following:

- doubleword data
- fullword data
- halfword binary data
- character string data
- aligned bit string data
- a mixture of character string and aligned bit string data

If a COBOL entry does appear, it is additional to the entry for the PL/I mapped version of the structure.

Before printing begins the aggregate length table entries are sorted so that the identifiers appear in collating sequence order.

REGISTER ALLOCATION LOGICAL PHASE

The register allocation phase inserts into the text the appropriate addressing mechanisms for all types of storage, and to allocate physical general registers where symbolic registers are specified or required as base registers.

This phase comprises two physical phases, each with a specific function. The first, Phase RA, processes the addressing mechanisms, while the second phase, Phase RF, allocates the physical registers.

An additional phase RD is called in between RA and RF when the optimization option is 2 or greater. This phase attempts to optimize the storing and loading of registers in use over compiler generated branches.

Phase RA (RB,RC)

Phase RA scans the text for dictionary references, the beginnings and ends of PROCEDURE and BEGIN blocks, and the starting points of the original PL/I statements.

A dictionary reference, when found, is decoded into a word-aligned dictionary address and a code. These are used to determine what is being referenced. The corresponding object time address as an offset and base is then calculated.

If the address required has an offset less than 4,096 and a base which is either an AUTOMATIC or STATIC data pointer, no extra instructions are generated. If this is not so, extra instructions are inserted in the text stream to calculate the required address. The calculation of this address is broken down into logical steps in a 'step table.' On completion, the table is scanned backwards to determine whether an intermediate result has been previously calculated. The steps which have not been previously calculated are then assembled into the pseudo-code.

The compiled code is added either to the output stream or to a separate file. The code in the separate file is terminated by a store instruction to save the calculated address. The extra "insertion file" is placed in the prologue of the relevant block by the next phase. Instructions are stored in-line if the referenced item is CONTROLLED, if it is a parameter, if fewer instructions are required to recalculate the base rather than load the stored address, or if the reference itself is in the prologue.

If no addressing code is generated, a special item is put in text to tell phase RF what base to use.

All relevant information for PROCEDURE and BEGIN blocks is stacked and unstacked at the start and end of the blocks respectively.

At the start of PL/I statements, code is compiled to keep the required PREFIX ON slots in the Dynamic Storage Area updated. On meeting the pseudo-code error marker, the calling sequence to the library error package is generated, and the error marker removed.

If the STMT option has been specified, code is generated at the start of each PL/I statement to keep the statement number slot in the current DSA up to date.

Phase RD

Phase RD examines all EQUs and determines their uses. A table is set up in scratch text blocks containing a four-byte slot for each EQU. The number of text blocks required is calculated from the value in the ZMAXEQ field in the communications region. The first text block, containing the slots for the first N-4 EQU values (where N = text block size), is locked into main storage so that these slots can be accessed by direct addressing.

The other slots are accessed via their text references, and their text blocks are brought into storage as needed, by the compiler control routines. A dictionary of text block numbers for each range of EQU values is kept in the phase. This allows for a maximum of 64 text blocks, i.e., under the smallest SIZE parameter a maximum of 16K EQU values are allowed.

The table is built up during a pass of the program text. At the end of the text pass the table is scanned. Any EQU which is not used is deleted. Any EQU which is either before the first use or used more than once is flagged by setting the first bit of the EQU value on. During this scan of the table, the current table text block is locked into storage and released when the scan is completed for the block.

Phase RF (RG,RH)

Phase RF scans the text for register occurrences, implicit and explicit, and the start and end of PROCEDURE and BEGIN blocks. At the beginning of PROCEDURE and BEGIN blocks all relevant information is stacked, and is later unstacked at the corresponding end.

Registers are classified as assigned, symbolic, or base.

Assigned registers require the explicitly mentioned register to be used. If that register is not free it is stored. Symbolic registers may occupy any register in the range 1 through 8. An even-odd pair may be requested. Base registers may occupy any of registers 1 through 8.

When a register is requested, a table of the contents of registers is scanned, to determine whether the register already has the required value. If it does, that is used. If it does not, and it is not an assigned register, a search is made for a free register and this is allocated if one is found. Should no register be free, a look-ahead is performed to determine which register it is most profitable to free.

If a register contains a base it need not be stored on freeing. If a register contains a symbolic or assigned register, it may require to be stored when freed, depending upon whether it has had its value altered since any storage associated with it was last referenced.

At a BALR (Branch and Link) instruction it is ensured that all the necessary parameter registers are in physical registers, and not in storage.

No flow trace is carried out by the compiler. Therefore, the register status is made zero at branch-in and branch-out points. An exception is at a conditional branch. Here the registers are not freed after having been saved.

Any coded addressing instructions are expanded when found in-line. At a specific "insertion point" in a prologue, any addressing instructions in the "insertion file" are brought in and expanded.

FINAL ASSEMBLY LOGICAL PHASE

The final assembly phase converts the pseudo-code output of the register allocation phase into machine code, the principal functions being the substitution of machine operation codes for pseudo-code operations, and the replacement of PL/I and compiler inserted symbolic labels by offset values.

Loader text is generated for program instructions, DECLARE control blocks, and OPEN file control blocks, initial values defined in the source program, parameter lists, skeleton dope vectors, symbol tables, etc. ESD and RLD cards are generated for external names and pseudo-registers. An object listing of the code generated by the compiler is produced if the option has been specified by the source programmer.

Phase TF

Phase TF scans the text, assigns offsets to compiler and statement labels, and determines the code required for instructions which reference labels.

The size of each procedure is determined and stored in the PROCEDURE entry type 1. A location counter of machine instructions is also maintained.

Phase TJ

Phase TJ scans the text until no further optimization can be achieved in the final assembly.

A location counter is maintained for assembled code, and offsets are assigned to labels.

The size of each procedure is determined and stored in the PROCEDURE entry type 1. The amount of code required for instructions to reference labels is also determined, while attempting to reduce this from the amount estimated by the first assembly pass.

This phase also attempts to reduce the number of Move (MVC) instructions by searching for consecutive MVC instructions which refer to contiguous locations.

Phase TO (TQ)

Phase TO sets the four byte slot ZPRNAM, in the communication region, to contain the first four characters of the first entry label of the external procedure, for the purpose of object deck serialization.

Phase TO also produces ESD cards for the compiled program. It first makes up six standard entries for:

1. Program Control Section (CSECT) (SD type) allowing room for the compiler subroutines if these are present
2. STATIC internal CSECT (SD type)
3. Invocation count (PR type)
4. Entry points to library routines, IHESADA and IHESADB (ER type)
5. IHEQERR (PR)
6. IHEQTIC (PR).

If the external procedure has the MAIN option, an entry for a one-word CSECT (SD type) is made up. An entry is made for the CSECT 1H entry and entries are made up for all entry labels in the external procedure (LD type).

The entry type 1 chain is scanned and an entry (PR type) is made up for each block and procedure.

The external section of the STATIC chain is scanned and entries are made up for:

1. Built-in functions and library functions (ER type)

2. Files (ER type)

3. STATIC external variables (SD type)

4. External entry names (ER type)

5. Programmer ON condition names (SD type).

The CONTROLLED chain is scanned and an entry is made up for each CONTROLLED variable and task name (PR type).

The size of the program control section is incremented to include the compiler subroutines.

All STATIC DSAs are put into the STATIC INTERNAL control section, their combined sizes being allowed for when the size of the CSECT is calculated.

Module TQ is used to produce a list of library conversion routines required for execution of the program. ER type entries are made up for each name in the list.

Phase TT

Phase TT scans the text and maintains a location counter for assembled code.

Loader text (TXT) and relocation directory (RLD) cards for requested combinations of load and punch files are generated.

Nested procedures are unnested at object time by suitable manipulation of the location counter. The offset of each procedure from the start of text is left in the PROCEDURE entry type 1.

Compiler labels are numbered for use by the object listing phase, and trace information is set up at entry points. Phase TT also generates the text for the compiler subroutines. These subroutines are put out in one of the following combinations:

1. EPILOGUE subroutine
   DYNAMIC PROLOGUE subroutine
   STATIC PROLOGUE subroutine

2. EPILOGUE subroutine
   DYNAMIC PROLOGUE subroutine

3. EPILOGUE subroutine
   STATIC PROLOGUE subroutine

Phase UA

Phase UA generates text for the static internal CSECT; initializes a CSECT for each static external variable; and, optionally (if the LIST option is present), lists all the text produced for the static internal CSECT and provides suitable comments.

The phase first scans to the start of the external section of the STATIC chain, generating text for entry labels, label constants, compiler labels, file attributes, label variable BCDs, and DEDs for temporaries. Simple variables found on this scan are used, together with the labels, to mark the start of the character string section of the chain.

The phase then scans to the end of the external section of the chain, initializing address constants for external variables, external entry names, built-in and library functions, programmer-defined ON-condition names, external files, and label constants. Text is made up for the constants pool.

The third scan of the STATIC chain starts at the point left by the previous scan, and generates text for dope vector skeletons, argument lists, RDVs and DVDs, and symbol tables. The scan is terminated at the end of the chain.

Phase UA makes up RLD cards for the address slots for STATIC DSA's and for the address slot of the start of the epilogue subroutine, if generated.

Text cards are output to initialize all AREA's, EVENT's, and TASK's. Arrays of AREA's, will have a text card for each element.

Phase UD

Phase UD generates RLD and TXT cards to set up dope vectors at link-edit and load time.

TXT cards are generated for each STATIC DSA, containing its length, which is found in the STATIC DSA entry.

TXT and RLD cards are generated to set up the dope vectors for structured items and any non-structured items appearing in the AUTOMATIC chains. The TXT cards are derived from the skeleton dope vector entries. The RLD cards are generated for each virtual origin slot.

When the last STATIC DSA has been processed control is released from phase UD.

Phase UE

Phase UE initializes those items on the STATIC chain not processed by Phase UA.

The phase first scans to the start of the external section of the chain, making up text for simple data, and listing label variables.

The second scan starts at the head of the character string section of the chain, and initializes dope vectors for all static internal variables which need them.

The third scan corresponds in extent to the third scan in Phase UA, but generates text for arrays, and simple and interleaved structures. At the end of this scan, a test is made to determine whether the external procedure of the program has the MAIN option. If so, a one-word CSECT (IHEMAIN) is made up, to contain the address of the principal entry point to the compilation.

The phase then executes its final scan, which extends over the external section of the chain, to initialize a CSECT for each external variable or external file.

Finally, any incomplete text and RLD cards are punched out, and an END card is produced for the compiled program. If the OBJNM parameter is present for batch compilation, phase UD punches a NAME card to follow the END card.

Phase UF (UH)

Phase UF scans the text, and lists, in assembly language format, machine instructions compiled for the source program. It inserts comments in the listing for statement numbers, statement labels, entry points, prologues, and procedure bases.

Phase UF contains module UH which generates NAME from a dictionary reference. UF also lists the text for the compiler subroutine. This is done by releasing UH and loading module UI which performs this function. Upon termination of this phase module UI passes control to phase XA.

ERROR EDITOR LOGICAL PHASE

The error editor phase is entered at the end of all compilations. The first phase, phase XA, examines the dictionary and determines whether there are any messages to be printed out. If there are none, this phase terminates the compilation. If there are diagnostic messages to be printed out, phase XB is entered. Phase XC is then entered and this, together with phase XA, causes additional modules (XF, and blocks XG to YY) to be entered. These modules process the error dictionary entries and print out the appropriate messages.

Phase XA

Phase XA examines the heads of the error chains in the first dictionary block, and the programmer options which specify the severity level of messages required. If there are no diagnostic messages to be printed, this phase prints out a completion message and completes the compilation. If diagnostic messages are required, phase XC and the message address block XF are called.

The error editor then scans down the error message chains and marks each error dictionary entry with an indication of where the associated message is to be found. This information is obtained from a table in module XF.

The text of all error messages is kept in modules XG through YY. The messages are ordered, by severity, within these modules. Module XA will have listed those modules which contain messages required for a particular compilation. Module XC loads and releases these modules, one at a time, and extracts the required messages. Having loaded a particular module, the phase scans down the associated error message chain in the dictionary for error entries associated with the module. It accesses the error message text and scans it.

The message to be printed is built up in a print buffer in internal compiler code. This involves a translation from EBCDIC mode, which is used for the message text skeleton. The message is completed by the insertion of a statement number, an identifier, or a numeric value as specified by the message dictionary entry. The message is segmented, where necessary, to avoid spilling over a print line, translated to external code, and finally printed out.

When all error message dictionary entries have been processed, module XB returns control to phase XA, which passes control to module AA for termination of the compilation.

Note: This routine for the handling of diagnostic messages is completely separate from, and should not be confused with, module XZ, which is responsible for producing conversational diagnostic messages at the user's terminal.

This section provides a complete guide to the compiler logic, in the form of flowcharts and associated tables and routine directories, arranged in phase order.

Flowcharts

The compiler flowcharts are presented at three levels of detail -- overall, logical phase, and physical phase. The overall compiler flowchart (Chart 00) points to the logical phase flowcharts (Charts 01 through 12), each of which appears at tne head of the set of physical phase flowcharts to which it points. The physical phase flowcharts point (by means of identifiers placed next to the blocks) to the various routines used. Entry points to physical phases are labeled.

The compiler control modules are referenced frequently throughout compilation. The control module flowchart (AA) indicates, to the right of each block, the control module being referenced to perform the function described.

Flowchart conventions and USASI symbols are described immediately preceding the flowcharts.

Tables and Routine Directories

For each physical phase, a table is provided which lists the operations performed, identifies the routines and subroutines contained in the phase, and states their function.

In some cases, a physical phase comprises more than one module; this means that routines contained in different modules may be listed together in one routine directory. To provide a cross-reference to the compiler listings, the following convention has been adopted: If a routine is contained in a module whose label is not identical to that of the phase under discussion, the label of the containing module is inserted in parentheses after the routine name in the directory.

In the case of a phase sharing a routine contained in another phase, the label of the containing module is indicated in parentheses after the routine name in the "Subroutines Used" column. The routine will not then appear in the routine directory for the phase under discussion, but will be found in the routine directory for the containing phase.

Chart and Table Identification

Identification of tables and physical phase flowcharts is based on the phase label. Individual modules within the compiler are named IEMTXX, where XX stands for two alphabetic characters. All references to these modules, in the flowcharts and throughout this manual, have been limited to the last two characters.

CONTROL PHASE TABLES

Table AA. Module AA Compiler Resident Control Phase (Part 1 of 2)

Statement or Operation Type	Main Processing Routine	Routine Called
Initializes the compiler Parameters passed: General register 1 points at the passed parameters Entry to TSS/360: XTRTM, REDTIM, CALL, SIR	ZINIT	LOADW, ABORT
Deletes a list of loaded phases Parameters passed: PAR1 -- address of list of phases to be deleted Entry to TSS: DELETE	RELESE	ZUERR, ABORT
Deletes a list of loaded phases and passes control to either the next requested phase or the next named phase Parameters passed: PAR1 -- address of list of phases to be deleted; PAR2 -- address of name of phase to which control is to be given, or zero Parameters returned: PAR1 -- load point of new phase Entry to TSS/360: DELETE, LOAD(EPLOC), CALL	RLSCTL	Module AD if inter-phase dumping is required; Module AE if it is end of read-in phase; ZUERR, ABORT
Loads the required phase and returns control to the caller. The phase may be loaded again Parameters passed: PAR1 -- address of name of phase to be loaded Parameters returned: PAR1 -- load point of phase Entry to TSS/360: LOAD(EPLOC)	LOADX	ZUERR, ABORT
Marks phases as 'wanted' and 'not wanted' Parameters passed: PAR1 -- address of list of phase names to be marked 'wanted;' PAR2 -- address of list of phase names to be marked 'not wanted' Entry to TSS/360: None	REQEST	ZUERR, ABORT
Puts a record out to SYSLIN Parameters passed: PAR1 -- address of output record Entry to TSS/360: PUT LOCATE(VSAM)	ZULF	LFERRX
Deletes currently called phases and passes control to the error editor Entry to TSS/360: LOAD(EPLOC) if dump option specified	ZABORT, ABORT	Module AD if dump option specified; RLSCTL
Calls module AK to perform finalization Entry to TSS/360: DELETE, CALL		Module AK

Table AA. Module AA Compiler Resident Control Phase (Part 2 of 2)

Statement or Operation Type	Main Processing Routine	Routine Called
Handles all program checks Parameters passed: ARINT holds address of routine wanting to handle interrupt. ARMASK holds mask indicating which interrupts it is desired to handle Entry to TSS/360: None	PIH	ZUERR

Table AA1. Module AA Routine/Subroutine Directory

Routine/Subroutine	Function
ABORT	Deletes currently loaded phases, passes control to error editor.
BLKERR	Enters message "REFERENCED BLOCK NOT IN USE", then terminates compilation.
CONSLD	Takes dictionary reference and points at relevant slot in dictionary control block area (DSLOTS).
CONSLT	Takes text reference and points at relevant slot in text block control area (TSLOTS).
LFERRX	Marks error on SYSLIN data set.
LOADX	Loads required phase and returns control to caller. The phase may be loaded again.
LOADW	Loads required phase and returns control to caller.
PIH	Handles all program checks.
PLERRX	Prints record on PLILIST data set. Pagination (paging action) is performed automatically.
RELESE	Releases all loaded phases.
REQEST	Marks phases as 'wanted' or 'not wanted.'
RLSCTL	Releases all loaded phases and passes control to next required or named phase.
ZABORT	Deletes currently loaded phases and passes control to error editor.
ZEND	Picks up the completion code for the compilation and returns control to ZINT to continue the batch, or to the operating system at the end of a single or batch compilation.
ZINIT	Initializes the compiler.
ZULF	Puts record out to PLILOAD data set.

Table AB. Module AB Compiler Control Initialization

Statement or Operation Type	Main Processing Routine	Routine Called
Prints initial heading and performs scan of option list. Default options are taken where necessary Parameters passed: General register 1 points to option list passed at invocation time Entry to TSS/360: EBCDTIME PUT LOCATE (VSAM)	OPTPROC	None
Makes the initial space allocation for text and dictionary blocks. Sets up communication region Entry to TSS/360: GETMAIN	OPENR	None
Loads intermediate file writer (Module AC). Sets buffer sizes for PLIMAC and opens the data set Entry to TSS/360: LOAD (EPLOC), OPEN	NODUMP	ZUPL (AA)
Prints out list of options for this compilation Entry to TSS/360: None	NDMP	ZUPL (AA)
Reads first card and stores. Uses as heading if required	RDCD	ZURD, ZUERR, ZUPL (all in AA)
Return to pre-initializer in IEMTAA	ABOUT	None
Loads dictionary handling control routines in IEMTAL. Entry to TSS/360: LOAD	LODCNTL	None

Table AB1. Module AB Routine/Subroutine Directory

Routine/Subroutine	Function
ABOUT	Returns control to pre-initializer in Module AA.
NDMP	Prints lists of options for current compilation.
NODUMP	Loads intermediate file writer module AC. Opens PLIMAC data set.
OPENR	Makes initial space allocation for text and dictionary blocks. Sets up communications region.
OPTPROC	Prints initial heading and performs scan of option list.
RDCD	Reads first card.

Table AC. Module AC Compiler Control Intermediate File Control

Statement or Operation Type	Main Processing Routine	Routine Called
Writes a record onto PLIMAC Parameters passed: PAR1 -- address of output record; PAR2 -- length of record Entry to TSS/360: PUT LOCATE(VISAM)	IEMAC	None
Link to file switching routine (Module AG) Entry to TSS/360: CALL	ENDED	None

Table AD. Module AD Compiler Control Interphase Dumping

Statement or Operation Type	Main Processing Routine	Routine Used
Debugging aids. This routine contains a dumping program which is invoked by use of the DUMP option	IEMAD	ZDRFAB, ZTXTAB, ZUPL (all in AA), DUMP

Table AD1. Module AD Routine/Subroutine Directory

Routine/Subroutine	Function
DUMP	Converts contents of specified area of main storage to hexadecimal, prints the result.

Table AE. Module AE Compiler Control Clean-Up Phase

Statement or Operation Type	Main Processing Routine	Routine Called
Input and intermediate file control. Current input file is closed and AC is deleted if present Entry to TSS/360: CLOSE(current input file), DELETE	Module AC	None

Table AF. Module AF Compiler Control Options

Function	Subroutines
This module contains no executable instructions; it contains a table with the default options for the compiler.	None

Table AG. Module AG Compiler Control Intermediate File Switching

Function	Subroutines
Switches PLIMAC from an output file to an input file Entries to TSS/360: OPEN and CLOSE	None

Table AK. Module AK Compiler Control Closing Phase

Function	Subroutines
Closes files, frees scratch core and deletes unwanted phases If batch compiling, scans batch delimiter card for correct syntax and updates completion code. Entries to TSS/360: XTRM, CLOSE, REDTIM, DELETE, and FREEMAIN	ZURC(AA), PLC

Table AL. Module AL Dictionary Phase (Part 1 of 4)

Statement or Operation Type	Main Processing Routine	Routine Called
Releases scratch storage allocated by ZUGC Parameters passed: PAR1 -- a count of the number of entries to ZUGC to be released Entry to TSS/360: FREEMAIN if storage being replaced is outside the guaranteed 4K block	ZURC	ZUERR, ABORT
Inserts diagnostic message in the dictionary and, if required, calls the conversational diagnostic outputter (XZ) Parameters passed: PAR5 -- numeric parameter (if any); PAR6 -- message number; PAR7 -- address of text (if any) or dictionary reference (if any); PAR8 -- length of text (if any) Entry to TSS/360: CALL	ZUERR	ZDRFAB, ZDICRF, ZDICAB, Module XZ
Takes a dictionary reference and points at the relevant slot in the dictionary block control area (DSLOTS) Parameters passed: PAR1 -- dictionary reference Parameters returned: Address of slot in GRA Entry to TSS/360: None	CONSLD	None
Takes a text reference and points at the relevant slot in the text block control area (TSLOTS) Parameters passed: PAR1 -- text reference Parameters returned: Address of slot in GRA Entry to TSS/360: None	CONSLT	
Allocates space for a text block Parameters passed: None Parameters returned: Address of block in GR0 Entry to TSS/360: GETMAIN (VC) if storage available.	TRYMRT	ZUPL, ABORT
Allocates space for a dictionary block Parameters passed: None Parameters returned: Address of block in GR0 Entry to TSS/360: GETMAIN (VC) if storage available.	TRYMRD	ZUPL, ABORT

Table AL. Module AL Dictionary Phase (Part 2 of 4)

Statement or Operation Type	Main Processing Routine	Routine Called
Reads a record from PLIINPUT Parameters passed: PAR1 -- address of input area Parameters returned: PAR2 -- record length Entry to TSS/360: GET MOVE (VISAM)	ZURD	None
Puts a record out to PLILIST. Pagination (paging action) is performed automatically Parameters passed: PAR1 -- address of output buffer. PAR3 -- address of output buffer containing page heading (if any) Entry to TSS/360: PUT LOCATE (VISAM)	ZUPL	PLERRX
Finds a new text block. Optionaly chains the new block to the current block and changes the status of the current block Parameters passed: PAR1 -- optionally, a reference to the current block. PAR2 -- a status and chain indicator Parameters returned: PAR1 -- reference to new block; PAR2 -- absolute address of the beginning of block Entry to TSS/360: None	ZUTXTC	CONSLT, TRYMRT, ZUERR, ABORT, BLKERR
Finds the next text block in the chain. Optionally, changes the status of the current block Parameters passed: PAR1 -- a reference to the current block; PAR2 -- a status indicator Parameters returned: PAR1 -- reference of the next block in the chain. PAR2 -- absolute address of next block in chain Entry to TSS/360: None	ZCHAIN	CONSLT, TRYMRT, BLKERR
Changes the status of the referenced text block Parameters passed: PAR1 -- a reference to the block. PAR2 + 3 -- required 'status' byte Entry to TSS/360: None	ZALTER	CONSLT, BLKERR
Converts a text reference to an absolute address and optionally, does not change status of the block Parameters passed: PAR1 -- reference to be converted and option indicator bit Parameters returned: PAR1 -- the absolute address Entry to TSS/360: None	ZTXTAB	CONSLT, TRYMRT, BLKERR
Converts an absolute address to a text reference Parameters passed: PAR1 -- a text reference to the block containing the absolute address; PAR2 -- the address to be converted Parameters returned: PAR1 -- the required text reference Entry to TSS/360: None	ZTXTRF	CONSLT, BLKERR, ZUERR, ABORT

Table AL. Module AL Dictionary Phase (Part 3 of 4)

Statement or Operation Type	Main Processing Routine	Routine Called
Enters message 'REFERENCED BLOCK NOT IN USE' into dictionary and then terminates compilation Entry to TSS/360: None	BLKERR	ZUERR, ABORT
Supplies storage space for scratch purposes. Allocation is made in 512 bytes at a time Parameters passed: PAR1 -- a count of the number of 512 byte blocks required Parameters returned: PAR1 -- address of the allocated storage Entry to TSS/360: None	ZUGC	TRYMRT, ZUERR, ABORT
Converts an absolute address to a dictionary reference Parameters passed: PAR1 -- any reference to the block containing the absolute address; PAR2 -- the absolute address to be converted Parameters returned: PAR1 -- the required dictionary reference Entry to TSS/360: None	ZDABRF	CONSLD, ZUERR, ABORT, BLKERR
Converts a dictionary reference to an absolute address Parameters passed: PAR1 -- the dictionary reference Parameters returned: PAR1 -- the absolute address Entry to TSS/360: None	ZDRFAB	CONSLD, TRYMRD, BLKERR
Makes an unaligned dictionary entry and returns an absolute address Parameters passed: PAR1 -- address of entry to be made; PAR2 -- length of entry Parameters returned: PAR1 -- address of entry in dictionary. PAR4 -- some reference to the block Entry to TSS/360: None	ZNALAB	ZDRFAB, ZDABRF, TRYMRD, ZUPL, ZUERR, ABORT, CONSLD
Makes an aligned dictionary entry and returns an absolute address Parameters passed: PAR1 -- address of entry to be made; PAR2 -- length of entry Parameters returned: PAR1 -- address of entry in dictionary. PAR4 -- some reference to the block Entry to TSS/360: None	ZDICAB	ZDRFAB, ZDABRF, TRYMRD, ZUPL, ZUERR, ABORT, CONSLD
Makes an unaligned dictionary entry and returns dictionary reference Parameters passed: PAR1 -- address of entry to be made; PAR2 -- length of entry Parameters returned: PAR1 -- reference of entry in dictionary. PAR4 -- absolute address of the entry Entry to TSS/360: None	ZNALRF	ZDRFAB, ZDABRF, TRYMRD, ZUPL, ZUERR, ABORT, CONSLD

Table AL. Module AL Dictionary Phase (Part 4 of 4)

Statement or Operation Type	Main Processing Routine	Routine Called
Makes an aligned dictionary entry and returns a dictionary reference Parameters passed: PAR1 -- address of entry to be made; PAR2 - LENGTH OF ENTRY Parameters returned: PAR1 -- reference of entry in dictionary. PAR4 -- absolute address of the entry Entry to TSS/360: None	ZDICRF	ZDRFAB, ZDABRF, TRYMRD, ZUPL, ZUERR, ABORT, CONSLD
Builds statement/line number table for use by conversational diagnostic routines. Parameters passed: VISAM line number Entry to TSS/360: GETMAIN	STLNBLD	None

Table AL1. Module AL Routine/Subroutine Directory

Routine/Subroutine	Function
TRYMRD	Allocates space for a dictionary block.
TRYMRT	Allocates space for a text block.
ZALTER	Changes status of referenced text block.
ZCHAIN	Finds next text block in chain.
ZDABRF	Converts an absolute address to a dictionary reference.
ZDRFAB	Converts a dictionary reference to an absolute address.
ZDICAB	Makes an aligned dictionary entry and returns absolute address.
ZDICRF	Makes an aligned dictionary entry and returns dictionary reference.
ZNALRF	Makes unaligned dictionary entry and returns dictionary reference.
ZNALAB	Makes unaligned dictionary entry and returns absolute address.
ZTXTAB	Converts text reference to an absolute address.
ZTXTRF	Converts absolute address to a text reference.
ZUERR	Inserts diagnostic message in dictionary.
ZURD	Reads a record from PLIINPUT.
ZUGC	Supplies storage space for scratch purposes.
ZURC	Releases scratch storage.
ZUPL	Puts record out to PLILIST data set.
ZUTXTC	Obtains a new text block.

Table AM. Module AM Compiler Control Phase Marking

Function	Main Processing Routine	Routines Used
Marks all non-optional phases and all phases influenced by compiler invocation-time options	Module AM	REQEST, RLSCTL (both in AA)
Obtains 4K of scratch storage <u>Entry to TSS/360</u>: GETMAIN		

COMPILE-TIME PROCESSOR TABLES

Table AS. Phase AS Resident Phase for Compile-time Processing

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes switches for compile-time processor	ADRP	None
Loads phases for compile-time processor	ADRP	LOADX (AA)
Determines whether Phase BC should be reloaded	ADRP	None

Table AS1. Phase AS Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ADRP	Initializes switches for compile-time processor.
BCKUP2	Backs up token pointer two places.
CHBLK	Changes currently busy IVB block status and gets a new block
CLSBUF	Handles calls to close and write out the buffer. Loads and bases phase BJ if necessary.
COMENT	Scans the limits of a comment, transfers each character into the output buffer.
ENDIVB	Closes an IVB chain.
FREVAL	Releases a chain of IVBs containing a no longer needed value and returns chain to free list.
GETIVB	Removes an IVB from the free chain for use by the calling routine.
GNC	Updates TOKPTR to point to the next character in a particular input stream.
HASH	Accepts an EBCDIC identifier as input and outputs an index. The index indicates the beginning of the HASH chain with which the identifier is associated.
INCTST	Determines whether Phase BC needs to be reloaded on return from Phase BG.
INPUT	Reads in an input record from the source data set or from included text.
INRD	Reads records from the included data set.
NXTTXT	Gets a new text block and sets up address slots.
OUTPTC	Outputs a single character into one of the three output media: IVB's, text blocks, or external records.
SRHDIC	Searches the dictionary for the presence of a named item.
STRING	Scans the limits of a string constant, transfers each character to output.

Table AS1. Phase AS Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
TOKSCN	Examines text, character by character recognizing and returning each logical unit of text (called a token). Tokens include identifiers, constants, operators, delimiters, etc. Handles CHAR48 for macro processing.
UPNEWL	Updates temporary linecount slot.
YAG2	Loads processor phases for the compile-time processor.

Table AV. Phase AV Macro Processing Initialization

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes communication area for compile-time processing	INIT	None
Allocates push down stack from scratch storage	INIT	None
Allocates translation tables	INIT	None
Enters SUBSTR into dictionary	INIT	None
Creates dictionary entries and values for constants pool	INIT	None

Table AV1. Phase AV Routine/Subroutine Directory

Routine/Subroutine	Function
INIT	Entry point to the initialization phase. This initializes the communication region for compile-time processing.
WWN048	Allocates the push down stack (to be used by Phases BC and BG) from scratch storage.
WWOVLP	Sets up tables to translate external code to EBCDIC; tests the BCD, EBCDIC option.
WWOBCD	Enters built-in function SUBSTR into dictionary.
WWCHNBEG	Creates dictionary entries and values for compile-time constant pool.
WWMOVEIT	Moves Subroutine package into core for use by BC.
INCLUDE	INCLUDE Processor
LABELS (BC	LABEL List Processor.
GOTO Subroutine	GOTO Statement Processor.
ACT Package)	Active/Deactivate Processor.
ELSE	ELSE Clause Processor.

Table BC. Phase BC Initial Scan and Translation

Statement or Operation Type	Main Processing Routine	Subroutines Used
Recognizes statement type	PH1SCN	TOKEN, DELETE
Scans until next % character	PH1SCN	FINDPC
Processes PROCEDURE statement	PH1SCN	TOKEN, DELETE, IDSRCH, ADDSP (FREVAL, OUTPTC)
Processes labels attached to statement	PH1SCN	IDSRCH
Encodes statement into internal text	PH1SCN	PARSE, TOKEN, IDSRCH, ADDSP DELETE, CHECK
Cleans up after INCLUDE in initial scan	PH1SCN	None
Begins statement identification process	PH1SCN	None

Table BC1. Phase BC Routine/Subroutine Directory

Routine/Subroutine	Function
ADCONS	Obtains the dictionary reference of a constant, entering it into the dictionary if necessary.
ADDSP	Adds a processor-created item to the dictionary.
ADICT	Adds a normal item to the end of the appropriate hash chain and returns the dictionary reference.
ADPROC (BF)	Processes PROCEDURE statement.
ASSIGN	Processes assignment statements.
CHECK	Checks back for undefined labels and identifiers not declared within the block.
DECLAR (BF)	Declare statement processor.
DELETE	Skips over bad text up to the end of a statement, field or procedure.
DO (BE)	DO statement processor.
DONE (BE)	Checks stack for possible THEN's or ELSE's after statement is completed.
FINDPC	Scans source text, character by character, searching for macro per- cent character.
IDSRCH	Obtains the dictionary reference of an identifier, entering it in the dictionary if necessary.
IF (BE)	IF statement processor.
KYWDSR	Checks for single or multiple keywords.
PARSE (BE)	Parses and generates interpretive macro code for compile-time expressions.

Table BC1. Phase BC Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
PIF4	Provides special handling for end of included text.
PH1SCN (BE)	Main controlling routine for phase.
RETURN	Processes RETURN statement for PROC.
STB3	Collects labels into label list and identifier statement type on first two tokens of statement.
STMT (BE)	Diagnoses statement type and builds label list.
TOKEN	Returns significant tokens to PH1SCN and writes out diagnostics for tokens in error.
UPDLIN	Generates an update linecount instruction.
Note: See also BC Subroutine Package in Table AV1.	

Table BG. Phase BG Final Scan and Replacement

Statement or Operation Type	Main Processing Routine	Subroutines Used
Final scan for replacements	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Recognition of end of text	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Recognition of an identifier	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Recognition of macro action	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Recognition of % character	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Recognition of other characters	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Terminates and cleans up INCLUDE handling	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Re-establishes scan at next higher level text	PH2SCN	OUTPUT, TOKSCN, SRHDIC
Performs replacement on activated identifiers	PH2SCN	OUTPUT, TOKSCN, SRHDIC

Table BG1. Phase BG Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
CLOUT (BJ)	Closes output buffer, and writes out record on PLIMAC.
CONVRT	Handles conversions between the three data types used in the compile-time processor.
DAEOB	Re-establishes scan at next higher level text.
DAEOBF	Recognizes and processes end of text condition.

Table BG1. Phase BG Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
DAIDEN	Recognizes and processes identifier in text.
DAMAC	Recognizes and processes macro action character.
DAOTHR	Recognizes character and outputs it.
DAPENT	Handles replacement operation for text identifiers.
DAPRTC	Recognizes % character and recalls Phase BC if appropriate.
FUNCTN(BJ)	Handles built-in functions.
GETDIC	Picks up a two-byte dictionary reference from scrubbed text, performs error checking, resolves indirect references, and returns both relative and absolute address.
INCONT	INCLUDE control routine. Opens DCB, finds member and sets up buffer.
INTPRT (BI)	Interprets the macro code generated by the Phase I scan.
OUTPT	Handles the output of tokens.
PH2SCN	Scans text blocks.
POP	Pops the top temporary off the Phase II stack.
PROINV (BI)	Special entry point to interpreter for invocation of procedures found in source program text.
PUSH	Pushes next available temporary onto the Phase II stack.
SYNCH	Synchronizes linecount, closing buffer if necessary.
TPEEK	Scans for procedure reference argument list left-parenthesis.
TRAI (BI)	Terminates INCLUDE text handling and frees text blocks containing included text.
ZAASIGN (BI)	Performs identifier assignments for INTPRT.
ZACOMP (BI)	Performs all logical comparison operations for INTRPT.
ZACONCAT (BI)	Performs string concatenations for INTPRT.
ZACVT (BI)	Converts stack items to required type by 'RETURNS' attribute.
ZALGCL (BI)	Performs all logical operations for INTPRT.
ZAPUSH (BI)	Performs stack maintenance for INTPRT.
ZARITH (BI)	Performs all arithmetic operations for INTPRT.
ZATRAI (BI)	Handles transfers from included text to including text.
ZATRAN (BI)	Performs all transfer operations for INTPRT.
ZJSUBS (BJ)	Built-in function SUBSTR.

Table BM. Phase BM Diagnostic Message Determination and Printing

Statement or Operation Type	Main Processing Routine	Subroutines Used
Determines whether error messages are to be printed	XA	None
Scans error message text skeletons and prints them out	XA8	XA50, XA70, XA90, XA110, ZUPL

Table BM1. Phase BM Routine/Subroutine Directory

Routine/Subroutine	Function
XA	Determines whether error messages are to be printed.
XA0	Sets severity code.
XA01	Establishes which message types to suppress.
XA1	Counts number of error chains to be processed.
XA2	Puts out messages if there are no diagnostics.
XA4	Prints out "COMPILER DIAGNOSTIC MESSAGES".
XA7	First scan of message chains.
XA8	Scans error message text skeletons and prints them.
XA9 (BN)	Scans to head of next non-empty chain.
XA12A	Selects and prints header for messages of given severity.
XA30 (BN)	Gets next entry in message chain.
XA32 (BN)	Builds up first part of message in buffer.
XA35 (BN)	Accesses message skeleton.
XA40 (BN)	Puts out completed message.
XA50 (BN)	Moves message text to print buffer.
XA70 (BN)	Converts binary statement number to character representation, and moves it to print buffer.
XA90 (BN)	Converts binary numeric value to character representation and moves it to print buffer.
XA110 (BN)	Moves identifier from dictionary entry to the print area.
ZUPL	Prints a line on PLILIST data set.

Table BW. Phase BW Clean-up Phase

Statement or Operation Type	Main Processing Routine	Subroutines Used
Resets all tables and communications region cells to the value required by the compiler proper	IEMTBW	None

48-CHARACTER SET PREPROCESSOR TABLE

Table BX. Phase BX 48-Character Set Preprocessor

Statement or Operation Type	Main Processing Routine	Subroutines Used
Translates keyword table to internal code and initializes	BA00	None
Reads a record	BA1	ZURD (AA)
Scans text	BA1A	None
Handles operators and keywords	BA5	None
Replaces operator keywords	BA11	None
Replaces comma-dot by semi-colon where applicable	BA20	None
Deals with quote marks	BA25	None
Maintains parenthesis level count	BA30	None
Replaces period-period by colon	BA40	None
Processes a slash	BA50	None
Reads one record ahead in case of need	BA70	None
Restores the situation when a read ahead has taken place	BA80	None
Puts out converted text	BA90	ZUBW

READ-IN PHASE TABLES

Table CA. Module CA Read-In Common Block 1

Function	Subroutines
Provides subroutines common to all five passes of the read-in phase	ACONST, DECINT, EXP, EXPAND, EXPLST, IDENT, MVCHAR, OPTOR, SCONST, SINGLE, SQUID

Table CA1. Module CA Routine/Subroutine Directory

Routine/Subroutine	Function
ACONST	Checks for a valid arithmetic constant.
DECINT	Checks decimal integer.
EXP	Diagnoses expressions.
EXPAND	Expands iterations of string constants and picture characters.
EXPLST	Checks for a list of expressions separated by commas but enclosed in parentheses.
IDENT	Checks for a valid identifier.
MVCHAR	Moves text from one address to another.
OPTOR	Checks for an operator and replaces the two-byte operators by one-byte codes.
SCONST	Checks for a valid string constant.
SINGLE	Diagnoses a single expression in parentheses.
SQUID	Checks for a valid subscripted and qualified identifier.

Table CC. Module CC Read-In Common Block 2

Function	Subroutines
Provides subroutines common to all five passes of the read-in phase	CHAR, CHECK, KEYWD, MESAGE, NONEX, NULINS, OPTEST, PICT, PREC, SOFLOW

Table CC1. Module CC Routine/Subroutine Directory

Routine/Subroutine	Function
CHAR	Diagnoses the CHARACTER and BIT data attributes.
CHECK	Tests the top entry in the stack.
KEYWD	Identifies keywords and hands back the replacement character to the caller.
MESAGE	Provides a diagnostic message.
NONEX	Checks stack for non-executable statements.
NULINS	Inserts null statement in output text.
OPTEST	Tests the output string and moves text to the output.
PICT	Diagnoses a picture. It uses a TRT table set up for the purpose.
PREC	Diagnoses the precision, and the attributes and format items which use it.
SOFLOW	Bumps stack pointer and checks for stack overflow.

Table CE. Modules CE, CK, CN, and CR Read-In Keyword Block

Function	Subroutines
Provides tables of keywords in internal code, together with replacement code. No functional code exists in these modules. Refer to Section 4 for details of keyword tables.	None

Table CI. Phase CI Read-In First Pass

Statement or Operation Type	Main Processing Routine	Subroutines Used
Controls main scan, identifies statements and analyzes some in detail, and calls a subroutine in AL to build statement/line number table.	RSTART	ASSIGN, BADST1, BEGIN, DO, ELSE, BUMP, END, EOP, ERROR, IF, ON, POPLST, PROC, READ, SIGRVT, STAT2, STRING, STLNBLD plus those subroutines contained in modules CA and CC

Table CI1. Phase CI Routine/Subroutine Directory

Routine/Subroutine	Function
ASSIGN (CG)	Diagnoses an assignment statement.
BADST1	Recovers from failure to recognize a statement type; skips to next semi-colon.
BEGIN (CG)	Checks the BEGIN statement and makes an entry in the first pass stack.
BUMP	Advances the input Data Pointer (DP), skips blanks, if any, forcing source text to be read into storage as necessary.
DO (CG)	Checks the DO statements and makes an entry in the first pass stack.
ELSE (CG)	Unstacks an IF compound statement.
END (CG)	Processes three different types of END statements; PROCEDURE-BEGIN; DO; iterative DO.
ENTRY	Processes ENTRY statement.
EOP	Processes end-of-program marker, and returns to compiler control in order to load next pass.
ERROR (CG)	Handles false starts on possible statements.
IF (CG)	Scans the IF statement and makes entry in first pass stack.
ON (CG)	Diagnoses the ON statement and makes entry in first pass stack.
POPLST	Removes prefix options from the text and places them in the dictionary.
PROC	Scans the PROCEDURE and ENTRY statement and makes an entry in the first pass stack.
READ	Reads source text into storage, translating it into internal code, except for character strings; removes comments; prints source listing and prefix options.
RSTART	Controls the first pass scan. Enters statement labels into the dictionary.
SIGRVT (CG)	Scans SIGNAL and REVERT statements.
STAT2 (CG)	Handles all other statements.
STID	Statement identifier routine.
STRING (CG)	Scans character strings.

Table CL. Phase CL Read-In Second Pass

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans for statements handled in this pass, analyzing them in detail. Skips over other statements	SCNA	BUMP, DELAY, DSPLAY, DO, FREE, GOTO, ITDO, LABEL, PROC, RETURN, TRTSC, plus those subroutines contained in modules CA and CC

Table CL1. Phase CL Routine/Subroutine Directory

Routine/Subroutine	Function
BUMP	Increments the input Data Pointer (DP), skipping over blanks, obtaining a new text block if necessary.
DELAY	Processes DELAY statements.
DSPLAY	Processes DISPLAY statements.
DO	Processes DO statements.
EOP	Processes end-of-program marker, and releases control to phase CO or CS, or CV (CO and CS are optional phases).
FREE	Processes FREE statements.
GOTO	Processes GOTO statements.
ITDO	Processes iterative DO statements.
LABEL	Diagnoses LABEL attributes.
OPTION	Handles OPTIONS attribute on PROCEDURE or ENTRY statements.
PROC (CM)	Analyzes PROCEDURE attributes and options, and completes the diagnosis of PROCEDURE and ENTRY statements.
RETURN	Processes RETURN statements.
SCNA	Main controlling routine of this pass.
TRTSC	Skips over all other statements.

Table CO. Phase CO Read-In Third Pass

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans for DECLARE, CALL, and ALLOCATE statements. Analyzes syntax of attributes by calling appropriate subroutines	SCAN2	ATTLST, BUMP, CALLOP, DECL, DEFIND, DIMS, ENTRY, ENVMNT, EOP, GENRIC, LABEL, LIKE, USES, IVLIST, and those subroutines contained in modules CA and CC

Table CO1. Phase CO Routine/Subroutine Directory

Routine/Subroutine	Function
ATTLST	Processes an attribute list. (Recursive)
BDCL	Processes DECLARE or ALLOCATE statement.
BUMP	Advances Data Pointer (DP), obtaining new input block if necessary.
CALLOP (CP)	Checks CALL statements and options.
DECL	Processes the DECLARE and ALLOCATE statements.
DEFIND	Checks the DEFINED attribute.
DIMS	Examines the dimension specifications.
ENTRY	Checks the ENTRY attribute.
ENVMNT (CP)	Removes environment information from the text and inserts it into the dictionary.
EOP	Processes the end-of-program marker, and releases control.
GENRIC	Processes the GENERIC attribute.
IVLIST (CP)	Processes the INITIAL attribute.
LABEL (CP)	Analyzes LABEL attribute.
LIKE	Processes the LIKE attribute.
PSQUID (CP)	Checks for a qualified subscripted identifier in parenthesis.
REFER (CP)	Checks the REFER attribute.
SCAN2	Scans for DECLARE, CALL, or ALLOCATE statements, moves others to the output string unaltered.
SCANT	Moves text to semicolon without alteration.
USES	Deletes the now obsolete USES and SETS attributes from text.

Table CS. Phase CS Read-In Fourth Pass

Statement or Operation Type	Main Processing Routine	Subroutines Used
Controls main scan and identifies I/O statements for further analysis	SCNA	EOP, FORMAT, GET, LIST, OPEN, READ, TRTSC, plus those subroutines contained in modules CA and CC

Table CS1. Phase CS Routine/Subroutine Directory

Routine/Subroutine	Function
EOP	Processes end-of-program marker and releases control.
FORMAT (CT)	Processes the FORMAT statement and format lists.
GET (CT)	Processes GET and PUT statements.
LIST	Processes data lists.
OPEN (CT)	Diagnoses OPEN and CLOSE statements.
READ	Checks the syntax of RECORD I/O statements READ, WRITE, REWRITE, and DELETE. This routine also checks for permissible combinations of these statements.
SCNA	Main scan of this pass.
TRTSC	Skips over all statements other than I/O, moving them to the output text.

Table CV. Phase CV Read-In Fifth Pass

Statement or Operation Type	Main Processing Routine	Subroutines Used
Identifies statements for which it must build chains	SCNA	CALLIN, CHAIN, DECL3, DO3, END3, ENTRY3, EOP, POA1, PROC3, TRTSC, and those subroutines contained in modules CA and CC.

Table CV1. Phase CV Routine/Subroutine Directory

Routine/Subroutine	Function
CALLIN (CW)	Makes up the CALL chain.
CHAIN	Forms chains.
CHECKON	Checks the fifth pass stack for ON entry, in order to insert PROC-END statements round the ON unit.
DECL3	Chains the DECLARE statement to the appropriate PROC or BEGIN statement.
DO3	Makes a stack entry for DO block.
END3	Checks the fifth pass stack.
ENTRY3	Makes an entry in the ENTRY chain.
EOP (CW)	Processes end-of-program marker, and releases control.
ILABSN (CW)	Creates pseudo-assignment statements for initial labels.
POA1	Analyzes prefix options in greater detail.
POC1	Processes check lists.
PROC3	Makes an entry in the PROCEDURE-BEGIN chain.
SCNA	Main controlling routine of the pass.
SCNZ	Extracts statement number for label entry.
TRTSC	Skips over statements not required for analysis in this phase.

DICTIONARY PHASE TABLES

Table ED. Phase ED, Initialization

Statement or Operation Type	Main Processing Routine	Subroutines Used
Sets up routines in scratch storage for phase EL	SETUP	None

Table ED1. Phase ED Routine/Subroutine Directory

Routine/Subroutine	Function
EVENT TASK CELL BASED POINTER OFFSET	Routines for processing declared attributes. These set up information in the attribute collection area of scratch core, for reference by CDICEN, etc., in phase EL.

Table EG. Phase EG Dictionary Initialization

Statement or Operation Type	Main Processing Routine	Subroutines Used
Hashes labels	CAA1	CHASH, CBCDL2
PROCEDURE-BEGIN chain	CA7	None
BEGIN	CA8A	None
PROCEDURE	CAPROC	CANATP, CFORP
ENTRY	CA10	CANATP, CFORP
Formal parameters	CFORP	CHASH, CBCDL2
Attribute list	CANATP	CAPRE1, CATCHA, CATBIT, CATPIC
Creates entry type 2 entries for labels	CTYPBL	ENT2F, CDEFAT

Table EG1. Phase EG Routine/Subroutine Directory

Routine/Subroutine	Function
CAA1	Scans label table and hashes labels.
CANATP	Processes attribute list.
CAPROC	Processes PROCEDURE statements.
CAPRE1	Processes precision data.
CATBIT	Processes BIT attribute.
CATCHA	Processes CHARACTER attribute.
CATPIC	Processes PICTURE attribute.
CA6	Scans the PROCEDURE-BEGIN chain for the relevant statements, and sets bits in Dictionary entries for optimization options on PROCEDURE and BEGIN statements.
CA8A	Processes BEGIN statements.
CA10	Processes ENTRY statements.
CBCDL2	Traverses the hash chain looking for entries with the same BCD as that just found.
CDEFAT	Completes data byte for entry type 2 entries by default rules.
CFORP	Processes formal parameter lists.
CHASH	Obtains an address in the hash table for an identifier.
CTYPBL	Creates entry type 2 entries for labels.
ENT2F	Creates or copies second file statements.
TYPW	Scans ENTRY chain.
OPTN1 (EF)	Checks containing block options, for inheritance.
OPTN2 (EF)	Processes procedure options.
OPTN3 (EF)	Performs post processing, makes STATIC DSA decisions.
ATTRBT (EF)	Processes POINTER, OFFSET, and AREA attributes.

Table EI. Phase EI Dictionary Declare Pass One

Statement or Operation	Main Processing Routine	Subroutines Used
Scans DECLARE statement	CCGS0	None
Scans text	CCGS2	None
Processes structure level	CCGSCM	None
Factored attribute, left parenthesis	CCFLP	CFPMCR
Factored attribute, right parenthesis	CCFRP	None
Data following DEFINED attribute	CCDEF	NEWBLK, CTXTRM
POSITION	POSIT	None
CHARACTER, BIT	CHABIT	CTXTRM
PICTURE	CATPIC	None
LIKE	LIKE	None
KEY	KEYED	None
Dimension	CDDIMS	CTXTRM, AST, TOMENE, ERRORB
Precision	CDPREC	ERRNEG, SCLBIG
INITIAL	EJINIT	CECON, EHINIT
INITIAL CALL	INCALL	CTXTRM
OFFSET	OFFSET	CTXTRM
BASED	BASED	PTVEXP
AREA	AREA	CTXTRM

Table EI1. Phase EI Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
AREA	Processes AREA attributes.
AST	Deals with the case of* dimension bounds mixed with non -* bounds.
BASED (EH)	Entry point in OFFSET routine, at which second file statement is made.
CATPIC	Processes PICTURE attributes.
CCDEF	Processes data following DEFINED attribute.
CCFLP	Processes factored attributes (left parenthesis).
CCFRP	Processes factored attributes (right parenthesis).
CCGSCM	Processes structure level.
CCGSAT	Attribute routine selector.
CCGSE	Scans DECLARE chain.
CCGS00	Scans text.

Table EI1. Phase EI Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
CCGS2	Scans source text.
CDDIMS (EJ)	Processes dimension attributes.
CDPREC (EJ)	Processes precision attributes.
CECON (EH)	Makes a dictionary entry for a constant unless one has already been made. Returns the dictionary reference of the constant entry.
CFPMCR	Obtains more storage for the factored attribute table.
CHABIT	Processes CHARACTER and BIT attributes.
CSGS00	Detects end of DECLARE chain.
CTXTRM	Tests for space in current text block and obtains new block if necessary.
EHINIT (EH)	Processes the INITIAL attribute except for the initialization of label variables and INITIAL CALL.
EJINIT (EJ)	Processes INITIAL attribute and LABEL with a label-constant list.
ERRNEG	Deals with the case of a negative precision specification.
ERRORB	Deals with the case of lower dimension bound declared greater than the upper bound.
Gentry	Keeps a count of parentheses in GENERIC and ENTRY processing.
INCALL (EJ)	Processes INITIAL CALL attributes.
IVROOM (EH)	Checks if there is space in scratch storage for another entry. If not, it makes a dictionary entry and chains it to the previous one or to the C8 in text as required.
IVPUTL (EH)	Places a dictionary reference in the 'initial list' for a label constant. If the constant is not known, a dummy reference is inserted.
IVPUTC (EH)	Places a dictionary reference in the 'initial list' for a constant.
IVPUT0 (EH)	Places the dictionary reference of zero in the 'initial list' for a negative or imaginary replication factor.
KEYED	Processes KEY attributes.
LIKE	Processes LIKE attributes.
NEWBLK	Obtains new text block.
OFFSET (EH)	Processes OFFSET attributes.
POSIT	Processes POSITION attributes.
PTVEXP (EH)	Entry point in OFFSET routine, at which secondfile statement is made.
SCLBIG	Deals with the case when a precision specification for fixed-point data is declared too large.
SECON	Creates a dictionary entry for a constant provided the appropriate entry has not been already made.
SETS	Processes USES and SETS attributes.
TOMENE	Deals with the case when the number of dimensions declared is greater than 32.

Table EL. Phase EL Dictionary Declare Pass Two

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans chain of DECLARE statements	CGENSC	CDCLSC
Scans each item of DECLARE statement	CDCLSC	ATLSCN, BCDPR, CDFLT, CDICEN, CDIMAT, DCIDPR, INTLZE, POSTPR, SELMSK, STRPR
Initializes each identifier declared	INTLZE	DCIDPR
Processes factor brackets and level numbers	DCIDPR	TEMSCN, BCDPR
Scans for next level number	TEMSCN	CDATPR
Processes BCD of identifier	BCDPR	BCDISB, CHASH, SELMSK
Hashes BCD of identifier	CHASH	None
Scans list of attributes following identifier	ATLSCN	CDATPR
Applies factored attributes	CDFATT	CDATPR
Applies implicit attribute	IMPATT	None
Attributes controlling routine	CDATPR	CDAT40, CDAT41, CDAT42, CDAT43, CDAT44, CDAT45, CDAT48, CDAT49, CDAT4A, CDAT4B, CDAT4C, CDAT4D, CDAT4F, CDAT54, CDAT55, CDAT56, CDAT57, CDAT58, CDAT59, CDAT60, CDAT61, CDAT62, CDAT63, CDAT64, CDAT69, CDAT6A, CDATB4, CDATB8

Table EL1. Phase EL Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ATLSCN	Scans the list of attributes following the identifier.
BCDISB	Checks for multiple declarations, etc.
BCDPR	Processes BCD of identifier.
CDATPR (EK)	Attribute controlling routine.
CDAT40 (EK)	Processes DECIMAL attribute.
CDAT41 (EK)	Processes BINARY attribute.
CDAT42 (EK)	Processes FLOAT attribute.
CDAT43 (EK)	Processes FIXED attribute.
CDAT44 (EK)	Processes REAL attribute.
CDAT45 (EK)	Processes COMPLEX attribute.
CDAT46 (EK)	Processes precision attributes.
CDAT48 (EK)	Processes VARYING attribute.
CDAT49 (EK)	Processes PICTURE attribute.
CDAT4A (EK)	Processes BIT attribute.
CDAT4B (EK)	Processes CHARACTER attribute.
CDAT4C (EK)	Processes FIXED DIMENSIONS attribute.
CDAT4D (EK)	Processes LABEL attribute.
CDAT4F (EK)	Processes ADJUSTABLE DIMENSIONS attribute.
CDAT58 (EK)	Processes ENTRY attribute.
CDAT59 (EK)	Processes GENERIC attribute.
CDAT5A (EK)	Processes BUILT-IN attribute.
CDAT60 (EK)	Processes EXTERNAL attribute.
CDAT61 (EK)	Processes INTERNAL attribute.
CDAT62 (EK)	Processes AUTOMATIC attribute.
CDAT63 (EK)	Processes STATIC attribute.
CDAT64 (EK)	Processes CONTROLLED attribute.
CDAT69 (EK)	Processes INITIAL attribute.
CDAT6A (EK)	Processes LIKE attribute.

Table EL1. Phase EL Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
CDAT6B (EK)	Processes DEFINED ATTRIBUTE.
CDAT6C (EK)	Processes ALIGNED attributes.
CDAT6D (EK)	Processes UNALIGNED attribute.
CDAT70 (EK)	Processes AREA attribute.
CDAT88 (EK)	Processes POS attribute.
CDCLSC	Scans each item of DECLARE statement.
CDFATT (EM)	Applies factored attributes.
CDFLT (EM)	Applies default attributes.
CDICEN (EM)	Constructs dictionary entry.
CGENSC (EM)	Performs phase initialization and scans chain of DECLARE statements.
CHASH (EM)	Hashes BCD of identifier.
DCID1	Main scan routine.
DCIDPR	Processes factor brackets and level numbers.
ECHSKP (EK)	Initializes and passes control to Module EM.
IMPATT (EM)	Applies implicit attributes.
INTLZE	Performs initialization for each identifier declared.
POSTPR	Postprocessor.
SCAN4 (EM)	Scans chain of DECLARE statements.
SELMSK	Selects correct test mask to be initialized.
STRPR	Processes inheriting of dimensions in structures.
TEMSCN	Scans ahead for next level number.

Table EP. Phase EP Dictionary Entry III and Call

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans for PROCEDURE entries type 1	ENTRY3	None
Follows chain of ENTRY statement entry type 1 entries from a PROCEDURE entry type 1	EPL40	None
Examines all labels belonging to an entry type 1, constructing an entry type 2 or 3, if necessary	LBPROC	None
Follows CALL chain in text making dictionary entries for entry points	EPL290	None
Examines the first character of an identifier and sets a flag indicating the range in which it lies	CDIMAT	None
Applies default rules	CDFLT	None
Given an identifier calculates its offset in the hash table	CHASH	None
Constructs a dictionary entry	CDICEN	None
Sets address slot to zero or the end of the dictionary	FNDEND	None
Constructs list of numbers of known blocks	BLDST2	None
Built in function name	SCANBF	None

Table EP1. Phase EP Routine/Subroutine Directory

Routine/Subroutine	Function
BLDST2	Constructs list of numbers of known blocks.
CDICEN	Constructs dictionary entry.
CDIMAT	Sets flag for default routine.
CDFLT	Applies default rules.
CHASH	Calculates offset in hash table for given BCD.
ENTRY3	Scans ENTRY chain for PROCEDURE statements.
EPL20	First entry in entry type 1 chain.
EPL40	Scans ENTRY chain for ENTRY statements type 1.
EPL75	Return point from LBPROC routine.
EPL100	Processes new entry label.
EPL290	Scans CALL chain.
EPL340	Searches built-in function table for BCD of identifier.
EPL360	Blanks out BCD in text.
EPL600	Scans the CALL chain.
FNDEND	Sets address slot for label.
LBPROC	Processes labels of PROCEDURE or ENTRY statements.
PHSINT	Initialization of phase.
PHSMRK	Marks later modules as 'wanted' or 'not wanted'.
SCANBF	Checks for built-in function name.

Table EW. Phase EW Dictionary LIKE

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans LIKE chain	EWBEGN	EWCOPY, EWELDM, EWINCH, EWONDM
Updates hash chain for new entry	EWHSCN	None
Calculates start of structure data from start of variable information	EWVART	None
Changes error entry to base element	EWCHEN	None
Copies dimension table entry and second file statement	EW2FNT	EWNWBK

Table EW1. Phase EW Routine/Subroutine Directory

Routine/Subroutine	Function
ALIGN (EV)	Provides correct alignment of base elements in likened structure.
BASED (EV)	Inserts or deletes defined slot, where only one structure is based.
CESCN	Scans dictionary to find entry corresponding to BCD in text.
EWBEGN	Scans LIKE chain.
EWCHEN	Changes error entry to base element.
EWCOPY	Copies dictionary entry into scratch storage.
EWDCCY (EV)	Copies initial dictionary entries and associated second file statements, etc.
EWELDM	Copies entry into scratch storage with dimension data removed.
EWELTS	Tests whether the likened structure is dimensioned.
EWEND	Handles transfer of control to next phase.
EWERNC	Processes erroneously "likened" major structure.
EWHSCN	Updates hash chain for new entry.
EWINCH	Completes entry copy and places it in dictionary.
EWNOLK	Tests whether original structure is dimensioned.
EWNWBK (EV)	Obtains new dictionary block and terminates current one in use.
EWONDM	Copies entry into scratch storage, inserting dimension information.
EWORDM	Processes dimension information in original structure.
EWSTRT	Tests validity of likened structure.
EW2FNT (EV)	Copies second file statement and associated dictionary reference.

Table EY. Phase EY Dictionary ALLOCATE

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for explicitly pointer-qualified based variables	IEMEX	EY14
Copies dictionary entries for explicitly qualified based variables	EY14	HASH, ATPROC, DICBLD, STRCPY
Second file pointers. Scans ALLOCATE statements	IEMEY	ATPROC, DICBLD, HASH, STRCPY
Completes copied dictionary entry for an allocated item	ATPROC with second entry point ATPROD	MOVEST
Controls ATPROC and ATPROD routines for each member of a structure	STRCPY	ATPROC, ATPROD

Table EY1. Phase EY Routine/Subroutine Directory

Routine/Subroutine	Function
ATPROC/ATPROD (EZ)	Complete copied dictionary entry for allocated item by including attributes from ALLOCATE and second file statements.
DICBLD	Collects attribute given for an identifier and copies its dictionary entry.
EY16	Processes ALLOCATE statements.
EY17	Processes identifier in ALLOCATE statement.
EY21	Processes major structures.
HASH	Hashes BCD of identifier to obtain its dictionary reference.
IEMEX	Scans text for explicitly pointer-qualified variables.
EY14	Copies dictionary entries for explicitly qualified based variables.
IEMEY	Scans second file, reverses pointers. Scans ALLOCATE statements.
MOVEST (EZ)	Copies second file statement and associated dictionary entry.
STRCPY	Controls ATPROC and ATPROD for each member of structure.

Table FA. Phase FA Dictionary Context

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	CE30	CENDTS, CETRAN
Reorders subscripts; makes dictionary entry for file and event variables	CEID	CESCN
Identifies keywords	CEKYWD	CEKEND, CEKEOB, CEKEOP, CEKON, CEKPRC, CEKSND
Scans dictionary	CESCN	CESTUC, CEYES, CFPDER, CFPDR2, CHASH, CE3XX
Makes dictionary entry for variables	CFPDR2	CDFLT, CDICEN, CDIMAT, CEONCK
Scans dictionary entry for constants and makes new entry, if necessary.	CECON	CHASH
Scans PICTURE chain entry and makes new entry, if necessary.	CEPICT	None

Table FA1. Phase FA Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
CDFLT	Determines default attributes for identifier.
CDICEN	Constructs default dictionary entry for identifier.
CDIMAT	Determines default scale for identifier.
CEBNK	Transfer point for zero or blank.
CECON (FB)	Scans dictionary entry for constants.
CEDWAX	Subscript prime text marker.
CEID	Reorders subscripts and makes dictionary entries for files and event variables.
CEINT	Transfer point for constant routine.
CEISUB	Transfer point for iSUB.
CEKCEN	Transfer point for CALL to get over chain.
CEKDCL	Removes SN from DECLARE statements.
CEKEND	Processes END keyword.
CEKEOB	Processes end-of-block marker.
CEKEOP	Handles end-of-program marker, or start of second file.
CEKEY	Transfer point for keyword.
CEKIDO	Transfer point for iterative DO.
CEKON	Processes ON keyword.

Table FA1. Phase FA Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
CEKPFR	Transfer point for picture format item.
CEKPRC	Processes PROCEDURE keyword.
CEKSN	Moves SN, etc., to output stream.
CEKSND	Processes start of second file statement.
CEKYWD	Identifies keywords.
CELP	Transfer point for left parenthesis.
CENDTS	End of text block in output file routine.
CEONCK	Makes entry for programmer-named ON condition.
CEPFDR	Makes dictionary entry for variables.
CEPICT (FB)	Scans picture chain entry.
CERP	Transfer point for right parenthesis.
CESCN	Scans dictionary.
CESMCL	Handles semicolon.
CESTUC	Points at next entry in structure chain.
CETRAN	Translates keyword into transfer instruction.
CEYES	Compares structure levels.
CE2L	Transfer point for second level marker.
CE30	Controlling scan of text.
CE31	Tests for end of block.
CE32	Moves one byte to output stream.
CE300	Switches to appropriate routine.
CE3XX	Compares identifier in text with entry in dictionary.
CFPDER (FB)	Makes dictionary entry for ordinary identifier.
CFPDR2 (FB)	Makes dictionary for formal parameter.
CHASH	Hashes identifier.
CHASHC	Hashes constant.
IEMFA	Initializes phase.

Table FE. Phase FE Dictionary BCD to Dictionary Reference

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	CE30	CENDTS, CETRAN
Scans dictionary	CESCN	CESTUC, CEYES, CFPDER, CFPDR2, CHASH, CE3XX
Checks for array, function, or pseudo-variable if left parenthesis is found	CELP	CEFNCT
Tests for end of text block	CENDTS	CEKEND, CEKIDO, CEKPRC
Identifies keywords	CEKYWD	CEKEOB, CEKEOP
Makes dictionary entry	None	CDFLT, CDICEN, CDIMAT

Table FE1. Phase FE Routine/Subroutine Directory

Routine/Subroutine	Function
CDFLT	Applies default rules.
CDICEN	Constructs dictionary entry.
CDIMAT	Sets flag for default routine.
CEFNCT	Tests validity of function reference in text.
CEKEND	Processes END keyword.
CEKEOB	Processes end-of-block marker.
CEKEOP	Processes end-of-program marker, or start of second file.
CEKIDO	Processes iterative DO keyword.
CEKPRC	Processes PROCEDURE keyword.
CEKYWD	Identifies keyword.
CELP	Checks for array, function, or pseudo-variable if left parenthesis is found.
CENDTS	Tests for end of text block in output file.
CESCN	Scans dictionary.
CESTUC	Points at next entry in structure chain.
CETRAN	Translates keyword into transfer instruction
CEYES	Compares structure levels.
CE30	Controlling scan of text.
CE3XX	Compares identifier in text with dictionary entry.
CFDICN (FF)	Makes dictionary entry.
CFPDER	Makes dictionary entry for statement with ordinary identifiers.
CFPDR2	Makes dictionary entry for formal parameters.
CHASH	Calculates offset in hash table for given BCD.

Table FI. Phase FI Dictionary Checking

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	CESTRT	CEKEYW
Identifies keywords	CEKEYW	CEKEOB, CEKEOP, CEKIDO, CEKSN
Checks GOTO statement references	CEGOTO	None
Converts GOTO to GOOB, if necessary	CEGOB	None
Checks file references	CEFILE	None
Checks data list items for validity	CEDTCK	None

Table FI1. Phase FI Routine/Subroutine Directory

Routine/Subroutine	Function
CECMBK	Tests value of previous second level marker.
CEDDOL	Processes function names used as control variables for DO groups.
CEDOND	Processes end of iterative DO groups.
CEDREF	Tests whether dictionary reference needs to be checked.
CEDTCK	Checks data list items for validity.
CEFILE	Checks file references.
CEFNMK	Processes function markers.
CEGOB	Converts GOTO to GOOB, if necessary.
CEGOTO	Checks GOTO statement references.
CEISUB	Processes iSUBs.
CEJUMP	Bumps scan pointer over dictionary reference.
CEKEND	Processes END statements.
CEKEOB	Processes end-of-block marker.
CEKEOP	Processes end-of-program marker.
CEKEYW	Identifies keywords.
CEKIDO	Processes iterative DO keyword.
CEKON	Processes ON statements.
CEKSN	Processes statement number.
CELRCT/CERPCT	Process left and right parentheses.
CEOOPS	Checks validity of keywords in the text.
CEPRBG	Processes PROCEDURE and BEGIN statements.
CERFMT	Processes remote format references.
CESMCL	Processes semicolons.
CESTRT	Controlling scan of text.

Table FK. Phase FK Dictionary Attribute

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans attributes area for SETS lists	FO1A	None
Scans SETS list	FO2	None
Processes constants	CONPRO	None
Processes identifiers	CESCN	CESTUC, CE3XX, CHASH

Table FK1. Phase FK Routine/Subroutine Directory

Routine/Subroutine	Function
CEIDLP	Scans qualified name.
CENQUL	Processes unqualified name.
CESCN	Processes identifier.
CESTUC	Finds address of next structure in chain.
CE3XX	Compares current BCD with BCD in hash chain.
CHASH	Calculates offset in hash table for given BCD.
CMPERR	Provides termination error action.
CONPA	Inserts constant in ordered stack.
CONPRO	Processes constants.
ENDFO	Releases control.
FOERR2	Diagnoses constant greater than 255.
FO1A	Scans attribute tidy-up area.
FO2	Scans SETS list.
FO4	Completes SETS dictionary entry.
GETSCR	Obtains scratch storage.

Table FO. Phase FO Dictionary ON

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans input text for ON, SIGNAL, and REVERT statements	FKMVIT	BEFTRN, CENDTS, QP
Moves second file from input text block to output text block	F2	CENDTS, BEFTRN
Makes dictionary entries for ON-conditions found in ON, SIGNAL, and REVERT statements	FKDCEN	LABCD
Examines BCD of file entries referenced in ON, SIGNAL, and REVERT statements; scans previous entries for ON conditions	MVSIG	CENDTS
Processes CHECK and NOCHECK list.	BEFCHL	CENDTS, LABCD
Creates dictionary entries for condition prefixes	NOMOVE	QP

Table FO1. Phase FO Routine/Subroutine Directory

Routine/Subroutine	Function
BEFCHL	Processes CHECK and NOCHECK list.
BEFTRN	Replaces statements containing dummy dictionary references by error statements, and generates error message.
CENDTS	Requests a new text block for output.
FKDCEN	Makes dictionary entries for ON conditions found in ON, SIGNAL, and REVERT statements.
FKMVIT	Scans input text for ON, SIGNAL, and REVERT statements.
FKNOCK	Processes CHECK and NOCHECK lists.
FKPROC	Scans input text for ON, SIGNAL, and REVERT statements.
FP010 (FP)	Chains initial label statements and makes second file dictionary entries for each label array initialized in this way.
F2	Moves second file from input text block to output text block.
LABCD	Creates a dictionary entry for each label constant and each entry label mentioned in a CHECK list.
MVSIG	Examines BCD of file entries referenced in ON, SIGNAL, and REVERT statements; scans previous entries for ON conditions.
NOMOVE (FP)	Creates dictionary entry for condition prefix.
Q3	Processes condition prefixes changed in current block.
QP	Determines which condition prefixes require dictionary entries.
R8	Moves statement to output buffer.

Table FQ. Phase FQ Dictionary Picture Processor

Statement or Operation Type	Main Processing Routine	Subroutines Used
Controls scan of PICTURE chain; initializes	CYBR3	CYEK, CYFIND, CYTABL
Picture character 9	CYNINE	None
Picture characters S, $, +, -.	CYSDPM	None
Picture character V	CYV	None
Picture character E	CYE	CYC21
Picture character K	CYK	CYC21
Picture characters CR, DB	CYCRDB	None
Picture characters 1,2,3	CYOTT	None
Picture character P	CYP	None
Picture character Z	CYZ	None
Picture character *	CYAST	None
Picture character Y	CYY	None
Picture character G	CYG	None
Picture characters 6, 7, 8, H	CYSSEH	None
Picture character M	CYSTM	None
Picture character F	CYF	None
Converts integer constants to scale factor	CYC97	CYCONV
Calculates scale factor	CYFNT	None

Table FQ1. Phase FQ Routine/Subroutine Directory

Routine/Subroutine	Function
CYAST	Processes picture character *.
CYBR2	Identifies picture character.
CYBR3	Controlling scan of PICTURE chain.
CYCONV	Converts integer constant to scale factor.
CYCPBS	Processes picture characters slash (/), comma(,), point (.), and B.
CYCRDB	Processes picture characters CR, DB.
CYC21	Adjusts data to terminate picture before illegal character.
CYC97	Converts integer constant to scale factor.
CYE	Processes picture character E.
CYEK	Completes entry for correct picture.
CYENDD	Releases control at end of picture chain.
CYF	Processes picture character F.
CYFIND	Obtains code for next character in picture.
CYFNT	Calculates scale factor.
CYG	Processes picture character G.
CYK	Processes picture character K.
CYNINE	Processes picture character 9.
CYOTT	Processes picture characters 1,2,3.
CYP	Processes picture character P.
CYSDPM	Processes picture characters S, $, +, -.
CYSS	Processes picture characters 6,7.
CYSSEH	Processes picture characters 8,H.
CYSTM	Processes picture character M.
CYTABL	Code table for picture characters.
CYV	Processes picture character V.
CYY	Processes picture character Y.
CYZ	Processes picture character Z.

Table FT. Phase FT Dictionary Scan

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans second file	AC1	None
Scans dictionary	B1	None
Data variables	DATVAR	None
Event or label variables	EVLABV	None
Dimension attributes	F0	None
Scans AUTOMATIC chain	G2	None
Scans STATIC chain	G3	None
Scans CONTROLLED chain	GE1	None
Sets dope vector required bit	P1A	None
ENTRY type 1 entries	QA4	None
ENTRY type 2 entries	QA3	PROPIC
ENTRY type 3 entries	QA2	None
ENTRY type 4 entries	QX	None
ENTRY type 5 and 6 entries	QA1	PROPIC
Constants	CONST	None
Structures	STRUCT	AJDMRT, MKDMTB, MVTXT

Table FT1. Phase FT Routine/Subroutine Directory

Routine/Subroutine	Function
AC1	Scans second file.
AC2	Detects second file statement marker.
AF3	Points relevant dictionary entry at statement.
AJDMRT	Modifies second file statements to initialize dope vectors for base elements, rather than for the containing structures.
B1	Scans dictionary.
BIA	Initializes dictionary scan.
CONST	Processes constants.
DATVAR	Processes data variables.
EVLABV	Processes event or label variables.
F0	Processes dimension attributes.
FULIN	Moves initial label statement to the second file, collecting together all statements for the same array.
GE1	Scans CONTROLLED chain.
G2	Scans AUTOMATIC chain.
G3	Scans STATIC chain.
MKDMTB	Creates dimension tables.
MVTXT	Moves text blocks.
PROPIC	Extracts precision data from picture tables.
P1A	Sets 'dope vector required' bit.
QA1	Processes ENTRY type 5 and 6 entries.
QA2	Processes ENTRY type 3 entries.
QA3	Processes ENTRY type 2 entries.
QA4	Processes ENTRY type 1 entries.
QX	Processes ENTRY type 4 entries.
STRUCT	Processes structures.
TRVECT	Transfer vector for appropriate chaining routine.

Table FV. Phase FV Dictionary Second File Merge

Statement or Operation Type	Main Processing Routine	Subroutines Used
Reverses second file pointers; scans text for block heading statements; allocates statements and references to dynamically defined data	IEMFV	DATCPY, DEFMOV, DEFTST, F2MOVE, MOVE
Examines ADF references in second file; completes defined item dictionary entry	DEFCOM	None
Detects dictionary references which refer to dynamically defined data	DEFTST	None
Examines dictionary references and moves any associated second file statements to the output string	DATCPY	F2MOVE, MOVE
Inserts dictionary reference of pointer in associated based variable entry	FVPTR	None
Processes adjustable extents on based arrays	FVADV	None
Processes adjustable lengths on based strings	FVSDV	None

Table FV1. Phase FV Routine/Subroutine Directory

Routine/Subroutine	Function
DATCPY	Moves second file statements associated with dictionary reference to output string.
DEFCOM (FW)	Examines ADF references in second file; completes defined item dictionary entry.
DEFMOV	Modifies text references to dynamically defined data.
DEFTST	Detects dictionary references which refer to dynamically defined data.
FV0	Scans second file reversing pointers.
FV9	Initializes text scan.
FV10	Scans text.
FV16	Releases control.
FV18	Processes ALLOCATE statements.
FV19	Processes PROCEDURE statements.
FV20	Processes BEGIN statements.
FV34	Scans AUTOMATIC chain.
FVPTR	Inserts D.R. of pointer in associated based variable entry.
FVADV	Processes adjustable extents on based array.
FVSDV	Processes adjustable lengths on based strings.
F2MOVE	Moves second file statement to output string.
IEMFV	Controlling scan of second file; invokes processing routines.
MOVE	Moves text from input string to output string.

PRETRANSLATOR PHASE TABLES

Table GA. Phase GA DCLCB Generation

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans STATIC chain	IL0100	IL0110, IL0120
Generates DECLARE control block entry	IL0110	CHKATT, IHEENV
Generates OPEN control block entry	IL0120	CHKATT

Table GA1. Phase GA Routine/Subroutine Directory

Routine/Subroutine	Function
CHKATT	Checks attributes and creates control words.
IHEENV	Checks environment options, and inserts them into DECLARE control blocks.
IL0000	Entry point from compiler control.
IL0100	Scans STATIC chain.
IL0110	Generates DECLARE control block entry.
IL0114	Test point for environment entry.
IL0115	Return point from environment processing.
IL0117	Processes file attributes entry.
IL0118	Branch point of SYSPRINT file found.
IL0120	Generates OPEN control block entry.
IL0200	Releases control.

Table GB. Phase GB Pretranslator I/O Modification

Statement or Operation Type	Main Processing Routine	Subroutines Used
Removes all second level markers	Throughout phase	None
Reorders options to put EDIT, DATA or LIST last	A8	SCNS, SCAN2
Moves DO specifications to precede relevant list in data lists, adds END statements	SCAN2	LLDOIT
Expands iteration factors in format lists	FORLST	None
Checks for use of COBOL files in READ, WRITE, and LOCATE Statements	A4	LOCATE, READ, WRITE, DELETE, MAP, COPY, STSCAN

Table GB1. Phase GA Routine/Subroutine Directory

Routine/Subroutine	Function
AFORMT	Processes FORMAT statements.
A4	Checks for use of COBOL files in READ, WRITE, and LOCATE statements.
A6	Scans source text for GET and PUT statements.
A8	Re-orders options to put EDIT, DATA, or LIST last.
A21	Scans GET or PUT statement for data specification.
COPY (GC)	Copies a structure and places the copy on the COBOL chain. Sets up text skeletons.
DELETE (GC)	Removes an offending I/O statement and inserts an error statement in the output text.
FORLST	Expands iteration factors in format lists.
F2	Creates and buys integer temporary.
F5	Scans and outputs format item.
F5A	Sells temporary.
F6	Tests for end of format list.
F6A	Tests for end of format specification.
F6B	Outputs end of format specification.
F7	Scans format list.
LAB17B	Processes format list in GET or PUT statement.
LLDOIT	Moves DO specifications to precede relevant list in data lists, adds END statements.
LOCATE (GC)	Checks for the use of a COBOL file, and puts out a warning diagnostic.
MAP (GC)	Compares the PL/I and COBOL and PACKED(NONSTRING) mappings of a structure.
MKROOM	Provides space in a statement in new source file.
MR	Initializes text blocks and pointers, and obtains scratch storage.
READ (GC)	If a READ IGNORE is encountered, no action is taken. If a READ SET is encountered, a warning diagnostic is given. If a READ INTO for a structure is encountered, PL/I and COBOL mappings are compared.
SCAN2	Scans option list for end of option or statement, expands DO specifications, and changes certain function markers into pseudo-variable markers.
SCNS	Scans option list for end of option or statement.
STSCAN (GC)	Stores the dictionary reference of the file and the INTO/FROM variable, and sets flags.
WRITE (GC)	For structures, the PL/I and COBOL mappings are compared.

Table GK. Phase GK Pretranslator Parameter Matching 1

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans source text for function markers	BASCAN	CPSTMT, CRSTMT
Processes function, puts out reference and initial code bytes	BAFM	SCANRP
Processes arguments	BALOOP	ADDTGT, SCNCRP
Checks numbers of arguments	ARGNOQ	None

Table GK1. Phase GK Routine/Subroutine Directory

Routine/Subroutine	Function
ADDTGT	Adds data to output text.
ARGNOQ	Checks number of statements.
BABT3	Tests for STOP marker.
BACALQ	Outputs function and first bytes of argument list.
BADELM	Tests for end of argument list.
BAFM	Processes function, puts out reference and initial code bytes.
BAFST	Locates SETS list and parameter list for function.
BALOOP	Processes arguments.
BALPQ	Tests whether argument list is present.
BAMORE	Accesses next argument in list.
BANORM	Sets STOP marker to scan argument.
BAPVM	Examines pseudo-variable.
BARECQ	Tests for nested function reference.
BARGFN	Outputs warning message.
BASCAN	Scans source text for function markers.
BASTOP	Outputs argument.
CPSTMT	Adds closing bytes of a statement to output text.
CRSTMT	Adds first bytes of a statement to output text.
SCANRP	Scans argument list.
SCNCRP	Scans argument.

Table GO. Phase GO Preprocessor Parameter Matching 2

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initialization and scratch core utilization for Parameter Matching 2	PMATCH	POLYMV

Table GO1. Phase GO Routine/Subroutine Directory

Routine/Subroutine	Function
PMATCH	General initialization and scratch core utilization for Parameter Matching 2.
POLYMV	Moves the routines POLY1, POLY2, POLY3, POLY4 and POLY5 into scratch storage (see Table GP).

Table GP. Phase GP Pretranslator Parameter Matching 2

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for procedure and function calls	BS1	ADDTT, STKINF, UNSTCK
Examines argument lists for expressions	BS4	EXSCAN, M1, M4, M16, SCANFR
Creates temporaries for scalar expressions and constants	M16	ADDTT, COPYTP, MKDCEN, SETBUY
Creates temporaries for array expressions	E2	ADDTT, CHCKB1, COPYTP, MKDCEN, SETBUY
Creates temporaries for partially subscripted array expressions	E3	ADDTT, CHCKB4, COPYTP, MKDCEN, SETBUY
Creates special temporaries for partially subscripted arrays	EX16	ADDTT, BS2, CHCKB4, CHECKT, COPYT1, MKDCEN, STKINF, UNSTCK, Z11, SETBUY, SETMT
Checks single arguments (except structures) with parameter descriptions	M4	CHECKT, M16
Checks single structure arguments	M5	CHECKS, CSTTMP
Creates temporaries for structure expressions	M21	CSTMP2, MKDCEN, CHCKB4, SETMT, ADDTT
Creates temporaries for partially subscripted structure	Z22	BS2, ADDTT
Compare the two arguments of the POLY function and create temporaries if the arguments are not both floating and do not have the same scale and precision	POLY1, POLY2, POLY3, POLY4, POLY5	BS2
Creates special dictionary entries for generic entry labels used as arguments	M37	None

Table GP1. Phase GP Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ADDTT (GR)	Adds text to output block.
BS1	Scans input text.
BS2	Scans input text.
BS4	Examines argument lists for expressions.
BS10	End-of-program routine.
BS33	Tests for constant argument.
CHCKB1 (GR)	Compares the bounds of argument and parameter arrays, and creates new dimension tables for temporary arrays.
CHCKB2 (GR)	Compares the bounds of argument and parameter arrays where the argument is partially subscripted, and creates new dimension tables for temporary arrays.
CHCKB3 (GR)	Creates a new dimension table from a parameter description.
CHCKB4 (GR)	Creates new dimension tables for partially subscripted array and structures.
CHCKS1 (GR)	Compares the structuring of argument and parameter structures.
CHECKB (GR)	Compares the bounds of argument and parameter arrays.
CHECKS (GR)	Compares structuring and data types of argument and parameter structures.
CHECKT (GR)	Compares data types of arguments and parameters.
COPYTP (GR)	Creates a temporary dictionary entry from a parameter description.
COPYT1 (GR)	Creates a temporary dictionary entry for a partially subscripted array from a parameter description.
CSTTMP/CSTMP2 (GQ)	Create temporary structure dictionary entries.
EXSCAN (GQ)	Scans expressions for arrays and structures.
EX16 (GQ)	Creates temporary arrays for partially subscripted array arguments.
EX36 (GQ)	Creates a chameleon dictionary entry.
E2 (GQ)	Creates temporaries for array expressions.
E3 (GQ)	Creates temporaries for partially subscripted array expressions.
MKDCEN (GQ)	Makes dictionary entries.
M1 (GQ)	Examines argument expressions.
M2 (GQ)	Examines single arguments with parameter descriptions.
M4 (GQ)	Compares single arguments with parameter descriptions.
M5 (GQ)	Examines structure arguments.
M6 (GQ)	Tests for structure parameter.
M10 (GQ)	Processes subscripted variable argument.

Table GP1. Phase GP Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
M12 (GQ)	Creates a warning message.
M13 (GQ)	Gets BUY text.
M14 (GQ)	Processes scalar argument.
M16 (GQ)	Creates temporaries for scalar expressions and constants.
M21 (GQ)	Creates temporaries for structure expressions.
M22 (GQ)	Processes data item parameter.
M23 (GQ)	Processes label parameter.
M24 (GQ)	Creates a structure temporary.
M37 (GQ)	Creates dictionary entries for generic entry labels which are arguments.
M41 (GQ)	Error routine.
M44 (GQ)	Processes dimensioned scalar argument.
POLY1, POLY2, POLY3, POLY4, POLY5 (all in GO)	Check the arguments to the POLY function and generate code to buy temporaries, if the arguments are not both floating and do not have the same scale and precision.
SCANFR	Scans for matching parentheses.
SETBUY (GQ)	Inserts skeletons to buy temporaries in the output text.
SETMT (GR)	Sets temporary dictionary references in MTF compiler functions for array and structure bounds.
STKINF	Stacks information on encountering nested functions.
TESTC	Tests for constant argument.
UNSTCK	Unstacks information.
Z11 (GR)	Generates text to set up the dope vectors of partially subscripted array temporaries.
Z22 (GR)	Generates text to assign the structure subscripts of partially subscripted structures to temporaries, and then to set up the dope vector for the partially subscripted structure temporary.

Table GU. Phase GU Pretranslator Check List

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans statement; checks if preceding SIGNAL statement is needed	BSCAN	CALL, LIST, MOVE, SUOPQ
Scans statements; checks if following SIGNAL statement is needed	ASCAN	None
Provides a SIGNAL CHECK statement	CALL	GENTST
Searches list for checked items	SUOPQ	CALL, LIST

Table GU1. Phase GU Routine/Subroutine Directory

Routine/Subroutine	Function
ABGNDO	Sets IF-switch for THEN or ELSE clause.
AFM	Signals checked items in argument list.
ASC	Tests statement identifier and takes action if necessary.
ASCAN	Scans statements; checks if following SIGNAL statement is required.
ASCL	Examines statement dictionary entry.
ASPECL	Examines statement dictionary entry which is not a label.
ASTMT	Housekeeping for end of statement.
ATEST4	Tests for argument list.
ATEST5	Tests for THEN.
ATST3	Tests for end of statement.
BENTON	Test whether argument list contains checked item.
BPC	Processes "possible check" statement.
BSCAN	Scans statement; checks if preceding SIGNAL statement is required.
BSTMT	Tests whether SIGNAL statement may be needed after statement output.
BTEST3	Tests for end of statement.
BTEST4	Tests for argument list.
BVARNO	Tests for END statement.
CALL (GV)	Outputs SIGNAL statement for checked item.
CALLBA (GV)	Tests whether SIGNAL precedes or follows statement responsible.
CALLEX (GV)	Exit from subroutine CALL.
CALLIF (GV)	Tests whether DO statement must be output.
CALSTM (GV)	Re-outputs overwritten statement after DO statement.
CALSYM (GV)	Outputs SIGNAL statement.
GENTST	Checks space in output text block.
LIST (GV)	Updates and searches list of currently checked items.
MOVE	Moves text from source to output.
SUOPQ (GV)	Searches list for checked items.

Table HF. Phase HF Pretranslator Structure Assignment

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for structure assignment statements, regions of nested statements, output list expressions, and structure references in input lists	MR	BYNAME, GENTST, LSTSCN, MOVE, NSTSCN, STRASS, STREXP, STRURE
Expands structure assignments and expressions into a set of scalar assignments or expressions corresponding to the base elements of the structure operands. Where the base elements are arrays, the corresponding component expressions or assignments are surrounded by appropriately iterating DO groups	BYNAME, STRASS, STREXP, STRURE	DVCON, GENTST, LSTSCN, MOVE, NSTSCN, SBGN
Scans regions of nested statements for structure assignments	NSTSCN	MOVE, NSTSCN, STRASS
Adds text to the output string	MOVE	GENTST
Determines space availability in an output text block	GENTST	MOVE
Scans function argument and subscript lists	LSTSCN	MOVE, NSTSCN
Constructs DO statements and checks bound equivalence	DVCON	GENTST
Constructs subscript lists for references to dimensioned structure base elements	SBGN	GENTST

Table HF1. Phase HF Routine/Subroutine Directory

Routine/Subroutine	Function
BYNAME (HG)	Expands BYNAME structure assignments.
BYN1 (HG)	Searches for matching BCDs down to base elements.
BYN11 (HG)	Returns to start of current output assignment statement.
BYN13 (HG)	Test for matching BCDs.
DVCON (HG)	Constructs DO statements, checks bound equivalence.
GENTST	Determines space in output text block.
LSGET	Tests for GET statement.
LSTSCN	Scans subscript arguments and subscript lists.
LS21	Tests for structure item in data specification.
LS23	Tests for data-directed data specification.
MOVE	Adds text to output string.
MR	Scans text for structure assignment statements, nested statements, output list expressions, and structure references in input lists.
MRBYN	Tests for BY NAME assignment statement.
MRTRT	Scans source text for structures.
NSTSCN	Scans regions of nested statements for structure assignments.
SADRAB (HG)	Builds up stack to show pattern of structure.
SAEND (HG)	Tests whether END statements need to be output.
SAOP (HG)	Examines dictionary reference found.
SATRT (HG)	Scans structure expression or assignment.
SAX1 (HG)	Tests whether item matches the stack pattern.
SA20 (HG)	Tests for start of structure expression.
SA32 (HG)	Outputs base element and replaces it in source text.
SA36 (HG)	Tests for BY NAME assignment statement.
SA73 (HG)	Outputs END statements.
SA79 (HG)	Resets scan pointer to start of expression/assignment.
SBGN	Constructs subscript lists for references to dimensioned structure base elements.
STRASS (HG)	Expands structure assignments into DO loops.
STREXP (HG)	Expands structure expressions.
STRURE (HG)	Expands structure references.

Table HK. Pretranslator Array Assignment

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for array and scalar assignment statements	MR	None
Scans text for nested array and scalar assignment statements	MR	NESTAT
Scans text for array expressions in I/O lists in GET and PUT statements	MR	ARRASS, LSTSCN
Expands arrays into DO loops and scalar assignments; checks dimensions and bounds	ARRASS	FRETMP, MDE, OPTST, SLGCH, SUBSKP

Table HK1. Phase HK Routine/Subroutine Directory

Routine/Subroutine	Function
AADOP (HL)	Examines leftmost operand.
AAMULA (HL)	Tests for multiple assignment.
AA3 (HL)	Checks pseudo-variables.
AETRT (HL)	Scans array expression.
ARRASS (HL)	Expands arrays into DO loops and scalar assignments; checks dimensions and bounds.
ARREXP (HL)	Generates DO loops and subscripts for array references.
ARRIN (HL)	Entry point for array expressions in input lists.
ARROUT (HL)	Entry point for array expressions in output lists.
FRETMP	Generates a SELL statement for temporaries bought in the current statement.
LSTSCN	Scans I/O lists for possible array expressions.
MDE	Makes a temporary dictionary entry.
MR	Scans text for array and scalar assignment statements, for nested array and scalar assignment statements, and for array expressions in GET and PUT statements.
MREOP	Tests for end of text.
MRTRT	Scans text.
NESTAT	Scans nested statements.
OPTST	Tests any given operand.
SLGCH	Generates and checks subscript lists.
SLMCG	Inserts subscripts in expanded array position.
SUBSKP	Skips a subscript or subscript list.

Table HP. Phase HP Pretranslator iSub Defining

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans source text for references defined by iSUB	MASCAN	MOVE
Processes references defined by iSUB	DEFSUB	GENTST, MOVE, SULIST, SUMOVE
Scans subscripts	SUMOVE (in SULIST)	None

Table HP1. Phase HP Routine/Subroutine Directory

Routine/Subroutine	Function
DEDONE	Resets pointers to scan first subscript list.
DEEND2	Creates and buys temporary.
DEFSUB	Processes references defined by iSUB.
DEGBD1	Tests for end of second subscript list.
DENEXT	Outputs first-list subscript and tests for end of list.
DENGUB	Tests whether dictionary reference is constant or integer variable.
DERCUR	Stacks parameters for recursive entry to DEFSUB.
DERETN	Returns to MASCAN or SUSCAN.
DETEMQ	Tests whether second-list subscript is simple dictionary reference.
GENTST	Checks space in output text block.
INIT	Initializes text blocks and pointers, gets scratch storage.
MASCAN	Scans source text for references defined by iSUB.
MOVE	Moves text from source to output.
SULIST	Scans subscript lists.
SUMOVE	Scans subscripts.
SUSCAN	Scans subscript.
SUSUBS	Replaces iSUB by corresponding subscript or temporary.

TRANSLATOR PHASE TABLES

Table IA. Phase IA Translator Stacker

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans source text	ESCAN	None
Compares transfer vector	EACTNC	EC00 to EC10
Stacks transfer vector	EACTNS	ES00 to ES2E
Generates triples	EGENR	EGENR2, EGENR3, ENEWBL, ENOREP, EREPL, ETRBMP

Table IA1. Phase IA Routine/Subroutine Directory

Routine/Subroutine	Function
EACTNC	Compares transfer vector.
EACTNS	Stacks transfer vector.
EC00 to EC10	Provide comparison action for each operator.
EGENR	Generates triples.
EGENR2	Generates triple for top stack operator, with blank first operand, then deletes the operator from the stack.
EGENR3	Generates triple with two blank operands.
ENEWBL	Obtains and chains new text block for output, resets output pointer.
ENOREP	Deletes top stack operator, flags new top operand as the result of the triple just generated.
EREPL	Replaces top stack operator by its prime, to indicate end of a list of function arguments or subscripts.
ESCAN	Scans source text.
ESTCAC	Places operand in stack.
ES00 to ES2E	Handle stacking of operators.
ETRBMP	Increments output point over one triple if end of text block is found.

Table IG. Phase IG Translator Pre-Generic

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for BUY aggregate argument dummies, end-of-block, and end-of-program triples	GS1	FR, BR, TRF1, GS12
Obtains next text block	GS12	None
Transfers text to output block	TRF1	None
Transfers text skeletons to output	TRF2	GS1, TRF1
Stacks and unstacks information on encountering function and function triples	FR, FRP	None
Inserts assignment statement for aggregate argument dummies	BR	GS1, TRF2

Table IG1. Phase IG Routine/Subroutine Directory

Routine/Subroutine	Function
BR	Inserts assignment statements for aggregate argument dummies.
BR1	Transfers point for IGNORE triple.
BR2	Inserts assignment into text.
BR3	Makes new dictionary entry for temporaries.
BR4	Processes second BUY.
FR, FRP	Stack and unstack information on encountering function and function' triples.
GS1	Scans text for BUY aggregate argument dummies, end-of-block, end-of-program triples.
GS12	Chains to next text block on encountering an end of block marker.
TRF1	Transfers text to the output block.
TRF2	Transfers text skeletons to the output block.

Table IK. Phase IK Translator Pre-Generic

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes phase and obtains text block storage for routine GNEOP (called by main generic phase), for translate table SCTRT used by the expression analyser and for nested function stack	ENTER	None
Moves routine GNEOP, and table SCTRT into text block storage	MOVETT	None
Loads Phase IL and transfers control to it	LOADIL	None

Table IL. Phase IL Translator Pre-Generic

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes phase, gets scratch storage and sets pointer to function table	BEGIL	None
Moves function table into scratch storage and sets pointer to nested function stack area	BASROU	None
Loads modules IM and IN and sets base for expression analyzer code	BEGIN	None
Gets text block storage for use by Phase IM. Sets pointer to it. Moves constants into scratch storage and sets pointer to first constant. Transfers control to Phase IM	GETEXT	None

Table IM. Phase IM Translator Generic

Statement or Operation Type	Main Processing Routine	Subroutines Used
Selects function for processing	GNFUNC	GNXTRP
Selects generic procedure	GNPLIG	GNDRTA, GNXTRP, GNFMID
Selects generic Library routines; determines function result	GNBIFH	GNARID, GNCBEF, GNCACI, GNCTBI, GNGNCR, GNPRSC, GNSACH, GNSAPC, GNSBAR, EXPANL, GNSAPR, GNSBRT, GNSFMS
Selects chameleon dummy and inserts it in relevant dictionary entry	GNCHAM	GNXTRP, EXPANL
Controls scan of text -- branches to processing routine	EXPANL	ARITH, LST1, SUBSPT, ASSIGN

Table IM1. Phase IM Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ARITH (IN)	Calculates type of result of arithmetic operation (except **).
ASSIGN (IN)	Returns to calling phase with result.
EXPANL (IN)	Controls scan of text -- branches to processing routine.
GNARID (IP)	Identifies argument of built-in function and converts it to valid type, if possible.
GNBIFH (IP)	Selects generic Library routine; determines function result.
GNB08 (IP)	Selects relevant family member.
GNB16 (IP)	Sets up result type of a built-in function.
GNCACI	Checks and converts a decimal integer.
GNCBEF	Standardizes argument code byte to a form for generic selection.
GNCHAM	Selects chameleon dummy and inserts it in relevant dictionary entry.
GNCTBI	Converts from decimal to binary.
GNDRTA	Analyzes dictionary type.
GNEND	Forms pointers and branches to routine GNEOP in text block storage.
GNEOB	Processes end-of-block marker.
GNEOP	End of program routine. Frees blocks and releases control.
GNFMID (IQ)	Identifies family member.
GNFUNC	Selects function for processing.
GNF04	Checks for nested function situation.
GNF027	Sets up result type of a PL/I function.
GNFM3 (IQ)	Replaces original reference in text.
GNGNCR	General conversion routine.
GNL06 (IQ)	Forms entry relating to particular invocation.
GNPLIG (IQ)	Forms table of family member descriptions.
GNPRSC (IP)	Selects highest mode, scale and precision of variable argument list.
GNSACH	Performs special argument check.
GNSAPC	Calculates scale and precision of a function result.
GNSAPR	Processes SUBSTR function and pseudo-variable arguments.
GNSBAR	Handles a subscripted argument.
GNSBRT (IP)	Examines all three arguments of SUBSTR and calculates the resulting type exactly.
GNSFMS (IP)	Replaces references to SUBSTR in text by reference to another entry giving detailed information about the arguments. Places a description of the resulting string in the text.

Table IM1. Phase IM Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
GNTRID	Scans source text.
GNXTRP	Gets next triple.
LST1 (IN)	Calculates type and length of result of string operation.
SUBSPT (IN)	Adds type of array to stack.

Table IT. Phase IT Post-Generic Processor

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans source text	PGTXSC	PGT01,PGEOB,PGEOP
Analyzes type of function detected	PGFUNC	None
Completes function handling	PGFNCP	PGNEXT
Detects 'chameleon' temporary references and deletes BUY and BUYS triples where possible	PGBUYS	PGBUY
Deletes 'chameleon' reference in an assignment triple and alters the argument triple to indicate an intermediate result	PGPASS	None
Deletes all other references to 'chameleon' temporaries where applicable	PGFNCM	PGBYAS,PGSELL

Table IT1. Phase IT Routine/Subroutine Directory

Routine/Subroutine	Function
PGASS	Deletes 'chameleon' assignments.
PGBYAS	Processes 'Buy Assignment' triples.
PGBUY	Processes BUY triples.
PGBUYS	Processes BUYS triples.
PGEOB	Deals with End of Text Block conditions.
PGEOP	Processes end of program marker.
PGFNCM	Replaces 'chameleon' reference by an intermediate result where applicable.
PGFNCP	Processes function prime marker.
PGFUNC	Analyzes function, and determines the type of processing required.
PGNEXT	Gets the next triple in source text.
PGSELL	Processes SELL triple.
PGTXSC	Scans text.
PGT01	Determines action to be taken for a significant triple.

Table IX. Phase IX Pointer and Area Checking

Statement or Operation Type	Main Processing Routine	Subroutines Used
Main scan routine	BUMP	TEST,ERASER

Table IX1. Phase IX Routine/Subroutine Directory

Routine/Subroutine	Function
BUMP	Scan routine.
TEST	Tests operands for pointer and area data types
ERASER	Processes bad statements.

Table JD. Phase JD Constant Expression Evaluator

Statement of Operation Type	Main Processing Routine	Subroutines Used
Initializes phase, gets scratch, etc.	INIT1	None
Scans text, for constant triples	SCANT	MORTXT, PREFIX, CONCAT
Handles stacking/unstacking of operands	STAKOP	UNSTAK

Table JD1. Phase JD Routine/Subroutine Directory

Routine/Subroutine	Function
CONCAT	Detects constant string operands, performs concatenation, makes new disk entry, and puts ref. in a slot for stacking.
INIT1	Gets scratch core for the stack, initializes slots and switches.
MORTXT	Gets next text block, resets pointer.
OUT	Puts out error message and aborts compilation if stack is not emptied.
PREFIX	Detects unary prefixed constant, makes new list entry and puts ref. in a slot, for stacking.
SCANT	Main scan routine.
STAKOP	Push down stack handler.
TRYFLAG	Tests if stack is full, and if so, aborts.
UNSTAK	Moves entry from the stack.
UPTXT	Updates text pointer.
WINDUP	Releases scratch core and returns control to the control phase.

AGGREGATES PHASE TABLES

Table JI. Phase JI Aggregates Structure Processor

Statement or Operation Type	Main Processing Routine	Subroutines Used
To re-order the STATIC AUTOMATIC and CONTROLLED chains and to process structures	SCANA	MAP,MAPA
To scan down the COBOL chain for COBOL-mapped structures	SCAN	MAP
To transfer items from the COBOL chain to the appropriate AUTOMATIC chain	RECHAN	None
To transfer control from IEMTJI to IEMTJM	TERMIN	None
To map COBOL structures	MAP	NXTRF1,NXTRF2
To check non-COBOL structures for constant length	MAPA	None
To find the next member of a structure	NXTRFI	None
To find the next element of a structure	NXTRF2	None

Table JI1. Routine/Subroutine Directory

Routine/Subroutine	Function
MAP	To map COBOL structures
MAPA	To check non-COBOL structures for constant length
NXTRF1	To find the next member of a structure
NXTRF2	To find the next element of a structure
RECHAN	To transfer items from the COBOL chain to the appropriate AUTOMATIC chain
SCAN	To scan down the COBOL chain for COBOL-mapped structures
SCANA	To reorder the STATIC, AUTOMATIC, and CONTROLLED chains and process structures
TERMIN	To transfer control from IEMTJI to IEMTJM

Table JK. Phase JK Aggregates Structure Processor

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans AUTOMATIC, STATIC, and CONTROLLED chains	CHNSCN	ADRDV, CHKDEF, MKDVD, MKRDV, PROCDT, PROCST, SETBRF, TERMWS
Processes DEFINED items	CHKDEF	CMPIL1, INOBJ, PROCDT, PROCST, STBASE
Processes structures (calculates offsets, multipliers, sizes, align-ments and padding; generates object code)	PROCST	CMPIL1, INOBJ, ELSIZ
Processes arrays (calculates multi-pliers and generates object code	PROCDT	CMPIL1, INOBJ, LOADCN, SP54
Calculates storage offsets for adjustable items in structures	PS25	CMPIL1
Calculates storage offsets for adjustable arrays	ALVACA	CMPIL1
Calculates storage offsets for adjustable strings	ALVACI	CMPIL1
Generates code to initialize string dope vectors for arrays of varying strings in structures	SVARY	CMPIL1, INOBJ, IPDV, VOBJC
Generates code to initialize string dope vectors for varying, non-structured arrays	VOBJC	CMPIL1, INOBJ, IPDV
Generates code to calculate the starting address of storage for overlay defined items	STBASE	CMPIL1
Adds text skeletons to the output stream	CMPIL1	None
Makes dictionary entries for dope vector descriptions	MKDVD	ELSIZ
Makes dictionary entries for record description vectors	MKRDV	MKCNST, CMPIL1
Generates code to set the address in a record description vector at object time	ADRDV	INOBJ, CMPIL1
Calculates the length and alignment of scalar data items	ELSIZ	None
Sets offsets for BASED variables	BASED	None

Table JK1. Phase JK Routine/Subroutine Directory

Routine/Subroutine	Function
ADRDV (JL)	Generates addressing code for AUTOMATIC RDVs.
ALVACA (JL)	Calculates storage offsets for adjustable arrays.
ALVACI (JL)	Calculates storage offsets for adjustable strings.
BASED	Sets offsets for BASED variables.
CHKDEF (JM)	Processes DEFINED items.
CMPIL1 (JL)	Adds text skeletons to the output stream.
ELSIZ	Determines size of storage required for structure base elements.
INOBJ (JL)	Initializes object code statements.
IPDV (JM)	Generates code to set up primary dope vectors.
LOADCN (JL)	Generates object code to load object registers with constants known at compile time.
MKDVD	Makes dictionary entries for DVDs.
MKRDV (JM)	Makes dictionary entries for RDVs.
NXTREF/NXTRF1 (JM)	Gets the next structure base reference.
PROCDT (JM)	Processes arrays.
PROCST	Processes structures.
PS25	Calculates storage offsets for adjustable items in structures.
CHNSCN (JL)	Scans AUTOMATIC, STATIC, and CONTROLLED chains.
SETBRF (JL)	Sets the reference to the current entry type 1.
SETDVS	Sets the dynamic dope vector size for non-adjustable structures.
SP54	Calculates base element multiples.
STBASE (JM)	Generates code to initialize starting address storage for overlay defined items.
SVARY (JL)	Generates code to initialize string dope vectors for arrays of varying strings in structures.
TERMWS (JL)	Terminates object code.
VOBJC (JL)	Generates code to initialize string dope vectors for varying, non-structured arrays.

Table JP. Phase JP Translator Defined Check

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans DEFINED chain; checks validity	IEMTJP	GETCLS, GETLTH, STRCMP
Checks that two structure descriptions are the same and that they may be validly overlaid	STRCMP	None

Table JP1. Phase JP Routine/Subroutine Directory

Routine/Subroutine	Function
GETCLS	Analyzes structure descriptions, and checks that all elements are of the same defining class.
GETLTH	Obtains length of string or numeric field from associated dictionary entry.
IEMJP	Controlling scan of DEFINED chain; checks validity.
JP8	Tests whether defined item is packed.
JP20	Tests whether base defined item is adjustable.
JP200	Tests whether item is a structure.
JP540	Tests whether defined item is coded arithmetic.
JP541	Compares base and defined item.
JP542	Tests whether defined item is dimensioned.
JP543	Tests whether base code is arithmetic.
STRCMP	Compares structure descriptions.

OPTIMIZER PHASE TABLES

Table KA. Phase KA Resident Control Module

Statement or Operation Type	Main Processing Routine	Subroutines Used
Handles KTAB BLDC/T operations	KAHBLD	ZTXTAB, KAHLOK, KAHULK, ZUTXTC, KAHERR
Handles KTAB DR operation	KAHMDR	KAHERR, ZTXTAB, ZALTER, KAHULK
Handles KTAB ULDR operation	KAHUDR	KAHULK
Handles KTAB DEACT operation	KAHDAC	KAHULK
Handles KTAB FREE operation	KAHFRE	KAHERR, ZALTER
Handles KTAB SCAN operation for non-text tables	KAHSCN	KAHERR, KAHLOK, ZTXTAB, KAHULK
Handles KTAB SET/SET Z operations	KAHSET	KAHERR, KAHULK
Place save area stack, DTCAs and block list table in scratch storage	KBSTUP(KB)	None

Table KA1. Phase KA Routine/Subroutine Directory

Routine/Subroutine	Function
KAHBLD	Handles KTAB BLDC/T operations
KAHDAC	Handles KTAB DEACT operation (non-text tables)
KAHERR	Produces error message and aborts
KAHFRE	Handles KTAB FREE operation
KAHLOK	Locks a table entry
KAHMDR	Handles KTAB DR operation (non-text tables)
KAHSCN	Handles KTAB SCAN operation (non-text tables)
KAHSET	Handles KTAB SET/SETZ operations (non-text tables)
KAHTXT	Handles all KTAB operations on text tables
KAHUDR	Handles KTAB ULDR operation
KAHULK	Unlocks a table entry
KBSTUP (KB)	Places save area stack, DTCAs and block list table in scratch storage

Table KC. Phase KC DO-Loop Specification Scan

Statement or Operation Type	Main Processing Routine	Subroutines Used
General text scan	NXTRP	SCAN (KA), DOLOOP, ONBLK
Sets ON mask for ON unit	ONBLK	SCAN (KA)
Initializes reordering scan after ITDO triple	DOLOOP	SCAN (KA), EXANAS, CVEND, MOVE, MOVER
Analyzes expression in loop specification	EXANAS	SCAN (KA), MOVER, RSCAN
Scans for ITDO nested in loop specification	RSCAN	SCAN (KA), MOVER
Completes reordering scan at end of loop specification	CVEND	SCAN (KA), RSCAN, MOVER, MOVET

Table KC1. Phase KC Routine/Subroutine Directory

Routine/Subroutine	Function
CVEND	Completes reordering scan at end of loop specification
DOLOOP	Initializes reordering scan after ITDO triple
EXANAS	Analyzes expression in loop specification
MOVE	Puts triple into MOVE list
MOVER	Puts triple into REORDER list
MOVET	Moves REORDER list into text
MXTRP	General text scan
ONBLK	Sets ON mask for occurrence of ON unit
RSCAN	Scans for ITDO nested in loop specification
SCAN (KA)	Scans text

Table KE. Phase KE Dictionary Scan and DO-Map Build

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initialization	KEINIT	KCDS, KESCAN, HTAB
Dictionary scan marking unsafe variables	KCDSIN	HTAB, ZDICRF, ZDRFAB, ZDABRF
Scans text and passes control to triple processing routines	KESCAN	KTAB

Table KE1. Phase KE Routine/Subroutine Directory

Routine/Subroutine	Function
KCDSIN	Dictionary scan marking unsafe variables
KECDME	Creates a DO-Map entry
KEDEND	Completes the DO-Map entry
KEERRH	Produces termination error message and aborts
KEINIT	Initialization
KELKUP	Scans list of procedures and pointers
KESCAN	Scans text calling triple processing routines
KESTCK	Makes entry in stack

Table KG. Phase KG DO-Examine Phase

Statement or Operation Type	Main Processing Routine	Subroutines Used
Main processing routine	KGMAIN	KGSCAN, KGSRGL, KGSORT, KGUSEN
Tests whether an ON-unit could be entered as a result of an interrupt occurring at the triple being considered	KGOTST	KGSRGL
Transfers control to appropriate triple routine	KGSCAN	KGERRR
Considers a variable for entry into the USE list	KGUSEL	KGUSEN

Table KG1. Phase KG Routine/Subroutine Directory

Routine/Subroutine	Function
KGDELT	Deletes non-compiler-created temporaries from USE list
KGDELU	Deletes unsafe variables from USE list
KGERRR	Produces a termination error message and aborts
KGMAIN	Main processing routine
KGNICE	Checks that a dictionary reference is for a real fixed binary scalar integer variable
KGOTST	Tests whether an ON-unit could be entered from the triple being considered
KGSCAN	Transfers control to appropriate triple routine
KGSORT	Sorts the USE list so that invariant variables appear first
KGSRGL	Makes an entry in the SUBS/REGION list
KGUSEL	Considers a variable for entry into the USE list
KGUSEN	Makes an entry in the USE list

Table KJ. Phase KJ Subscript Table Build

Statement or Operation Type	Main Processing Routine	Subroutines Used
To build the SUBS TABLE from the Subs/Region List and test the loop initial, step, and limit for use in BXLE and BXH code	KJSB	KJSRBXCH, KJSRCHKP, KJSRSOPC, KJSRTDED, ZDRFAB, KTAB

Table KJ1. Phase KJ Routine/Subroutine Directory

Routine/Subroutine	Function
KJSB	Builds SUBS TABLE from Subs/Region List and tests the loop initial, step, and limit, for use in BXLE and BXH code
KJSRBXCH	Checks that current loop is optimizable for BXLE, BXH loop control code
KJSRCHKP	Sets a series of flags stating the attributes of the expression being analyzed
KJSRSOPC	Sets a series of flags stating the attributes of a given triple operand within the context of the expression analysis
KJSRTDED	Sets the target DED in the dictionary entry for a constant to be used in BXLE/BXH code. If necessary, new data dictionary entries are created and the reference in text modified.
KJSRUSEL	Searches the given USE list for a given variable
KODECN	Tests a given dictionary entry for a REAL, FIXED, BINARY, SCALAR, INTEGER variable, or a BINARY or DECIMAL INTEGER constant
KONICE	Tests that a given variable is REAL, FIXED, BINARY, SCALAR, INTEGER
KOPRSN	Tests if a given dictionary reference is: 1. a variable of precision less than 30 bits or 2. a decimal constant of precision less than 8 digits or 3. a binary constant of less than or equal to 30 bits

Table KN. Phase KN Subscript Optimization

Statement or Operation Type	Main Processing Routine	Subroutines Used
Sets up the physical phase KN data area, KNDATA in scratch storage. KNCODE, which is a group of lower level subroutines, is also moved into scratch storage. The scratch storage area is provided by KA. A scan is made of the chain of DO-map entries, and subroutine KNCLOF is called for each to remove offsets, and prepare potentially optimizable subscripts for matching.	KNINIT	KNCLOF
Scans the subs entries of the subs-region table for the loop. It cleans up potentially optimizable code, removes offsets, accumulating the total offset in the spare operand of the appropriate subs triples, and calculates hash values for optimizable COMA's.	KNCLOF	KNCOMU, KNHASH, KNANAL, KNOPTY
Accumulates hash total and computes hash for specified triple.	KNHASH	None
Analyzes the type of triple operand and sets a return code value accordingly.	KNANAL	ZDRFAB
Analyzes the type of triple operand and sets a return code value accordingly.	KNOPTY	ZDRFAB
Converts a decimal constant to binary and multiplies it with a given binary value. An option may be specified to allow conversion only of the given decimal constant to binary.	KNCOMU	ZDRFAB

Table KN1. Phase KN Routine/Subroutine Directory

Routine/Subroutine	Function
KNANAL	Analyzes type of triple operand and sets a return code value accordingly.
KNCLOF	Scans subs entries of subs-region table for loop. Cleans up potentially optimizable code, removed offsets, and calculates hash values for optimizable COMA's.
KNCOMU	Multiplies decimal and binary values.
KNHASH	Accumulates hash total and computes hash for specified triple.
KNINIT	Sets up code and data areas in scratch storage, and scans chain of DO-map entries, calling KNCLOF to remove offsets.
KNOPTY	Analyzes type of triple.

Table KO. Phase KO Subscript Optimization (Part 1 of 5)

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initialization is performed for the phase. The next DO-map entry in processing sequence is obtained and put in scratch storage. Module KP and KQ are loaded, and the Subs/ Region Table is updated from the patch file. The iterative specification and DO-map are checked in order to amend iterative specification. A subroutine is called to form a match chain in the Subs/ Region Table, once for Transforms and Invariants and once for commoning. When end of DO-map is reached return is made to Compiler control	KOINIT	ZLOADX, RELESE, KTAB (Macro routines in KA), KNOPTM, KOBXCH, KPUPDT
The sindex number for the loop is set to zero. The routine looks at the DO-map entry and iterative specification triples and makes a patch over the ITDO and ITD' triples if BXLE/BXH is to be generated for the loop	KOBXCH	KTAB (Macro routines in KA), KOSNDX, KOPTCH
The match chain is processed and entries are made in the patch file. The patch entries contain optimized code for three types of subscript. Patches are also created for the BXLE/BXH code for optimized loop control	KOMTCH	ZDRFAB, ZDICRF, KTAB (Macro routines in KA), KOCVTX, KOSNDX, KOMAKC, KOPTCH, KOMCOM, KOMCHN, KOMOVE, KOSSB3, KOSSB2, KOSSB1
Creates part of patch for Transforms and Invariants. It is called from KOMTCH	KOMCOM	KOMOVE, KOPTCH
Makes an entry in the patch file from the patch build area. Options are available to move the patch data to the patch build area before making an entry in the patch file. Entries are chained together if they are to be inserted at the same point in text. A PTCH triple is placed in text at the point of insertion. The overwritten triple is placed in the patch	KOPTCH	KTAB (macro in KA), KOMOVE, KOPCOM
Moves the triple to be overwritten into the patch and moves the patch into the patch file. The symbolic reference of the patch is moved to the PTCH triple in workspace. The triple in text is then overwritten with PTCH triple	KOPCOM	KOMOVE

Table KO. Phase KO Subscript Optimization (Part 2 of 5)

Statement or Operation Type	Main Processing Routine	Subroutines Used
The triples pointed to by the text references in the Subs/Region Table elements in the current match chain are amended to refer to a value calculated in patch code. The chain is then deleted and all COMA's processed are marked in the Subs/Region Table as dealt with and optimized	KOMCHN	KTAB (macro routines in KA)
Tests the type of triple at the address given and sets a return code accordingly	KNTRTY	none
Moves an item to the next available address in the patch build area in scratch storage	KOMOVE	none
Allocates a sindex register. The sindex register counter is incremented, the sindex available counter is decremented and the symbolic register counter is incremented	KOSNDX	ZUERR, ZABORT
The first part of a subs list, consisting of the SUBS triple and the COMA's before the first matched triple, is moved to the patch build area. The SUBS is changed to SSUB and a symbolic register number is placed in the second operand. A null value is inserted in the second operand of the COMA triples. All other triples are not moved	KOSSBS	KTAB (macro routines in KA), KOSNDX, KOMOVE
Moves the last part of a subs list, consisting of the COMA's between the last matched triple and the SUB' triple, to the patch build area. The SUB' is changed to SSB' and all COMA triples have their second operand set to null value. No other triples are moved	KOSSBE	KOMOVE
Tests whether any operand in a list of triples is a reference to a control variable of the current loop	KOCVTX	KTAB (macro routines in KA), KNTRTY
The message 'Invalid input type V to optimizing phase KO' is put out	KOEROR	ZUERR, ZABORT
Moves a subscript list into the patch build area changing the SUBS/SUB' triples to SSUB/SSB'. All matched COMA expressions are copied with amendments as follows: (1) References to the control variable are replaced by references to the step. (2) All additive invariant parts of the expression are deleted. All unmatched COMA expressions are replaced by COMA - NULL	KOSSB3	KOSSBS, KNTRTY, KOMOVE, KOSSBE

Table KO. Phase KO Subscript Optimization (Part 3 of 5)

Statement or Operation Type	Main Processing Routine	Subroutines Used
A subscript list is moved to the patch build area with SUBS/SUB' changed to SSUB/SSB' triples. All matched COMA expressions are copied except the dictionary references to the control variable which are replaced by dictionary references to the 'initial' elements. All unmatched COMA expressions are replaced by COMA - NULL	KOSSB2	KNTRTY, KOMOVE, KOMAKC, KOSSBE, KOEROR, KOSSBS
A subscript list is moved into the patch build area with SUBS/SUB' triples changed to SSUB/SSB' triples. All matched COMA expressions are copied. All unmatched COMA expressions are replaced by COMA - NULL	KOSSB1	KOMOVE, KOSSBE, KOSSBS
Produces a binary constant, if it is possible, given as parameters the dictionary reference of two constants in the operands of a given triple. A dictionary entry is made for the new constant, the dictionary reference of which is an output parameter. A return code value is also given indicating whether or not such a constant has been created	KOMAKC	KOTSTO, KNTRTY, ZDICRF, KOADDC, KOSUBC, KOMLTC
A dictionary entry is obtained from the given dictionary reference and tested to see if it is for a decimal or binary integer constant. If it is, the effective precision is found and the constant is converted to binary if necessary	KOTSTO	ZDRFAB, KNCOMU, KOPREC
Calculates the effective precision of the binary fullword given	KOPREC	none
A binary fullword result is obtained by multiplying together the two input binary values	KOMLTC	none
A binary fullword result is obtained by adding together the two input binary values	KOADDC	none
A binary fullword result is obtained by the subtraction of the two input binary values	KOSUBC	none
Converts a decimal constant to binary and multiplies it with a given binary value. An option may be specified to allow conversion only of the given decimal constant to binary	KNCOMU	ZDRFAB

Table KO. Phase KO Subscript Optimization (Part 4 of 5)

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans the subscript lists of a DO-loop looking for matching COMA's or COMA - expressions which are possible candidates for transforming, moving out of the loop as invariants, or commoning	KNOPTM	KTAB (macro routines in KA), KNSECO, KNTRMV, KNCHRG, KNCMPR, KNALRG, KNOMAC, KNMKVL, KOMTCH
Forward scans the Subs Table entry (equivalent to a backwards text scan) looking for the first group of COMA's that are optimizable as indicated by a switch	KNSECO	none
Analyzes the type of triple operand and sets a return code value accordingly	KNANAL	ZDRFAB
Clears the match chain	KNOMAC	KTAB (macro routines in KA)
Match area code is compared with text. The start point, finish point, and length of matched code is passed back. Only complete COMA's or COMA - expressions are matched	KNCMPR	KTAB (macro routines in KA)
Text between specified triples is scanned. Cleaned up triples are moved into scratch storage	KNTRMV	KTAB (macro routines in KA), KNTRTY, KNANAL, ZTXTAB, KNCOMU
A list of dictionary references of all variables in operands of triples in a scratch work area is made. The list is terminated with a halfword of zeros. A flag is also set if any of the variables in the work area are unsafe	KNMKVL	KNANAL
A check is made to determine if the given region entry is an end region for commoning for the matched code in the scratch work area	KNCHRG	none
A check is made for region boundaries between specified subscripts	KNALRG	KTAB (macro routines in KA), KNCHRG
Controls the search through the patch file for SSUB triples and the subsequent processing of the restricted types of expressions found after the SSUB triples	KPUPDT	KTAB (macro routines in KA), KPSSUB
Shortened version of phase KJ. Processes those triples following a SSUB triple	KPSSUB	KTAB (macro routines in KA), ZDRFAB, KPCHKP

Table KO. Phase KO Subscript Optimization (Part 5 of 5)

Statement or Operation Type	Main Processing Routine	Subroutine Used
The USE list is searched to see if the given dictionary reference is contained in the list. A return code is set depending on the part of the USE list in which the reference is found	KPUSEL	KTAB (macro routines in KA)
Examines an operand of a triple and sets flags in a code byte giving the information required on the operand during the analysis of the expression within which it occurs	KPSOPC	ZDRFAB, KPUSEL
The operands of triples following a SSUB triple are examined to determine the type of expression under consideration	KPCHKP	KPSOPC

Table KO1. Phase KO Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
KNALRG	Checks region boundaries between specified subscripts
KNANAL	Analyzes type of triple
KNCHRG	Checks for end region for commoning
KNCMPR	Compares code in two matching areas
KNCOMU	Multiplies decimal and binary values
KNMKVL	Lists variables in scratch storage
KNOMAC	Clears the match chain
KNOPTM	Scans subscript lists of DO-loop for matching COMA's
KNOPTY	Analyzes type of triple
KNSECO	Scans Subs Table entry for optimizable group of COMA's
KNTRMV	Removes offsets, tidies up, and moves code to match area
KNTRTY	Tests triple type
KOADDC	Adds two binary values
KOBXCH	Checks DO-loop and patches over ITDO and ITD' triples
KOCVTX	Tests for reference to control variable
KOEROR	Aborts
KOINIT	Initialization for physical phase

Table K01. Phase KO Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
KOMAKC	Creates binary constant
KOMCHN	Amends triples to refer to value in patch
KOMCOM	Creates part of patch for transforms and invariants
KOMLTC	Multiplies two binary values together
KOMOVE	Moves item to next place in patch build area
KOMTCH	Processes match chain and makes an entry in patch file
KOPCOM	Overwrites triple in text with PTCH triple
KOPREC	Obtains effective precision of binary value
KOPTCH	Makes an entry in the patch file
KOSBSC	Gets next entry from Subs/Region table
KOSNDX	Allocates a sindex register
KOSSBE	Moves last part of Subs list to patch build area
KOSSBS	Moves first part of Subs list to patch build area
KOSSB1	Moves a subscript list to patch build area
KOSSB2	Moves a subscript list to patch build area
KOSSB3	Moves a subscript list to patch build area
KOSUBC	Subtracts two binary values
KOTSTO	Tests if dictionary entry is for binary or decimal constant
KPCHKP	Checks operands of triples in SSUB list
KPSOPC	Sets a code byte after examining a triple operand
KPSSUB	Processes triples following a SSUB triple in a patch
KPUPDT	Searches patch file for SSUB lists
KPUSEL	Searches USE List for given dictionary reference

Table KT. Phase KT Pseudo-Code Scan

Statement or Operation Type		Main Processing Routine	Subroutines Used
SCINIT	Initialization	LA0005	UT01,UT02
SC1	Search for triple of interest	LA0010	UT01,UT03,UT06,UT07,UT08
SC2	Move current triple then search	LA0011	UT01,UT03,UT06,UT07,UT08
SC3	Delete current triple then search	LA0012	UT01,UT03,UT06,UT07,UT08
SC4	Skip current triple - text wanted	LA0020	UT06
SC5	Skip current triple - text free	LA0021	UT06
SC6	Move current triple - text wanted	LA0025	UT03,UT06
SC7	Move current triple - text free	LA0026	UT03,UT06
SC10	Symbolic input pointer to absolute	LA0035	UT01
SC11	Skip pseudo-code - text wanted	LA0040	UT06
SC12	Skip pseudo-code - text free	LA0041	UT06
MV2	Move user pseudo-code to contiguous OP	LA0050	UT04
MV3	Move user pseudo-code to OP	LA0055	UT04
MV3A	Move user triples to OP	LA0056	UT03
DV1	Generate dope vector for based aggregate	LA0070	UT07,UT10,UT11

Table KT1. Phase KT Routine/Subroutine Directory

Routine/Subroutine	Function
DV1	Generate dope vector for based aggregate.
MV2	Move user pseudo-code to contiguous output text.
MV3	Move user pseudo-code to output.
MV3A	Move user triples to output.
SCINIT	Initialize input and output text blocks.
SC1	Searches for triple of interest to user as indicated by TRT table.
SC2	Move current triple to output then search for triple of interest to user.
SC3	Delete current triple then search for triple of interest to user.
SC4	Skip over current triple and mask input WANTED.
SC5	Skip over current triple and mark input FREE.
SC6	Move current triple to output and mark input WANTED.
SC7	Move current triple to output and mark input FREE.
SC8	Move input pseudo-code to output and mark input WANTED.
SC9	Move input pseudo-code to output and mark input FREE.
SC10	Convert symbolic input pointer to absolute.
SC11	Skip over input pseudo-code and mark input WANTED.
SC12	Skip over input pseudo-code and mark input FREE.
UT01	Get a new input text block.
UT02	Get a new output text block.
UT03	Move pseudo-code to output.
UT04	Move triples to output.
UT05	Move text to output.
UT06	Test for end of block and chain to next block if necessary.
UT07	Convert dictionary reference to absolute.
UT08	Move input pseudo-code to output.
UT10	Set adjustable bound values in a dope vector.
UT11	Convert output text references to absolute.

Table KU. Phase KU DO-loop Control and Merge Patches (Part 1 of 2)

Statement or Operation Type	Main Processing Routine	Subroutines Used
The phase KU control routine. This is highest level routine in phase	KUMAIN	MV3A(KT) + all routines in modules KU and KV except KVJUMP, KVSSUB, KVSSBP
Phase initialization	KUINIT	ZLOADW, ZUGC, SCINIT(KT)
Processing initialization performed before each return to main scan	KUSETS	none
Primary phase scan of text	KUSCN1	SC3(KT), SC1(KT)
Secondary scan of DO-loop specification elements only	KUSCN2	SC3(KT), KVERRS
ITDO triple test routine. Loops that are optimizable are detected	KUITDO	KVERRS
CV and *CV triple processing routine	KUCVAR	SC5(KT), ZDRFAB, KVERRS
Determination of type of step	KUSTEP	ZDRFAB
Fill in loop control skeleton for variable step with no sindex registers	KUSKL1	ZDICRF, MV3A(KT), MV3(KT)
Fill in loop control skeleton for variable step with sindex registers	KUSKL2	MV3A(KT), MV3(KT)
Fill in loop control skeleton for constant step	KUSKL3	MV3A(KT), MV3(KT)
Phase finish. Release scratch storage KV and patch file. Return to control	KUENDS	ZURC, RLSCTL, KVERRS
Patch triple processing routine. Each patch is located and inserted	KVPTCH	ZTXTAB, MV3A(KT), KVSSUB, KVERRS KVITDP, KVSSBP, KVCOMA, KVCOMR, KVJUMP
Process all COMR triples	KVCOMR	none
Process all COMA triples	KVCOMA	none
Process JUMP triples. Used only while processing a patch	KVJUMP	MV3(KT)
Process SSUB triple. Used only while processing a patch	KVSSUB	ZDRFAB, ZTXTRF
Process SSB' triples. Used only while processing a patch	KVSSBP	ZTXTAB
SUBS and SUBO triple processing routine	KVSUBS	ZDRFAB, ZTXTRF, MV3A(KT)
SUB' triple processing routine	KVSUBP	ZTXTAB

Table KU. Phase KU DO-loop Control and Merge Patches (Part 2 of 2)

Statement or Operation Type	Main Processing Routine	Subroutines Used
ITD' triple processing. Insert epilogue into text for optimizable loops	KVITDP	MV3(KT), MV3A(KT)
Set up phase error message number and parameters	KVERRS	ZUERR, ZABORT
Search register alias table for SSUB register	KVALAS	None

Table KU1. Phase KU Routine/Subroutine Directory

Routine/Subroutine	Function
KUCVAR	Processes CV and *CV triple in optimizable loop
KUENDS	Phase finish. Releases KV, scratch storage and patch text
KUINIT	Initializes phase KU processing
KUITDO	Detects DO-loops flagged as optimizable
KUMAIN	Phase KU control routine
KUSCN1	Primary scan for phase
KUSCN2	Scan for DO-loop specification elements
KUSETS	Processing initialization
KUSKL1	Sets up variable step sindexes available loop control code
KUSKL2	Sets up variable step no sindexes loop control code
KUSKL3	Sets up constant step loop control code
KUSTEP	Determines type of step
KVALAS	Searches register alias table for SSUB register
KVCOMA	Processes COMA triples
KVCOMR	Processes COMR triples
KVERRS	Processes phase KU errors
KVITDP	Inserts loop control epilogue
KVJUMP	Processes pseudo-code within patches
KVPTCH	Processes PTCH triples by reference to patch file
KVSSBP	Processes SSB' triples occuring within patches
KVSSUB	Processes SSUB triple occuring within patches
KVSUBP	Processes all SUB' triples
KVSUBS	Processes SUBS triples

PSEUDO-CODE PHASE TABLES

Table LB. Phase LB Pseudo-Code Initial

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for PROCEDURE, BEGIN, and ALLOCATE triples	SCAN	SCINIT, SC1, SC3, SC5 (all in KT), SFSCAN, ENDRTN, MAIN, SCAUTO, AUTO12
Scans automatic chain	SCAUTO	MAIN
Processes INITIAL attribute dictionary items	MAIN	CNSTWK, ARRENT
Processes IDV statements	AUTO12	ARRENT
Processes INITIAL arrays	ARRENT	CNSTWK

Table LB1. Phase LB Routine/Subroutine Directory

Routine/Subroutine	Function
ARRENT (LC)	Generates triples and pseudo-code for arrays declared with INITIAL.
AUTO12	Processes IDV (initial dope vector) statements.
CNSTWK	Creates initialization triples.
ENDRTN	Releases phase and scratch storage.
MAIN	Processes INITIAL attribute dictionary items.
SCAN	Scans text for PROCEDURE, BEGIN, and ALLOCATE triples.
SCAUTO	Scans AUTOMATIC chain.
SFSCAN	Scans through second file statements.

Table LD. Phase LD Pseudo-Code Initial

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans the STATIC chain for any variable with the INITIAL attribute	STATIC	ENDRTN, ARRENT, CNSTWK, LOVNAS, STRADD

Table LD1. Phase LD Routine/Subroutine Directory

Routine/Subroutine	Function
ARRENT	Processes the initial value string for arrays.
CNSTWK	Creates constant entries for initial values.
CNVERT	Converts decimal integer constants used as replication factors to fixed binary.
ENDRTN	Releases the phase and scratch storage.
GAA1	Scans array initial value string.
GAC3	Makes slot for converted constant for arrays.
LOVNAS	Calculates the equivalent length in bits or bytes of a constant for variable or adjustable length strings.
STATIC	Scans the STATIC chain.
STRADD	Addresses elements of structures.
ST0006	Locates initial value list.
ST0088	Resets initial value entry.
ST9999	Makes slot for converted constant for scalars.

Table LG. Phase LG Pseudo-Code DO Expansion

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	LG0002	SC1 (KT)
For iterative DO triples, pushes down stack and examines control variable	LG0011	PSHDWN, SC5 (KT), CVSCAN
Pushes down DO stack	LG0013	PSHDWN
For iterative DO' and DO' triples, pushes up stack and removes top entry	LG0012	EXPEVL, POPUP
For CV triples, reverts to normal scan	LG0015	EXPEVL
For TO and TO' triples, examines argument and assigns to temporary, if necessary	LG0017	EXPEVL, TESTOP
For BY and BY' triples, examines expression and determines signs of constants; assigns variables to temporary	LG0019	EXPEVL, TESTOP
For WHILE and WHILE' triples, marks loop as iterative; generates test triples	LG0021	CODE3
DO EQUALS triples, assigns expression as a temporary; generates code to control loop if end of specification	LG0024	CODE2, TESTOP
Sets up control variable text in DO stack	CVSCAN	CVCOPY, PSTYP0, PSTYP1
Generates loop control code	CODE2	CVCODE, DICENT, COMPAR, SWITCHP, LMV3AU, LMV3A5, PSTYP0, PSTYP1
Tests expression result type and assigns to temorary if not constant	TESTOP	DICCHN, LMV3A5
Moves text from DO stack to output	CVCODE	LMV3AU

Table LG1. Phase LG Routine/Subroutine Directory

Routine/Subroutine	Function
CODE2	Generates loop control code.
CODE3	Generates loop control code for WHILE.
COMPAR	Generates triples to test upper limit control expression.
CVCODE	Moves text from DO stack to output.
CVCOPY	Moves input text to DO stack.
CVSCAN	Sets up control variable text in DO block.
DICCHN	Chains dictionary entries.
DICENT	Makes a dictionary entry.
EXPEVL (LH)	Analyzes expression to determine result type.
LG0000	Initializes phase.
LG0002	Scans text.
LG0010	When EOP triple encountered, releases scratch storage and passes control to next phase.
LG0011	For iterative DO triples pushes down stack and examines control variable.
LG0012	For iterative DO' and DO' triples pushes up stack and removes top entry.
LG0013	Pushes down DO stack.
LG0015	For CV triples reverts to normal scan.
LG0017	For TO and TO' triples, examines argument and assigns to temporary if necessary.
LG0019	For BY and BY' triples, examines expression and determines sign of constants. Assigns variables to temporary.
LG0021	For WHILE and WHILE' triples, marks loop as iterative and generates text triples.
LG0022	When WHILE' triple encountered, branches to generate comparison triples.
LG0024	For DO EQUALS triples, assigns expression to a temporary: generates code to control loop if at the end of specification.
LMV3AU	Moves triples to output.
LMV3A5	Moves one triple to output.
POPUP	Removes item from DO stack.
PSHDWN	Pushes down DO stack and initializes new stack entry.
PSTYP0/PSTYP1	Test pseudo-variable argument type.
SWITCHP	Changes DO stack text markers.
TESTOP	Tests expression result type and assigns to temporary if not constant.

Table LS. Phase LS Pseudo-Code Expression Evaluation

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text and branches to processing routines; marks phase LW and releases control to next phase	LB0	ARITH, FUNCT, LZZ1, MOVEPC, SCAN (KT), STRING, SUBSPT
Calculates result type and generates pseudo-code for +, -, *, /, prefix +, prefix -, compare operators, and ADD, MULTIPLY, and DIVIDE functions	ARITH, ARITH2	ADDSTK, ASSIGN, CONVT, DICDES, EXPONT, GENRPD, GETADX, GETFR, GETGR, MOVEPC, RELSTK, SETCPX, STRING, SWOP
Calculates result type for string operators	STRING	LZZ1, MOVEPC, STALRG
Inserts symbolic register in subscript triple and stacks result	SUBSPT	ADDSTK, DICDES
Inserts workspace description in TMPD triples after function, and stacks result. Stacks arguments for ADD, MULTIPLY, and DIVIDE functions. Adds pseudo-variable markers to stack	FUNCT	ADDSTK, ARITH, DICDES, GETFR, GETGR, SCAN(KT)
Calculates result type and generates pseudo-code for ** operator. Generates calling sequences to library subroutines for complex arithmetic	EXPONT	ADDSTK, ARITH2, CONVT, GETADX MOVEPC, STALRG, SWOP
Calculates target type and generates assignment triple for conversion; sets dictionary entries for constants	CONVT	ADDSTK, ASSIGN, GETFR, MOVEPC, STALRG
Interchanges operands; optionally loads first operand	SWOP	GETADX, GETFR, GETGR
Obtains free floating or fixed arithmetic register; stores it, if necessary	GETFR, GETGR	GETADX, STALRG
Adds items to, and releases items from intermediate result stack	ADDSTK, RELSTK	None
Generates calling sequence for complex * and / operators, supervises complex arithmetic	SETCPX	EXPONT, GETADX
Inserts TMPD triples after zero operands	LZZ1	RELSTK, SCAN(KT)

Table LS1. Phase LS Routine/Subroutine Directory

Routine/Subroutine	Function
ADDSTK (LT)	Adds items to intermediate result stack.
ARITH/ARITH2 (LT)	Calculate result type and generate code for +, -, *, /, prefix +, prefix -, compare operators, and ADD, MULTIPLY, and DIVIDE functions.
ASSIGN	Generates an assignment triple and TMPD in the output text.
CONST	Sets up dictionary entry for constant operand.
CONVT	Calculates target type and generates assignment triple for conversion.
DICDES	Constructs operand description from dictionary entry.
EOP2	Marks phases wanted/not wanted and releases control.
EXPONT (LU)	Calculates result type and generates pseudo-code for ** operator, and generates calling sequence to Library subroutines for complex arithmetic.
FCTDES	Inserts workspace description in TMPD triples after function, and stacks result.
FUNCT	Inserts workspace description in TMPD triples after function, and stacks result. Stacks arguments for ADD, MULTIPLY, and DIVIDE functions. Adds pseudo-variable markers to stack.
FXC1 (LT)	Generates fixed binary pseudo-code.
GENRPD	Generates pseudo-code for packed decimal operations.
GETADX (LT)	Sets up address of pseudo-code instruction.
GETFR/GETGR (LT)	Obtain free floating or fixed arithmetic register; store it, if necessary.
LB0	Scans text and branches to processing routines.
LBE21 (LT)	Tests for operand conversions and constants.
LBFL1 (LT)	Generates floating pseudo-code.
LZZ1	Inserts TMPD triples after zero operands.
MOVEPC	Moves pseudo-code to output text.
PSI	Adds pseudo-variable marker to stack.
RELSTK (LT)	Releases items from intermediate result stack.
SETCPX (LU)	Generates calling sequence for complex * and / operators; supervises complex arithmetic.
STALRG	Generates pseudo-code to store all arithmetic registers currently in use.
STRING	Calculates result type for string operators.
SUBSPT	Inserts symbolic register in subscript triple and puts result in stack.
SWOP	Interchanges operands and optionally loads first operand.

Table LV. Phase LV Pseudo-Code String Utilities

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes module; releases control to next module	STRUT0	None
Converts data item to string; calculates string length	STRUT1	SCAN (KT), STRUT2
Produces a string dope vector description from a standard string description	STRUT2	None

Table LV1. Phase LV Routine/Subroutine Directory

Routine/Subroutine	Function
LSUT17	Tests whether string length is greater than 256, and if necessary generates fixed length calling sequence.
LSUT22	Tests whether string dope vector result is required.
LSUT26	Generates any assignment and TMPD triples.
LSUT27	Sets up assignment and TMPD triples.
STUT0	Initializes module; releases control to next module.
STRUT1	Converts data item to string type; calculates string length.
STRUT2	Produces string dope vector description from standard string description.
ZSTUT1	Transfer vector to STRUT1.
ZSTUT2	Transfer vector to STRUT2.

Table LX. Phase LX Pseudo-Code String Handling

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes phase, scans text and branches to processing routines; releases control to next phase	BEGIN	FUNPT, SCAN (KT), STROP, SUBSPT, TMPDT
Processes TMPD triples. Arithmetic type TMPDs are ignored. String TMPDs are replaced by the top item from the string stack	TMPDT	GETMPD, MOVSEL, RELSTK, SCAN (KT) SETMPD
Processes function and function argument triples. Arithmetic type functions are ignored. Dictionary entries are created for the results of string type functions. A library calling sequence is generated for the BOOL function using the mechanism for packed bit operations. The result descriptions are added to the string stack	FUNT	ADDSTK, DICDES, GETADS, GETMPD, MOVEPC, RELSTK, SCAN (KT), SETMPD, STROP
Processes subscript triples. Arithmetic type subscripts are ignored. A symbolic register or workspace offset is added to string type subscript triples and the string description is added to the string stack	SUBSPT	ADDSTK, DICDES, SBGNOR, SCAN (KT)
Processes string operations CONCAT, AND, OR, NOT and comparisons with string type operands. For simple cases, in-line pseudo-code is generated; otherwise calling sequences to the library are generated. The results are added to the string stack.	STROP	ADDSTK, DICDES, GETADS, GETADX, GETMPD, MOVEPC, MOVSEL, RELSTK, SCAN(KT), STRUT(LV), ASSIGN, GETWS4, GETWS8, SBGNER, SBGNR

Table LX1. Phase LX Routine/Subroutine Directory

Routine/Subroutine	Function
ADDSTK	Adds strings to the intermediate string result stack.
ADSTR (LY)	Constructs dope vector and string descriptions from a given descriptor which may describe either a string, or its dope vector.
ASSIGN	Generates an assignment triple and associated TMPDs in the output text.
BEGIN	Main controlling routine for phase.
DICDES	Constructs operand description from dictionary entry.
FUNPT	Processes result returned by functions.
FUNT	Processes funtion and function argument triples.
GETADS/GETADX	Construct address part of pseudo-code instruction.
GETMPD	Constructs operand description from TMPD triples.
GETWS4	Allocates 4 bytes of aligned workspace.
GETWS8	Allocates 8 bytes of aligned workspace.
LB	Terminates phase at end of program.
LIB1	Generates Library calls for string operations.
LIL2 (LY)	Generates pseudo-code for NOT operation.
LIL3 (LY)	Generates pseudo-code for concatenation operation.
LIL6 (LY)	Generates pseudo-code for comparison operation.
LIL8 (LY)	Generates pseudo-code for AND/OR operation.
L11	Generates pseudo-code to convert to string.
MOVEPC	Moves pseudo-code from buffer to output text.
MOVSEL	Moves SELL triples to output text.
MVC1/MVC2(LY)	Creates MVC instructions.
RELSTK	Removes strings from the intermediate string result stack.
SBGNER	Gets next even-odd pair of symbolic registers.
SBGNOR	Gets next symbolic register.
SBGNR	Gets next symbolic register.
SETMPD	Constructs TMPD triples from description.
STROP	Processes string operations CONCAT, AND, OR, NOT, and comparisons with string type operands.
SUBSPT	Processes subscript triples.
TMPDT	Processes TMPD triples.
T5	Sets flags for triple types.

Table MA. Phase MA Pseudo-Code Translate and Verify Functions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans source text	Phase KT (SCAN)	SC1(KT)
Function marker triple (FNC) processor	FUNC	SC2(KT), SC3(KT), SC5(KT)
Double coma triple (FNCM) processor	SDCOM	SC2(KT), SC3(KT), SC5(KT)
Function prime triple (FNC') processor	SFNPM	SC5(KT)
TRANSLATE function processor	TV10A	TV31A, TV11
Creates compile time table	TV11	TV13A, TV31A
Converts constant from internal to external form and vice versa	TV13A	None
Initializes VERIFY compile time table	TV15A	TV13A
VERIFY function floating table build	TV17A	MV3(KT)
Pseudo-code build for VERIFY function	TV18A	MV3(KT)
Floating table build for TRANSLATE	TV21A	MV3(KT)
TRANSLATE function in line code	TV22A	MV3(KT)
VERIFY function processor	TV24A	TV11, TV31A
Tests for duplicate character constant	PTTRAN	ERROR
Floating table search	TV31A	None
Library calling sequence generator	TV35A	MV3(KT), TEMPW
Updates function dictionary reference	TV38A	None
Obtains workspace	TEMPD	None

Table MA1. Phase MA Routine/Subroutine Directory

Routine/Subroutine	Function
ERROR	Produces error message
FUNC	Processes function marker triple (FNC)
PTTRAN	Tests for duplicate character constant
SDCOM	Processes double coma triple (FNCM)
SFNPM	Processes function prime triple (FNC')
TEMPD	Obtains workspace
TEMPW	Gets temporary workspace
TV10A	Processes TRANSLATE function
TV11	Creates compile time table
TV13A	Converts constant from internal to external form and vice versa
TV15A	Initializes VERIFY compile time table
TV17A	Builds VERIFY function floating table
TV18A	Builds pseudo-code for VERIFY function
TV21A	Builds floating table for TRANSLATE
TV22A	TRANSLATE function in line code
TV24A	Processes VERIFY function
TV31A	Searches for floating table
TV35A	Generates library calling sequence
TV38A	Updates function dictionary reference

Table MB. Phase MB Pseudo-Code Pseudo-Variables

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans source text	MB0001	SC1 (KT)
PSI operator; starts new entry in stack for pseudo-variable	MB0011	SWITCH
PSI' operator; completes stack entry and generates code for data list items	MB0012	SWITCH, TARGET
ASSIGN completes stack and rescans group of assignments, putting target descriptions out in correct sequence; generates code for pseudo-variables in stack	MB0013	DRFTMP, MMV3A5, MVTMPD, OUTMPD, TARGET
Multiple ASSIGN; places only target descriptors in stack	MB0014	MVTMPD
Constructs pseudo-variable stack entry	MB0020	MVTMPD
Places temporary descriptor in output	OUTMPD	MMV3A5
Gets temporary workspace for pseudo-variable, if necessary	TARGET	GETWKS

Table MB1. Phase MB Routine/Subroutine Directory

Routine/Subroutine	Function
DRFTMP	Makes temporary descriptor from a dictionary reference.
GETWKS	Obtains workspace to accommodate a variable of given type.
MB0001	Scans source text.
MB0004	Multi-switch for triples of interest.
MB0010	On reaching end-of-text marker, releases remaining block, and releases control of phase.
MB0011	PSI operator; starts new entry in stack for pseudo-variable.
MB0012	PSI' operator; completes stack entry and generates code for data list items.
MB0013	ASSIGN; completes stack and rescans group of assignments, putting target descriptions out in correct sequence, generates code for pseudo-variable in stack.
MB0014	Multiple ASSIGN; places any target descriptors in stack.
MB0020	Constructs pseudo-variable stack entry.
MB1310	Resets input pointer to start of sequence of ASSIGNS.
MB1311	Rescans ASSIGNS and associated TMPDS from stack in reverse order.
MB1316	Tests for end of stack.
MB1318	Tests for pseudo-variable TMPD.
MB1320	Generates code for pseudo-variable.
MMV3A5	Moves one triple to output.
MVTMPD	Places temporary descriptor in stack.
OUTMPD	Places temporary descriptor in output string.
SWITCH	Changes scanning table.
TARGET	Obtains temporary workspace for pseudo-variable, if necessary.

Table MD. Phase MD Pseudo-Code In-Line Functions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	Phase KT (SCAN)	None
Builds up function stack	LFARIN	None
Builds up argument stack	LFCOM	None
Moves generated code to output block	LFMOVE	MV3(KT)
Generates in-line code and library calling sequences	LFEOF2	SNAKE,ROPE

Table MD1. Phase MD Routine/Subroutine Directory

LFARI1	Continues scan for in-line functions.
LFARIN	Builds up function stack.
LFCOM	Builds up argument stack.
LFDR	Unpacks dictionary reference of argument when argument triple found.
LFEOF2	Calls subroutines to generate in-line code.
LFIGN	Removes triple from text if inside an in-line function.
LFSPEC	Branches if IGNORE triple or not an in-line function.
ROPE	Generates code for STRING function.
SNAKE	Generates code for ADDR function.

Table ME. Phase ME Pseudo-Code In-Line Functions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans and moves text	Phase KT (SCAN)	SC1,SC2,SC3,SC5,MV3
Builds up function stack	SFUNC	ZDRAOF
Constructs result TDB and branches to routines for INDEX, UNSPEC, COMPLETION, and STATUS	SFNPM	MS4,MS5,MSB,RTAA,RTAB,INDEX, ILUNSP,EVENT,ZDRAOF,STATUS
Deletes current triple	SIGN	None
Builds up argument stack	SDCOM	ZDRAOF
Inspects arguments and branches to appropriate subroutine	MSB	RTB,RTC,RTD,RTE,RTF,RTG,RTH

Table ME1. Phase ME Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
EVENT	Generates in-line code for COMPLETION function.
FINISH	Passes control to the next phase.
ILUNSP	Generates in-line code for the UNSPEC function.
INDEX	Generates in-line code for optimizable invocations of the function INDEX.
MSB	Calls subroutines to generate in-line code.
MSG	Resets current flag and continues scan.
RLCTOF	Releases module and passes to next phase.
RTAA	Generates in-line code when the result is in a register by name, and the second argument is constant.
RTAB	Generates in-line code when the result is in a register by name, and the second argument is variable.
RTB	Generates in-line code for the case when the first argument is an aligned bit string, and the second and third arguments are both constant.
RTC	Generates in-line code in the case when the first argument is a character or aligned bit string, the second argument is constant and the third variable.
RTD	Generates in-line code when the first argument is a character or aligned bit string, the second is constant and the third is not present.
RTE	Generates in-line code when the first argument is a packed bit string, and the second is constant.
RTF	Generates in-line code when the first argument is a character string, and the second and third are both variable.
RTG	Generates in-line code when the first argument is a character string, the second is variable, and the third is not present.
RTH	Generates in-line code when the first argument is a bit string, and the second is variable.
SBERR	Error routine.
SBGNER	Gets the next even register and sets the even/odd bit on.
SBGNOR	Gets the next odd register and sets the even/odd bit on.
SBGTNR	Gets the next available symbolic unassigned register.
SCAN	Scans for the next triple of interest.
SCINIT	Initializes pointers and text blocks.
SDCOM	Builds up argument stack.
SFNPM	Generates in-line code.
SFUNC	Builds up function stack.

Table ME1. Phase ME Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
SIGN	Deletes current triple.
STATUS	Generates code for STATUS function.
STRUT2	Constructs a string dope vector.
SUB1	Generates code to place the address of the first argument plus a literal offset into a symbolic register.
SUB3	Generates a ST and DROP instruction, optionally followed by a MVI instruction.
SUB4	Constructs a dictionary entry for the constant JJ, and generates an MVC instruction.
SUB5	Generates two STH instructions, followed by a DROP instruction.
SUB6	Generates an RX instruction to operate on a TDB by a register, optionally followed by an instruction to drop any register used in addressing the TDB item.
SUB7L	Generates SR, SLDL, OR instructions.
SUB7R	Generates SR, SRDL, OR and DROP instructions.
SUB9	Calculates correct values for ILEN, IOFF and Y.
ZDRAOF	Converts a dictionary reference to an absolute address.
ZURCOF	Releases scratch core.

Table MG. Phase MG Pseudo-Code In-Line Functions 1

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	PHASE KT (SCAN)	None
Builds up function stack	LFARIN	None
Builds up argument stack	LFCOM	None
Move generated code to output block.	LFMOVE	MV3 (KT)
Generates in-line code	LFEOF2	ABBFLL, ABBFLS, ABSFB, ABSFD, ALLOC2, CEILB, CEILD, CEILL, CEILS, CMPLXB, CMPLXD, CMPLXL, CNASTR, CNVINT, CONJGB, CONJGD, CONJGL, CONJGS, ERRFUN, FLOORB, FLOORD, FLOORL, FLOORS, IMAGB, IMAGFD, IMAGL, IMAGS, REALB, REALFD, REALL, REALS, SBGTNR, TRUNCB, TRUNCD, TRUNCL, TRUNCS, UNSPEC, UTTEMP

Table MG1. Phase MG Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ABBFLL	Generates in-line code for ABS function with long floating-point argument.
ABBFLS	Generates in-line code for ABS function with short floating-point argument.
ABSFB	Generates in-line code for ABS function with fixed binary argument.
ABSFD	Generates in-line code for ABS function with fixed decimal argument.
ALLOC2	Generates in-line code for ALLOCATION function.
CEILB (MH)	Generates in-line code for the CEIL function with fixed binary argument.
CEILD (MH)	Generates in-line code for the CEIL function with fixed decimal argument.
CEILL (MH)	Generates in-line code for CEIL function with long floating-point argument.
CEILS (MH)	Generates in-line code for the CEIL function with short floating-point argument.
CMPLXB	Generates in-line code for COMPLEX function with fixed binary argument.
CMPLXD	Generates in-line code for COMPLEX function with fixed decimal argument.
CMPLXL	Generates in-line code for COMPLEX function with long floating-point argument.
CNASTR	Constructs assignment triple and associated TMPDS.
CNVINT	Converts a decimal integer constant to fixed binary.
CONJGB	Generates code for the CONJG function with fixed binary arguments.
CONJGD	Generates in-line code for the CONJG function with fixed decimal arguments.
CONJGL	Generates in-line code for the CONJG function with long floating-point arguments.
CONJGS	Generates in-line code for the CONJG function with short floating-point arguments.
ERRFUN	Aborts if Phase IM discovers an error in a function.
FLOORB (MH)	Generates in-line code for the FLOOR function with fixed binary argument.
FLOORD (MH)	Generates in-line code for the FLOOR function with fixed decimal argument.
FLOORL (MH)	Generates in-line code for the FLOOR function with long floating-point argument.
FLOORS (MH)	Generates in-line code for the FLOOR function with short floating-point argument.
IMAGB	Generates in-line code for IMAG function with fixed binary argument.

Table MG1. Phase MG Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
IMAGFD	Generates in-line code for IMAG function with fixed decimal argument.
IMAGL	Generates in-line code for IMAG function with long floating-point argument.
IMAGS	Generates in-line code for IMAG function with short floating-point argument.
LFARIN	Builds up function stack.
LFARI1	Continues scan for in-line functions.
LFCOM	Builds up argument stack.
LFDR	Unpacks dictionary reference of argument when argument triple found.
LFEOF2	Calls subroutines to generate in-line code.
LFEOF3	Depending on start of argument list, branches to produce in-line code.
LFIGN	Removes triple from text if inside an in-line function.
LFMOVE	Moves generated code to output block.
LFSPEC	Branches if IGNORE triple or not an in-line function.
REALB	Generates in-line code for REAL function with fixed binary argument.
REALFD	Generate in-line code for REAL function with fixed decimal argument.
REALL	Generate in-line code for REAL function with long floating-point argument.
REALS	Generates in-line code for REAL function with short floating-point argument.
SBGTNR	Get next available symbolic register.
TRUNCB (MH)	Generates in-line code for the function TRUNC with fixed binary argument.
TRUNCD (MH)	Generates in-line code for the TRUNC function with fixed decimal argument.
TRUNCL (MH)	Generates in-line code for the TRUNC function with long floating-point arguments.
TRUNCS (MH)	Generates in-line code for the TRUNC function with short floating-point argument.
UNSPEC (MH)	Generates in-line code for the UNSPEC function.
UTTEMP	Gets a required amount of temporary work space.

Table MI. Phase MI Pseudo-Code In-Line Functions 2

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	PHASE KT (SCAN)	None
Builds up function stack	LFARIN	None
Builds up argument stack	LFCOM	None
Move generated code to output block	LFMOVE	MV3 (KT)
Generates in-line code	LFEOF2	MAXB, MAXD, MAXL, MAXS, MINB, MIND, MINL, MINS, MODB, MODD, MODL, MODS, ROUNDB, ROUNDD, ROUNDL, ROUNDS

Table MI1. Phase MI Routine/Subroutine Directory

Routine/Subroutine	Function
LFARIN	Builds up function stack.
LFCOM	Builds up argument stack.
LFEOF2	Calls subroutines to generate in-line code.
LFMOVE	Moves generated code to output block.
MAXB/MINB (MJ)	Generate code for MAX/MIN function with fixed binary arguments.
MAXD/MIND (MJ)	Generate in-line code for MAX/MIN function with fixed decimal arguments.
MAXL/MINL (MJ)	Generate in-line code for MAX/MIN function with long floating-point arguments.
MAXS/MINS (MJ)	Generate in-line code for MAX/MIN function with short floating-point arguments.
MODB (MJ)	Generates in-line code for MOD function with fixed binary arguments.
MODD (MJ)	Generates in-line code for MOD function with fixed decimal arguments.
MODL (MJ)	Generates in-line code for MOD function with long floating-point arguments.
MODS (MJ)	Generates in-line code for MOD function with short floating-point arguments.
ROUNDB	Generate in-line code for ROUND function with fixed binary argument.
ROUNDD	Generates in-line code for ROUND function with fixed decimal argument.
ROUNDL	Generate in-line code for ROUND function with long floating-point arguments.
ROUNDS	Generate in-line code for ROUND function with short floating-point arguments.

Table MK. Phase MK Pseudo-Code In-Line Functions 3

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	PHASE KT (SCAN)	None
Builds up function stack	LFARIN	None
Builds up argument stack	LFCOM	None
Move generated code to output block	LFMOVE	MV3 (KT)
Generates in-line code	LFEOF2	DIM, HBOUND, LBOUND, LENGT, SIGNFB, SIGNFD, SIGNL, SIGNS, FREBIF

Table MK1. Phase MK Routine/Subroutine Directory

Routine/Subroutine	Function
DIM	Generates code for DIM function.
FREBIF	Generates code for FREE function.
HBOUND	Generates code for HBOUND function.
LBOUND	Generates code for LBOUND function.
LENGT	Generates code for LENGTH function.
LFARIN	Builds up function stack.
LFCOM	Builds up argument stack.
LFEOF2	Calls subroutines to generate in-line code.
LFMOVE	Moves generated code to output block.
SIGNFB	Generates code for SIGN function with fixed binary argument.
SIGNFD	Generates code for SIGN function with fixed decimal argument.
SIGNL	Generates code for SIGN function with short floating point argument.
SIGNS	Generates code for SIGN function with short floating point argument.

Table ML. Phase ML Pseudo-Code Calls and Functions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	PHASE KT (SCAN)	None
Identifies argument of procedure invocation	FPFNAR	None
Selects generic built-in function	FPBIF	FPARD1
Selects PL/I generic entry name	FPGAR	FPARD2, FPARD3, GNSECO

Table ML1. Phase ML Routine/Subroutine Directory

Routine/Subroutine	Function
FPA01	Scans for next argument.
FPARD1	Obtains parameter descriptions relating to built-in function arguments.
FPARD2	Obtains successive parameter descriptions relating to the entry description of a PL/I generic procedure.
FPARD3	Obtains and stacks full parameter description of a PL/I generic procedure.
FPBIF	Selects generic built-in functions.
FPEPCO	Constructs an entry parameter.
FPFNAR	Identifies arguments of procedure invocations.
FPGAR	Selects PL/I generic entry name.
GNFM2	Replaces generic reference testing for uniqueness.
GNSECO	Makes entry in stack of parameter descriptions.

Table MM. Phase MM Pseudo-Code Calls and Functions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	PHASE KT (SCAN)	None
Scans list, counts arguments and identifies storage class	CFCALL	CFARID, CFFBIR, CFFDVS, CFMVTR, CFMVCD
Rescans list and generates calling sequence for library routine	CFCFSS	CFARHA, CFCALP, CFBIFH, CFMLBR, CFMVCD, CFNEST, UTTMPW, CFA1F1, BASED

Table MM1. Phase MM Routine/Subroutine Directory

Routine/Subroutine	Function
BASED (MO)	Generates relocation code for based variables.
BEGIN	Initializes phase.
CFALF1 (MO)	Places address of invoked routine at the head of its argument list.
CFARHA	Generates calling sequence.
CFARID (MO)	Counts arguments and sets STATIC/AUTO flag.
CFBIFH	Further built-in function identification with relevant parameter setting.
CFB04	Restores previous environment.
CFB021	Tests nature of function found.
CFB036	Restores pointer to start of invocation.
CFCALL	Scans lists, counts arguments, identifies storage class.
CFCALP	Completes calling sequence and, if necessary, generates code to initialize dope vector.
CFC03C	Tests for nested function.
CFCFSS	Rescans list and generates calling sequence for Library routine.
CFEXIT	Transfer vector after first scan.
CFFBIR	Identifies built-in functions, sets parameters for calling sequence generation.
CFFDVS (MN)	Reserves output text area for generation of code to initialize dope vector when a function returns a string.
CFL06	Generates code to set up result dope vector.
CFL043	Generates code to place result address in argument list.
CFMLBR (MN)	Generates code to move a skeleton parameter list which is greater than 256 bytes.
CFMVCD	Generates pseudo-code into the output text block.
CFMVTR	Generates triple into the output text block.
CFNEST	Handles a nested situation.
CFY007	Sets parameters to produce special calling sequences.
UTTMPW (MN)	Allocates one word of workspace.

Table MP. Phase MP Pseudo-Code BUY Reorder

Statement or Operation Type	Main Processing Routine	Subroutines Used
Main scan routine for phase	MP1	SCAN(KT), ZDRFAB, ZTXTRF, ZUERR

Table MP1. Phase MP Routine/Subroutine Directory

Routine/Subroutine	Function
UT05	Adds SELL dictionary reference to SELL list if not already there.
MP1	Main controlling routine for rearranging BUY and SELL statements involved in obtaining VDAs for adjustable length string temporaries.
MP3	Processes EOP triple. Releases control of phase.
MP4	Processes BUYS triple.
MP4A	Processes BUYX triple.
MP8	Continues text scan if not string or arithmetic data, or not structure.
MP23	Continues scan of text.
MP26	Processes BUYS triple.
MP27	Processes BUY ASSIGN triple.
MP28	Processes BUY triple.
MP29	Processes SUBSCRIPT triple.
MP30	Processes ASSIGN triple.
MP31	Accesses top stack entry.
MP86	Tests triple for BUYX, and processes.
MP87	Scans for BUYS, BUY, and SELL triples.
MP5	Processes SELL triple.
SCAN(KT)	General scan routine.
ZDRFAB	Converts dictionary reference to absolute address.
ZTXTRF	Changes absolute address to a text reference.
ZUERR	Makes error message entries.

Table MS. Phase MS Pseudo-Code Subscripts

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	SBSCAN	None
Calculates element offset	SBSTIH	SBASS, SBCOBI, SBGNOR, SBMVCD, SBNEST, SBSUBP, SBSUDV, SBXOP, UTTEMP, SBOPT, SBFSUB, UTTMPH
Checks subscript range	SBSBRN	None

Table MS1. Phase MS Routine/Subroutine Directory

Routine/Subroutine	Function
SBASS	Updates scan pointer over an assignment.
SBCOBI (MT)	Converts subscript to binary integer.
SBCOMR (MT)	Handles COMR triple
SBERR (MT)	Puts error message into dictionary.
SBFSUB	Tests the FIRST flag setting if it is not already set, and exits unless FIRST was unset on entry
SBGNOR (MT)	Allocates an odd symbolic register.
SBMVCD (MT)	Generates pseudo-code and moves it into output text block.
SBNEST (MT)	Handles nested subscript situation.
SBOFFS (MT)	Handles OFS triple
SBOPT	Calculates element offset in optimizable cases.
SBSBRN (MT)	Checks subscript range.
SBSCAN	Branches to KT for scan.
SBSTIH	Generates code to calculate element offset.
SBSUBI	Saves array name.
SBSUBP (MT)	Handles end of subscript list.
SBSUDV	Generates code to set up the dope vector of an array of adjustable strings.
SBS05	Generates code to multiply subscript by multiplier.
SBS06	Compiles code to convert to fixed binary.
SBS002	Checks for occurrence of subscript.
SBS029	Generates code to multiply subscript by 4 or 8.
SBS059	Generates multiply halfword code modifications
SBTRID	Scans for comma, subscript prime, or subscript triple.
SCAN(KT)	Controlling scan of text.
UTTEMP (MT)	Allocates workspace.
UTTMPH(MT)	Gets two bytes of storage on a halfword boundary

Table NA. Phase NA Pseudo-Code Branches, ON, Returns

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes text block	NAINIT	SCINIT (KT)
Scans text for next triple of interest to user	NASC1, NASC2, NASC3	SC1, SC2, SC3 (all in KT)
Processes STOP statements	STOP	NAUT1
Processes EXIT statements	EXIT	NAUT1
Processes IF triples	IF	NAUTD, NAUT16, NAUT21, ZSTUT1
Processes ON triples	ON	NAUTD, NAUT6, NAUT16, SC5(KT)
Produces Library call at end of each PROCEDURE or BEGIN block in source text	PROCP, BEGINP	NAUT1
Processes RETURN triples	RETURN	NAUT1
Processes function RETURN statements for one data type	NA3002	NAUTB, NAUTCA, NAUT1, NAUT12
Processes function RETURN statements for more than one data type	NA3013	NAUTA, NAUTB, NAUTCA, NAUTD, NAUTF, NAUT1, NAUT7, NAUT8, NAUT9, NAUT11, NAUT12
Processes GOTO triples	GOTO	NAUTD
Processes GOLN triples	GOLN	NAUTD
Processes GOOB triples	GOOB	NAUT5, NAUTD, NAUT16, SC5 (KT)
Processes SIGNAL triples	SIGNAL	NAUTD, NAUT6, NAUT16, NAUT8, NAUT10, NAUT21
Processes REVERT triples	REVERT	NAUTD, SC5 (KT)

Table NA1. Phase NA Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
EXIT	Processes EXIT statements.
GOOB	Processes GOOB triples.
GOTO	Processes GOTO triples.
GOLN	Processes GOTO label number (GOLN) triples.
IF	Processes IF triples.
NAINIT	Initializes text blocks.
NASC1/NASC2/NASC3	Scan text for next triple of interest to user.
NAUTA	Generates pseudo-code to test switch value at RETURN (function value) statement for more than one data type.
NAUTB	Generates assignment triple to RETURN function result.
NAUTCA	Generates assignment triple set up by NAUTB.
NAUTD	Generates indicated pseudo-code.

Table NA1. Phase NA Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
NAUTF	Generates pseudo-code to branch to EQU value.
NAUT1	Generates call to indicated library routine.
NAUT2	Moves indicated pseudo-code, deletes current triple, continues text scan.
NAUT5	Makes dictionary entry for indicated library routine.
NAUT6	Updates current symbolic register value.
NAUT7	On entry, register BR points at an entry label dictionary entry. On normal exit from the routine, register BR points at the next label dictionary entry. Abnormal exit indicates that there are no further labels on the current PROCEDURE or ENTRY statement.
NAUT8	Bump EQU* value for branch pseudo-code item.
NAUT9	Bump return switch value to be used for current entry label.
NAUT11	For current entry label, generate appropriate EQU* pseudo-code item.
NAUT12	Converts current label dictionary reference to an absolute address.
NAUT16	Converts dictionary reference of triple second operand to absolute address, loads address into register BR.
NAUT17	Makes dictionary entry for maximum negative number.
NAUT18	Makes indicated dictionary entry.
NAUT21	Generates pseudo-code to compare source bit string, making library comparison routine dictionary entry, if necessary.
NA1100	Tests for SNAP.
NA1140	Using NAUTD, generates code for ON-units.
NA3002	Processes function RETURN statements for one data type.
NA3005	Outputs assignment triple.
NA3013	Processes function RETURN statements for more than one data type.
NA8003	Generates pseudo-code for branch and mask, labels.
NA8010	Converts ID to bit-string.
NA8012	Outputs pseudo-code. Compares bit-string to zero.
ON	Processes ON triples.
PROCP/BEGINP	Produce Library call at end of each procedure in source text.
RETURN	Processes RETURN triples.
REVERT	Processes REVERT triples.
SIGNAL	Processes SIGNAL triples.
STOP	Processes STOP triples.
ZSTUT1	String utility in Phase LV to provide a dope vector for a specified string.

Table NG. Phase NG Pseudo-Code Operating System Services

Statement or Operation Type	Main Processing Routine	Subroutines Used
Processes ALLOCATE statements for based variables	ALOCAT	CALIB, FALUT1
Processes DELAY statements	DLAY	CALIB, INTEG, SCAN (KT)
Processes DISPLAY statements	DSPY	CALIB, CHAR, ENDLST, SCAN (KT), STORAD
Processes FREE statements for based variables	FREE	CALIB, FALUT1
Processes WAIT statements	WAIT	CALIB,INTEG,SCAN(KT),OPLAST

Table NG1. Phase NG Routine/Subroutine Directory

Routine/Subroutine	Function
CALIB	Generates part of calling sequence and makes dictionary entry for Library routine.
CHAR (NH)	Converts a given argument to character string.
DLAY	Processes DELAY statements.
DSPY	Processes DISPLAY statements.
DSPY3	Tests that operand is character variable.
DSPY4	Makes dictionary entry for parameter list.
DSPY10	Scans for REPLY option.
ENDLST	Completes parameter list and makes dictionary entry for it.
FALUT1	Examines argument of ALLOCATE or FREE statements to see if variable is based and forms RDV in workspace to prepare for call to the library.
INTEG (NH)	Converts a given argument to an integer.
NG0	Scans to next statement.
OPLAST	Builds up parameter list in workspace.
STORAD	Stores an address in a parameter list.

Table NJ. Phase NJ Pseudo-Code RECORD I/O (Part 1 of 3)

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initialize Phase NJ by calling in block NK and initializing SCAN utility	STRTNJ	ZLOADW (AA), SCINIT (KT), SC1 (KT)
Initializes switches and flags to indicate start of new statement. Determines RECORD-oriented I/O verb and goes to appropriate routine	NUSTAT	TXTEST
Gets next triple of interest, converts to internal code and selects the appropriate routine to process it	SCNOPT	SC3 (KT), TXTEST, SCAN01, CMPERR TXTERR, ZABORT (AA)
Processes FILE option of RECORD-oriented I/O by placing dictionary reference of FILE Declare DCB in the appropriate slot of the parameter list. The parameter list is in STATIC if file constant, WORKSPACE if file parameter	FILOPT	TXTARG, DYNMPL, LAONLY, STDROP, CMPERR, TXTERR, WRKSPC, MVPSCD, ZTXTRF (KT), SYMREG, MV3 (KT)
Establishes the record dope vector (RDV) for the triple operand and places the address in the second slot of the parameter list unless the operand of the INTO triple is A varying string, in which case it places the address of the string dope vector of the operand in the second slot in the parameter list.	INTFRM	TXTARG, CMPERR, DYNMPL, LAONLY, STDROP, LAOSM2, CRDV, TXTERR, ZABORT (AA), WRKSPC, MVPSCD, TXTRF, SYMREG, ZDRFAB (AA), CALLIB, ZDICRF (AA), REFRDV, SCALAR, PNTRDR, BSDRDV
Processes the operand of the LOCATE triple by establishing the RDV for the triple operand and placing this address in the second slot of the parameter list. It establishes the pointer qualifier of the based variable and saves this, either to be used, or to be overwritten by the operand of a SET triple. It establishes a compiler label and puts this in the third slot of the parameter list in order to tell the library where to return, so that code assigning the pointer value returned in the RDV to the saved pointer operand is avoided. It then initializes the based variables just allocated	LOCOPT	TXTARG, ZDICRF, PNTRDR, SCALAR, LOCRDV, CMPERR
Processes KEYTO option of RECORD-oriented I/O by verifying that its argument is a character string, then placing it in the appropriate parameter list slot, which may be in STATIC or WORKSPACE	KYTOPT	TXTARG, SCALAR, DYNMPL, LAONLY, STDROP, NXTMPD, ZSTUT2 (STRUT2 in LV), LAOSM2, LAOSM1, TXTERR, ZDRFAB (AA), SC5 (KT), WRKSPC, MVPSCD, MV3 (KT), SYMREG

Table NJ. Phase NJ Pseudo-Code RECORD I/O (Part 2 of 3)

Statement or Operation Type	Main Processing Routine	Subroutines Used
Processes the KEY or KEYFROM option of RECORD-oriented I/O by converting the argument to a character string if it is not already a character string and placing the result in the appropriate parameter list slot; this is either in STATIC or WORKSPACE	KEYOPT	TXTARG, SCALAR, DYNMPL, LAONLY, STDROP, NXTMPD, ZSTUT1 (STRUT1 in LV), LAOSM1, LAOSM2, TMPSEL, TXTERR
Processes the IGNORE option of RECORD-oriented I/O by first checking that the argument is a scalar and then converting to a binary fixed integer if it is not already a binary fixed integer. The address of the argument is placed in the appropriate parameter list slot in STATIC or WORKSPACE	IGNOPT	TXTARG, SCALAR, CINTEG, DYNMPL, LAONLY, STDROP, MVPSCD, WRKSPC, MVTRPL, LAOSM1, ZDRFAB (AA), CMPERR, TMPREF, NXTMPD, MV3A (KT), WRKSPC, SYMREG, MV3 (KT)
Processes the event option of RECORD-oriented I/O by checking that the argument is a scalar EVENT variable and placing its address in the appropriate parameter list slot. The parameter list is either in STATIC or WORKSPACE, depending upon the storage class of the argument.	EVTOPT	TXTARG, DYNMPL, LAONLY, STDROP, NXTMPD, TMPREF, TXTERR, WRKSPC, MVPSCD, ZTXTRF (KT), SYMREF, MV3 (KT)
Processes the pointer operand of a SET triple. If part of a READ statement, the address of the pointer variable is placed in the second slot of the parameter list. If part of a LOCATE statement, the pointer operand overwrites the pointer taken from the based variable in the LOCATE statement, to be used in the pointer assignment code produced by ENDIO.	SETOPT	TXTARG, SCALAR, NXTMPD, DYNMPL, TMPREF, STDROP, TXTERR
At end of I/O statement, places REQUEST_CODE (i.e., IODEF) in static constant chain, puts STATIC parameter list in STATIC chain. Creates external Library calling sequence for RECORD-oriented I/O statement as follows: EPRM LA 1, PARM.LIST L 15, RECORD.IO.LIBRARY.ROUT BALR 14,15 EPRM	ENDIO	ZDICRF (AA), LAONLY, LAOSM1, CALLIB, MVPSCD, ZTXTAB (AA), SELL, SC3 (KT) SYMREG, TMPREF, ZDRFAB, MVTRPL, RCBCMN

Table NJ. Phase NJ Pseudo-Code RECORD I/O (Part 3 of 3)

Statement or Operation Type	Main Processing Routine	Subroutines Used
If there is a WORKSPACE parameter list, it updates the MVC or parameter list from STATIC to WORKSPACE. It checks whether a LOCATE statement is being processed, for which it generates pseudo-code to assign the pointer value from the RDV to the pointer variable and to initialize the REFER variable of a self-defining structure. It generates an allocate triple to indicate possible initialization of TASK and/or EVENT variables, and a compiler label triple to mark the end of initialization code for the library. It generates any SELL triples accumulated throughout the statement on SELL chain. It cancels the RECORD-oriented I/O option triple codes from the SCAN TRT interest table. It gets the next triple of interest and goes to NUSTAT to process as a new statement.	ENDIO (cont'd)	
Indicates presence of NOLOCK option.	NLKOPT	None
Delete the SELL triple encountered during scan of RECORD-oriented I/O statement but puts dictionary reference in the SELL chain so that SELL triple can be regenerated at end of I/O statement	SELL routine at SVSELL or TMPSELL entry point	ZDRFAB (AA), MV3A (KT)
At end of program, releases own modules and turns control over to next requested phase.	PRGEND	RLSCTL

Table NJ1. Phase NJ Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
BSDRDV(NK)	Entry point to CRDV routine which marks it as processing a based variable in an INTO or FROM option.
CALLIB	Creates pseudo-code to call library routine; indicates call in dictionary if not previously noted.
CINTEG	Checks whether argument is a binary fixed integer.
CMPERR	Indicates compiler error and ABORT, error code in HOLD register.
CRDV (NK)	Constructs a record dope vector (RDV) entry in WORKSPACE. If the dope vector descriptor bit is on, then the routine generates a library call to generate the RDV. If the variable has static extents and is not a string, the RDV is constructed from information in the RDV dictionary entry. If the variable is a string, then the RDV is constructed from its string dope vector.
DEFER	Indicates compiler error in the case of a deferred feature not detected by earlier phase.
DELETE	Establishes DELETE code as REQUEST_CODE.
DYNMPL (NK)	Establishes a parameter list in workspace if one is not already established. Calculates workspace offset to particular slot requested. Establishes a symbolic working register. Establishes skeleton pseudo-code for LA, ST, and DROP of register into workspace offset.
ENDIO	Handles operations at end of I/O statement.
EVTOPT	Processes EVENT option. (Not implemented in second version.)
FILOPT	Processes FILE option.
IGNOPT	Processes IGNORE option.
INTFRM	Processes INTO/FROM option.
KEYOPT	Processes KEY or KEYFROM option.
LAONLY (NK)	Outputs pseudo-code for LA into symbolic work register of a dictionary reference without any offset modifiers.
LOCRDV (NK)	Entry point to CRDV routine which marks it as processing a based variable of a LOCATE statement.
KYTOPT	Processes KEYTO option.
LAOSM1 (NK)	Establishes pseudo-code for a LA instruction into a symbolic work register with the address of WORKSPACE and a literal offset which is pointed to the argument register.
LAOSM2 (NK)	Generates LA pseudo-code in which both base and offset are in registers.
MVPSCD (NK)	Puts pseudo-code assembled in pseudo-code area into output text block.
MVTRPL (NK)	Invokes SCAN utility to move generated triples into output text block.
NLKOPT	Indicates presence of NOLOCK option.

Table NJ1. Phase NJ Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
NUSTAT	Handles operations at start of new statement.
NXTMPD	Invokes SCAN utility to get next triple, which is checked to see if it is a TMPD; if not, it is an error.
PNTRDR	Establishes the seven-byte pointer information slot in a BASED variable dictionary entry.
PRGEND	Releases control to next phase at end of program.
RCBCMN	Commons the dictionary entries of request code blocks.
READ	Establishes READ code as REQUEST_CODE; establishes parameter list size.
REFRDV (NK)	Establishes the address of the RDV dictionary entry in the ARG register when given the data variable dictionary address in INDX1.
REWRIT	Establishes REWRITE code as REQUEST_CODE.
SCALAR	Confirms that dictionary code byte refers to scalar item; ascertains whether item is a constant.
SCAN01	Indicates compiler error in the case of a deferred feature not detected during Read-In.
SCNOPT	Gets next triple of interest, branches to appropriate routine.
SCRHOP	Searches options, inserts RECORD-oriented I/O option entries into SCAN TRT interest table.
SELL (NK)	Generates SELL triples for all dictionary references in the SELL chain.
STDROP (NK)	Outputs pseudo-code to ST contents of symbolic work register into parameter list slot in workspace set up by DYNMPL, and the drop of the symbolic register.
STRTNJ	Initializes phase.
SYMREG (NK)	Establishes symbolic work register.
TMPREF (NK)	Generates the appropriate LA pseudo-code to load the address of the temporary described by TMPD.
TMPSEL (NK)	Adds temporary entry to SELL chain for generation of SELL triple upon completion.
TXTARG	Processes second argument of triple. If dictionary reference, establishes absolute address in INDX1. Returns to LR if zero, i.e., TEMP, LR+4 if dictionary reference. If null, indicates compiler error.
TXTERR	Writes error message.
TXTEST	Converts function code of triple interest TRT table to internal key, and invokes PRGEND if end of program is indicated.
UNLOCK	Establishes UNLOCK code as REQUEST_CODE. (Not implemented in second version.)
WRITE	Establishes WRITE code as REQUEST_CODE.
WRKSPC (NK)	Establishes the requested workspace area, starting on fullword boundary.

Table NM. Phase NM Pseudo-Code Executable I/O

Statement or Operation Type	Main Processing Routine	Subroutines Used
Processes GET and PUT statements	GET	INSERT, STORAD, INSTFL, GENPC, GENTR, MVTRSP, ENDLST, CALIB, CHAR, INTEG, UTTMPW, SRCERR, SCAN (KT), STRUT1 (LV), STRUT2 (LV)
Processes OPEN and CLOSE statements	OPEN	INSERT, STORAD, INSTFL, GENPC GENTR, MVTRSP, ENDLST, CALIB, CHAR, INTEG, UTTMPW, SRCERR, SCAN (KT), STRUT1 (LV), STRUT2 (LV)

Table NM1. Phase NM Routine/Subroutine Directory

Routine/Subroutine	Function
CALIB (NN)	Generates part of calling sequence and makes dictionary entry for Library routine.
CHAR (NN)	Converts a given argument to character string.
ENDLST (NN)	Completes parameter list and makes dictionary entry for it.
GENPC (NN)	Moves pseudo-code to output.
GENTR (NN)	Moves generated triples to output.
GET	Processes GET and PUT statements.
GET00	Initializes switches for GET/PUT.
GET20	PAGE option.
GET85	Processes end of I/O statement.
INSERT (NN)	Inserts dictionary reference in parameter list.
INSTFL (NN)	Inserts file reference in parameter list.
INTEG (NN)	Converts a given argument to integer.
MVTRSP (NN)	Moves data and format list triples to output.
NMR1	Begins scan for triples of interest.
OPEN	Processes OPEN and CLOSE statements.
OPEN00	Initializes switches for OPEN/CLOSE.
SRCERR (NN)	Makes error dictionary entry.
STORAD (NN)	Generates pseudo-code to store symbolic register in parameter list.
UTTMPW (NN)	Obtains temporary workspace.

Table NT. Phase NT Pseudo-Code Data and Format

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes phase, obtains scratch storage	NT0000	None
Scans text	NT0003	NT0011, NT0014, NT0017, NT0021, NT0023, NT0024, SC2 (KT)
Collects remote format items and saves until end of block	NT0011	None
Associates remote format items with data list items	NT0014	NTUT10
Makes entries for Library routines required for EDIT-directed I/O and copies skeletons for phase NU into scratch storage, then releases phase	NT0017	NTUT20
Identifies type of data list item and enters the type code in a list	NT0021	None
Associates format and data list items and marks INCLUDE matrix	NT0023	NTUT10
Identifies type of format list item and enters the type code in a list	NT0024	None
Sets bits in INCLUDE matrix to represent STREAM I/O conversion requirements at execution time	NTUT10	None
Makes dictionary entry for Library Routine	NTUT20	None

Table NT1. Phase NT Routine/Subroutine Directory

Routine/Subroutine	Function
NT0000	Initializes phase, obtains scratch storage.
NT0001	Initializes phase address slots.
NT0003	Scans text.
NT0011	Collects remote format items.
NT0014	Associates remote format items with data list items.
NT0017	Makes entries for Library routines for EDIT-directed I/O.
NT0021	Identifies types of data list items.
NT0023	Associates format and data list items.
NT0024	Identifies types of format list items.
NT1700	No EDIT-directed I/O, therefore no scan pass.
NTUT10	Sets bits in INCLUDE matrix.
NTUT20	Makes dictionary entry for Library routine

Table NU. Phase NU Pseudo-Code Data and Format Lists

Statement or Operation Type	Main Processing Routine	Subroutines Used
Generate Library calling sequences for data items in DATA-directed I/O statements	NU0022	INSERT,UT24,UT11,UT23
Generate Library calling sequences for data items in LIST-directed I/O statements	NU0023	INSERT,UT11,UT25,UT14,UT23,UT09
Generate code for data items in EDIT-directed I/O statements	NU0024	UT09,14
Scan text	NU0002	SC1(KT), SC2(KT), SC3(KT)
Generate Library calling sequences for format list items	NU0029,NU0030 NU0033,NU0037, NU0050	UT15,UT18,BCDCNV,UT10

Table NU1. Phase NU Routine/Subroutine Directory

Routine/Subroutine	Function
BCDCNV (NV)	Convert decimal constant to binary.
INSERT	Add an entry to an argument list.
NU0002 (NV)	Scan text.
NU0022	Generate Library calling sequence for DATA-directed data list item.
NU0023	Generate Library calling sequence for LIST-directed data list item.
NU0024 (NV)	Generate cards for EDIT-directed data list item.
NU0029 (NV)	Generate Library call for A or B format item.
NU0030 (NV)	Generate Library call for E or F format item.
NU0033 (NV)	Generate code for R format item.
NU0037 (NV)	Generate Library call for P format item.
NU0050 (NV)	Generate Library call for X, PAGE, SKIP, LINE, C, or COLUMN format item.
UT09	Make dictionary entry for constant in EDIT or LIST list.
UT10	Convert a constant entry to one of specified type.
UT11	Generate Library calling sequence passing argument list.
UT14	Generate code for intermediate result items in EDIT and LIST data lists.
UT15	Make dictionary entry for FED or DED.
UT18 (NV)	Generate an assignment triple.
UT23	Generate Library call code.
UT24	Construct symbol table dictionary entry.
UT25	Set bit in the INCLUDE matrix.

Table OB. Phase OB Pseudo-Code Compiler Functions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for BUY, BUY ASSIGN statements and compiler function and compiler pseudo-variables (see "Second File Statements" in Section 4), and transfers to appropriate routine	ST1	SCAN (KT)
Replaces MTF compiler functions (see "Second File Statements" in Section 4) by pseudo-code move character instructions, adjusting the target field to controlled or temporary type 2 workspace where necessary	MTFR	BUFIZE, FR"STOP, SC3 (KT)
Replaces ADV compiler functions (see "Second File Statements" in Section 4) by pseudo-code instructions to load specified element of a dope vector into a register	ADVR	SC3 (KT)
Replaces SDV compiler functions (see "Second File Statements" in Section 4) by instructions to load the maximum length from a string dope vector into a register	SDVR	SC3 (KT)
Replaces compiler pseudo-variable triples and compiler assignment triples by pseudo-code instructions which store the value assigned in specified part of dope vector	ST4	BUFIZE, STACK, MV3A (KT), FRSTOP, DROPRG, USTACK, SC5 (KT)
Remove BUY, BUY ASSIGN, and SELL statements for scalar non-adjustable temporary variables from the text, and allocate storage in the pseudo-code workspace for the temporaries	ST8, ST10, ST7	SC2, SC3 (both in KT)
Generates code to drop a symbolic register, or mark a literal register not wanted	DROPRG	None
Determines whether the target dictionary reference of MTF function, or ADV or SDV pseudo-variable is controlled or a temporary type 2. If it is, the dictionary reference is replaced by the dictionary reference of the controlled or temporary type 2 workspace, with the appropriate offset, if the target is a structure base element	FRSTOP	SETDVF
Stack and unstack the information specifying the target field of compiler pseudo-variable assignment	STACK, USTACK	None
Calculates the offset of the dope vector of a structure base element from the start of the structure dope vector	SETDVF	None
Place triples from the source text in an internal buffer.	BUFIZE	SC5 (KT)

Table OB1. Phase OB Routine/Subroutine Directory

Routine/Subroutine	Function
ADVR	Replaces ADV compiler functions by pseudo-code instructions to load the specified element of a dope vector into a register.
AT7	Generates pseudo-code.
AT8	Replaces operand by workspace reference.
BUFIZE	Places triples from the source text in an internal buffer.
BY5	Tests length of string.
BY19	Processes string temporary (dope vector only).
DROPRG	Generates code to drop a symbolic register or mark a literal register not wanted.
FRSTOP	Replaces the target field of MTF function or compiler pseudo-variable by controlled workspace where necessary.
MTFR	Replaces MTF compiler functions by pseudo-code move character instructions.
SDVR	Replaces SDV compiler functions by pseudo-code instructions to load the maximum string length into an object register.
SETDVF	Calculates the offset from the start of a structure dope vector to the dope vector of a particular base element.
STACK/USTACK	Stack and unstack information specifying target field of compiler pseudo-variable assignment.
ST1	Scans text for BUY and BUY ASSIGN statements, compiler functions, and compiler pseudo-variables.
ST4, ST6	Replaces compiler pseudo-variables and compiler assignment triples by pseudo code instructions to set the assigned expression, converted if necessary in the specified part of a dope vector.
ST7,ST8,ST10	Remove BUY, BUY ASSIGN, and SELL statements for fixed scalars from the text, and allocate space for the temporary variables in the pseudo-code workspace.

Table OD. Phase OD Pseudo-Code Assignment

Statement or Operation Type	Main Processing Routine	Subroutines Used
Get block of scratch core	SCRCOR	None
Move routines, constants and tables to block	MOVTAB	None

Table OD1. Phase OD Routine/Subroutine Directory

Routine/Subroutine	Function
SCRCOR	Obtains block of scratch core.
MOVTAB	Moves routines, tables and constants into scratch core.

Table OE. Phase OE Pseudo-Code Assignment

Statement or Operation Type	Main Processing Routine	Subroutines Used
Generates pseudo-code for assignment triples	ASS00	ASC00, ASCD00, ASDROP
Generates Library calling sequences for ALLOCATE, FREE, BUY, and SELL triples	ALLOC, FREE, BUY, or SELL	CALIB

Table OE1. Phase OE Routine/Subroutine Directory

Routine/Subroutine	Function
ALLOC (OF)	Processes ALLOCATE triples.
ASC00	Inserts target types for constants.
ASCD00	Controls assignment of real and complex data.
ASDROP	Drops symbolic registers.
ASFB00	Generates code for fixed binary assignments.
ASFD00 (OF)	Generates code for fixed decimal assignments.
ASFL00	Generates code for floating-point assignments.
ASL00	Generates code for label assignments.
ASPO00	Generates code for pointer/offset assignment.
ASAR00	Generates library calling sequence for area assignment.
ASS00	Processes assignment triples.
ASS032	Tests for special assignment triple.
ASTR00 (OF)	Generates code for string and numeric field assignments.
BUY (OF)	Processes BUY triples.
CALIB (OF)	Generates Library calling sequences.
ENABLE	Enables for SIZE prefix option.
FREE (OF)	Processes FREE triples.
GENCNV	Generates convert macro instruction.
GENRX0	Generates RX instruction.
GENSS0	Generates SS instruction.
GETDES	Obtains operand description.
RMNDX	Removes index from operand.
SBGTNR	Obtains next symbolic register.
SELL (OF)	Processes SELL triples.
SPASS (OF)	Processes special assignment triples.

Table OG. Phase OG Library Calling Sequences

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans triples and takes action on their values	TRSCAN	EOBIN,TSCSNO,TSCCLB,TSCPRC,FMT001, TSCEOP,TSCEP2,JSCJMP,MOVITM,MOVOUT
Scans pseudo-code deleting IGNORE items and detecting CONVERT items	PCSCAN	CNVFND,MOVITM,MOVOUT
Examines fields of CONVERT, and determines whether the conversion is to be done in-line	IEMOH	MJG201,IEMOI,CODCAL
Generates Library calling sequence	MJG201	MJG203,MJG204,MJG298
Generates in-line code for selected conversions	IEMOI	BITODI,FDTOFB,FIBFLT,DECFLT,PICHAR
Generates in-line code for pictures containing not more than four of the insertion characters (/ , . blank), if the target field is fixed decimal, fixed binary, float decimal or float binary	IEMOL	SILCON, MOVOUT

Table OG1. Phase OG Routine/Subroutine Directory

Routine/Subroutine	Function
EOBIN	Entered when an end of input block marker is detected.
TSCSNO	Processes SN, SN2, and SL triples.
TSCCLB	Processes CL triples.
TSCPRC	Processes PROC, PROC[1], BEGIN, and BEGIN[1] triples, and sets up counts for work space requirements.
FMT001	Handles the workspace requirements for FORMAT and FORMAT LIST.
TSCEOP	Processes EOP triple.
TSCEP2	Processes EOP2 triples and terminates phase.
TSCJMP	When a JUMP triple is found the routine sets up a counting mechanism and enters PCSCAN.
MOVITM	Moves from input an item which spans blocks.
MOVOUT	Moves an item to the output block.
CNVFND	When a CONVERT is found passes control to IEMOH and outputs pseudo-code generated on return.
MJG201 (OH)	Generates pseudo-code to call the Library conversion package.
CODCAL (OH)	Given a DED generates a code byte used by the in-line conversions.
MJG203 (OH)	Generates pseudo-code to point registers at data.
MJG204 (OH)	Generates pseudo-code to call Library conversion module.
MJG298 (OH)	Sets bits in include card matrix.
BITODI (OI)	Generates in-line code for binary to decimal conversion.
FDTOFB (OI)	Generates in-line code for decimal to binary conversion.
DECFLT (OI)	Generates in-line code for decimal to float conversion.
FIBFLT (OI)	Generates in-line code for fixed binary to float conversion.
PICHAR (OI)	Generates in-line code for picture to character string conversion.
SILCON (OL)	Generates in-line code for pictures containing not more than four of the insertion characters (/ , . blank), if the target field is fixed decimal, fixed binary, float decimal or float binary conversion.

Table OM. Phase OM In-line Data Conversions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for CNVC macros	TEXTSC	None
Passes control to code generation routines	CNVCDE	PACK,UNPACK,EDIT,EDMK

Table OM1. Phase OM Routine/Subroutine Directory

Routine/Subroutine	Function
CNVCDE	Passes control to the in-line code generation routines.
EDIT	Generates conversion code, based on the ED instruction, for FIXED DEC to PICTURE conversion which includes punctuation and/or zero suppression.
EDMK	Generates conversion code, based on the EDMK instruction, for FIXED DEC to PICTURE conversion which includes a drifting sign.
PACK (ON)	Generates conversion code, based on the PACK instruction, for PICTURE to FIXED DEC conversion when the picture contains only 9's and V, and has only external sign or edit characters.
PTNGEN	Generates the editing constant or mask used by the ED or EDMK in-line instructions.
TEXTSC	Scans text for CNVC macros.
UNPACK (ON)	Generates conversion code, based on the UNPK instruction, for FIXED DEC to PICTURE containing only 9's and V, and with only external sign or edit characters.

Table OP. Phase OP Further In-line Conversions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initialize and perform test scan to search for convert macros	TEXTSC	CNVCDE
Examine convert macro and select routine to generate in-line code	CNVCDE	BNTOBT,BTTOBN,FLTOBN

Table OP1. Phase OP Routine/Subroutine Directory

Routine/Subroutine	Function
BNTOBT	Generate in-line code for conversion from fixed binary to bit string
BTTOBN	Generate in-line code for conversion from bit string to fixed binary
FLTOBN	Generate in-line code for conversion from float binary to fixed binary

Table OS. Phase OS Constant Conversions

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans constants chain for double word constants	SCAN1	POOLSC, SCN010, STPTST
Scans constants chain for single word constants	SCAN2	POOLSC, SCN010, STPTST
Scans constants chain for unaligned constants	SCAN3	CONVRT, IADENT, SCN010, STPTST
Scans through constants chain for all constants used to initialize STATIC storage	SCAN4	CONVRT, STPTST
Sets up parameter and branches to the correct conversion routine	CONVRT	ARARD, ARBTD, ARCHD, CHARD, ERROUT, IACONV, IASTRN, IHEVFA, IHEVFB, IHEVFC, IHEVFD, IHEVFE, IHEVKF, IHEVKG, IHEVPA, IHEVPB, IHEVPC, IHEVPD, IHEVPE, IHEVPF, IHEVPG, IHEVPH, UPAA, UPAB, UPBA, UPBB, VSAA, VSCA, VSDA, VSEA, ZEROPT

Table OS1. Phase OS Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ARARD	Handles the linking of routines required for any arithmetic to arithmetic conversions (corresponding Library module IHEWDMA).
ARBTD	As above for arithmetic to bit conversion (corresponding Library routines IHEWDNB).
ARCHD	Arithmetic to character (IHEWDNC).
CHARD	Character to arithmetic (IHEWDCN).
CONVRT	Sets up parameters and branches to correct conversion routine.
ERROUT	Handles the output of error messages for the conversion routines.
IACONV	Handles conversion to arithmetic type.
IADENT	Makes dictionary entry in the constant pool, generating a new constant pool block if necessary.
IASTRN	Handles conversion to string type.
IHEVFA (OT)	Converts radix long floating-point binary to packed decimal intermediate.
IHEVFB (OT)	Converts long precision floating-point number to fixed binary.
IHEVFC (OT)	Converts long floating-point number to floating-point variable.
IHEVFD (OT)	Converts fixed point binary integer with scale factor to long precision floating-point intermediate.
IHEVFE (OT)	Converts floating-point number of specified precision floating-point.
IHEVKF (OU)	Converts packed decimal intermediate to decimal fixed or floating-point numeric field with specified precision.

Table OS1. Phase OS Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
IHEVKG (OU)	Converts packed decimal intermediate to a sterling numeric field, with specified precision.
IHEVPA (OT)	Converts packed decimal intermediate to long float.
IHEVPB (OU)	Converts packed decimal intermediate to an F format item.
IHEVPC (OU)	Converts packed decimal intermediate to an E format item.
IHEVPD (OT)	Converts packed decimal intermediate to a decimal integer with specified precision and scale factor.
IHEVPE (OT)	Converts an F or E format item to packed decimal intermediate.
IHEVPF (OT)	Converts a decimal integer with specified precision and scale factor to packed decimal intermediate.
IHEVPG (OT)	Converts binary fixed or floating-point constant to long precision floating-point.
IHEVPH (OT)	Converts bit string constant with up to 31 significant bits, to floating-point with long precision.
LDCONP	Points to head of constant chain.
POOLSC	Given a converted constant in scratch storage, scans the existing pool for an identical entry. If such an entry is found, the pool offset and dictionary reference of the entry is moved into the dictionary entry for the constant.
SCAN1	Scans constants chain for double word constants.
SCAN2	Scans constants chain for single word constants.
SCAN3	Scans constants chain for unaligned constants.
SCAN4	Scans constants chain for constants used to initialize static storage.
SCN010	Controls the calling of the conversion routine CONVRT and pool scan routine POOLSC and, if required, IADENT. Also handles the case of a constant given in internal form.
STPTST	Checks for the end of the constant chain.
UPAA (UPAB) (OT)	Produces zero real (imaginary) part for CAD (corresponding Library module IHEWUPA).
UPBA (UPBB) (OT)	Produces zero real (imaginary) part for numeric field (IHEWUPB).
VSAA (OT)	Convert from bit string to bit string (IHEWVSA).
VSCA (OT)	Convert from character string to character string (IHEWVSC).
VSDA (OT)	Convert from character string to bit string (IHEWVSD).
VSEA (OT)	Convert from character string to pictured character string (IHEWVSE).
ZEROPT	Produces a zero real or imaginary part for a constant given in internal form.

STORAGE ALLOCATION PHASE TABLES

Table PA. Phase PA DSAs in STATIC Storage

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans Entry Type 1 chain for blocks eligible for STATIC DSAs	PADSA	DSASIZ,DVSIZE
Makes a dictionary entry for each STATIC DSA	DICENT	None
Sorts STATIC chain (called from PD)	SCSORT	None
Scans STATIC chain for INTERNAL arrays; calculates number of elements for those arrays needing initialization. Allocates storage for arrays and, if necessary, for secondary dope vectors	ARRSCN	None

Table PA1. Phase PA Routine/Subroutine Directory

Routine/Subroutine	Function
ARRSCN	Scans STATIC chain for INTERNAL arrays; allocates storage for arrays and secondary dope vectors (called from PH).
DICENT	Makes a dictionary entry for each STATIC DSA.
DSASIZ	Calculates size of DSA excluding Register Allocator Workspace.
DVSIZE	Scans AUTOMATIC chain for variables requiring dope vectors, and calculates size of dope vectors.
PADSA	Determines eligibility of a block for a STATIC DSA.
SCSORT	Sorts STATIC chain (called from PD).

Table PD. Phase PD Storage Allocation Static 1

Statement or Operation Type	Main Processing Routine	Subroutines Used
Reverses second file dictionary pointers	PD	NXBLCK
Sorts STATIC chain	SCSORT (in PA)	None
Allocates storage for simple, non-structured, non-external items	STATIC	SDSA1
Allocates dope vectors for all non-external items	DVALOC	None
Allocates 4-byte addressing slots; calculates control section size for all external items	TVALOC	STRCDV
Allocates storage for constants.	CONALC	None

Table PD1. Phase PD Routine/Subroutine Directory

Routine/Subroutine	Function
CONALC	Allocates storage for constants.
DVALOC	Allocates dope vectors for all non-external items.
NXBLCK	Obtains next text block.
PD	Scans text file and reverses second file pointers.
SDSA1	Allocates a 4-byte address slot for each STATIC DSA.
SCSORT	Sorts STATIC chain.
STATIC	Allocates storage for simple, non-structured, non-external items.
STRCDV	Allocates relative offsets of structure member dope vectors.
TVALOC	Allocates 4-byte addressing slots; calculates control section size for all external items.

Table PH. Phase PH Storage Allocation Static 2

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans AUTOMATIC chain; allocates dope vector	PBSCAN	AUTO4, SKDV1, SKENT3, STRSCN, TEMPDV
Scans CONTROLLED chain	CONSCN	AUTO4, SKDV1, STRSCN
Allocates storage for skeleton argument lists appearing in STATIC chain	SKARGL	None
Scans STATIC chain for INTERNAL arrays; calculates number of elements for those arrays needing initializing. Allocates storage for arrays and, if necessary, for secondary dope vectors	ARRSCN (in PA)	None
Scans STATIC chain for INTERNAL structures. Calculates number of elements in structured arrays needing initializing. Calculates size of storage for all structures and bumps location counter.	STRALO	None

Table PH1. Phase PH Routine/Subroutine Directory

Routine/Subroutine	Function
ARRSCN (in PA)	Scans STATIC chain for INTERNAL arrays; allocates storage for arrays and secondary dope vectors.
AUTEND	Tests for end of AUTOMATIC chain.
AUTO4	Calculates size of dope vectors for dynamic temporaries and CONTROLLED variables.
CONSCN	Scans CONTROLLED chain.
CSCN2	Tests for end of STATIC chain.
END513	Stores STATIC location counter and releases control.
PBSCAN	Scans AUTOMATIC chain; allocates dope vectors.
PBS1	Gets next item in chain.
SKARGL	Allocates storage for skeleton argument lists appearing in STATIC chain.
SKARG1	Allocates storage required.
SKDV1	Creates skeleton dope vector dictionary entries for non-structured variables in AUTOMATIC and CONTROLLED storage.
SKENT3	Constructs skeleton dope vector dictionary entries for function values.
STRALO	Calculates number of elements in structure arrays to be initialized; calculates size of storage for all structures.
STRSCN	Creates skeleton dope vector dictionary entries for structures in AUTOMATIC and CONTROLLED chains.
TEMPDV	Creates skeleton dope vector dictionary entry for temporary workspace.

Table PL. Phase PL Storage Allocation Symbol Table and DEDs

Statement or Operation Type	Main Processing Routine	Subroutines Used
Allocates STATIC storage for all symbol tables and DEDS	IEMPL	BCSCAN, CCSCAN, CNSCAN, SCSCAN
Scans STATIC chain for symbol and DED variables	SCSCAN	DEDAL1, STRSCN, SYMTAB
Scans CONTROLLED chain for symbol and DED variables	CCSCAN	DEDAL1, STRSCN, SYMTAB
Scans PROCEDURE block chain of ENTRY type 1 entries	BCSCAN	ACSCAN, DEDAL1
Scans AUTOMATIC chain for symbol and DED variables	ACSCAN	DEDAL1, STRSCN, SYMTAB
Scans chain of members of particular structure for symbol and DED variables	STRSCN	DEDAL1, SYMTAB
Allocates storage for symbol tables	SYMTAB	DEDAL2
Allocates storage for DEDS	DEDAL (two entry points: DEDAL1, DEDAL2)	None

Table PL1. Phase PL Routine/Subroutine Directory

Routine/Subroutine	Function
ACSCAN	Scans AUTOMATIC chain for symbol and DED variables.
BCSCAN	Scans procedure block chain of ENTRY type 1 entries.
CCSCAN	Scans controlled chain for symbol and DED variables.
CNSCAN	Scans constants chain for DED variables.
DEDAL1 (PM)	Allocates storage for DEDs.
IEMPL	Allocates STATIC storage for symbol tables and DEDs.
SCSCAN	Scans STATIC chain for symbol and DED variables.
STRSCN	Scans chain of members of particular structure for symbol and DED variables.
SYMTAB (PM)	Allocates storage for symbol tables.

Table PP. Phase PP Storage Allocation Sort of AUTOMATIC Chain

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans BEGIN-ENTRY for ENTRY type 1 entries	RA0	SETCH, SCRUB1, SORCH
Scans AUTOMATIC chain from each ENTRY type 1 entry	SETCH	EXDT, SRCH2
Adds ON conditions to first AUTO-MATIC zone	SC24	None
Adds temporaries (type 2) and inde-pendent items to first zone	SC31	None
Adds dependent items to subsequent zones	SC44	None
Determines list of dependencies from INITIAL attribute	SC39	SCNCHN, SRCH2
Determines list of dependencies from DEFINED attribute	SC40	SCNCHN, SRCH2
Determines list of dependencies for array bound expressions	SC35	EXDT, SCNCHN
Determines list of dependencies for string length expressions	SC50	SCNCHN, SRCH2
Removes independent item dictionary references upon which items in the AUTOMATIC chain depend.	SCRUB1	None

Table PP1. Phase PP Routine/Subroutine Directory

Routine/Subroutine	Function
EXDT	Scans dimensions tables for second file statements with adjustable bounds.
RA0	Scans BEGIN-ENTRY for entry type 1 entries.
RA1	Tests for end of ENTRY type 1 chain.
RA4	Creates an AUTOMATIC chain delimiter.
RA7	Tests for end of chain.
SCNCHN	Scans current AUTOMATIC chain; determines whether reference belongs to it.
SCRUBI	Removes independent item dictionary references from the stack of dictionary references upon which items in the AUTOMATIC chain depend.
SC24	Adds ON conditions to first automatic zone.
SC31	Adds temporaries (type 2) and independent items to first zone.
SC35	Determines list of dependencies for array bound expressions.
SC39	Determines list of dependencies from INITIAL attribute.
SC40	Determines list of dependencies from DEFINED attribute.
SC44	Adds dependent items to subsequent zones.
SC50	Determines list of dependencies for string length expressions.
SETCHN	Scans AUTOMATIC chain from each ENTRY type 1 entry.
SORCH	Sorts chain in order of dependencies; creates zone delimiter dictionary entries.
SRCH2	Scans second file statements for dictionary references of labels, data items, and structures, which may belong to the current AUTOMATIC chains.

Table PT. Phase PT Storage Allocation AUTOMATIC Storage

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans stacked CONTROLLED chain for largest dope vector	MYNAM	DVSIZE
Initializes ENTRY type 1 chain scan and DSA	DSALOC	MKSTAT
Allocates slots for ON conditions	DSA4	MKSTAT
Allocates storage for workspace and for DSA addressing vector	DSA10	None
Scans AUTOMATIC chain and allocates storage for dope vectors	DSA16	COPY, DVSIZE, INITDV, MKSTAT, STDVIN
Allocates BUY workspace	DSA17	None
Allocates storage for parameters	DSA19	None
Allocates storage for double precision variables	DSA25	None
Allocates storage for single precision variables	DSA29	None
Allocates storage for character strings and halfword binary	DSA38	None
Allocates storage for bit strings	DSA46	None
Allocates storage for arrays and secondary dope vectors	DSA54	COPY, INITDV, MKSTAT, SDVCDE
Allocates storage for structures	DSA68	COPY, MKSTAT
Gets VDA and initializes dope vectors for adjustable regions of AUTOMATIC chain	DSA72	COPY, INITDV, MKSTAT, STDVIN
Allocates storage for DEFINED items	DSA98	None

Table PT1. Phase PT Routine/Subroutine Directory

Routine/Subroutine	Function
CONT1	Scans controlled chain for size of longest dope vector.
COPY	Compiles code to copy skeleton dope vector into real dope vector.
DSALOC	Initializes ENTRY type 1 chain scan and DSA.
DSA4	Allocates slots for ON conditions.
DSA5	Allocates standard save area and flag bytes.
DSA10	Allocates storage and workspace for DSA addressing vector.
DSA16	Scans AUTOMATIC chain and allocates dope vectors.
DSA17	Allocates BUY workspace.
DSA19 (PU)	Allocates storage for parameters.
DSA25 (PU)	Allocates storage for double precision variables.
DSA29 (PU)	Allocates storage for single precision variables.
DSA38 (PU)	Allocates storage for character strings and halfword binary.
DSA46 (PU)	Allocates storage for bit strings.
DSA54	Allocates storage for arrays and secondary dope vectors.
DSA68	Allocates storage for structures.
DSA72	Initializes dope vectors for adjustable regions of AUTOMATIC chain.
DSA74	Stores pointer to skeleton second file statement.
DSA98	Allocates storage for DEFINED items.
DSA161	Allocates storage required for dope vectors.
DSA162	Compiles code to initialize dope vectors.
DSA952	Gets VDA for this region of AUTOMATIC chain if required.
DVSIZE (PU)	Determines size of dope vectors.
INITDV	Compiles code to initialize address slot in dope vector.
MKSTAT	Makes a second file statement.
MYNAM	Scans CONTROLLED chains.
SDVCDE (PU)	Compiles code for secondary dope vectors.
STDVIN	Initializes structure member dope vectors.

Table QF. Phase QF Storage Allocation Prologues

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text for statement labels, PROCEDURE statements, BEGIN statements, BEGIN END statements, and end-of-program marker	QF0000	QBEGEP, QBPROL, QEOP, QMOVE, QPROL, QSL
Processes statement label pseudo-code items	QSL	QMOVE
Frees text storage at end of phase; releases control	QEOP	QMOVE
Creates stereotyped prologue for a BEGIN block requiring a dynamic storage area	QBPROL	QADJAL, QFSKIP, QF0201, QMOVE
Creates stereotyped or special prologues for PROCEDURE statements, depending on conditions. Processes statement label pseudo-code items	QPPROL	QADJAL, QFSKIP, QF0201, QMOVE, QONPRL
Creates a compiler label marking the return from a BEGIN block	QBEGEP	QADJAL, QF0201, QMOVE
Creates a prologue for ON block	QONPRL	QADJAL, QFSKIP, QF0201
Assembles code to initialize DSA dope vector data areas, and to allocate variable data areas	QADJAL	QMOVE1
Skips second file statements following a block heading statement	QFSKIP	None
Obtains new buffer and chains it to the previous one	QF0201	None
Moves input text being skipped from input buffer to output buffer	QMOVE	None
Moves a second file statement, pointed at by PAR1, to the prologue being generated	QMOVE1	QMOVE

Table QF1. Phase QF Routine/Subroutine Directory

Routine/Subroutine	Function
QADJAL	Assembles code to initialize DSA dope vector, variable data areas, and to allocate variable data areas.
QBEGEP	Creates a compiler label marking the return from a BEGIN block.
QBPROL (QG)	Creates stereotyped prologue for a BEGIN block requiring a dynamic storage area.
QEOP	Frees text storage at end of phase; releases control.
QFSKIP (QG)	Skips second file statements following a PROCEDURE or BEGIN statement.
QF0000	Scans text for statement labels, PROCEDURE statements, BEGIN statements, BEGIN END statements, and end-of-program marker.
QF0201 (QG)	Moves code to output buffer; obtains new buffer if required.
QF0360	Tests for external procedure.
QF0370	Generates prologue for GET DSA.
QF0570	Generates code to copy argument and target addresses.
QF0625	Tests for entry points.
QF0860	Tests end of chain.
QF1172	Tests end of first region.
QF1194	Extracts mapping code from second file.
QF1215	Tests for storage required.
QF1511	Removes VDA accumulator assignment code from mapping code.
QMOVE	Moves text from input buffer to output buffer.
QMOVE1	Moves second file statement to prologue being generated.
QONPRL (QH)	Creates prologue for ON block.
QPPROL (QG)	Creates stereotyped or special prologues for PROCEDURE statements, depending on conditions.
QSL	Processes statement label pseudo-code items.

Table QJ. Phase QJ Storage Allocation Dynamic Storage

Statement or Operation Type	Main Processing Routine	Subroutines Used
General scan of text for ALLOCATE, BUY and FREE statements	GS1	ALLOC, BUY, BUYP, FREE, TRF1.
Allocates items not requiring dope vector	AL20	AL15, TRF2
Generates code to move skeleton dope vector into workspace for controlled variables	MOVEDV	TRF2
Looks ahead to reverse pointers for ALLOCATE statements	REVPT	GS1, TRF1
Allocates storage for controlled string	AL28	GS1, LIBC1, LIBC2, SCANSF, TRF2
Allocate storage for controlled array	AL27	ABOUND, LIBC1, MOVEDV, PREVAL, SCANSF, TRF2
Allocates storage for controlled structure	AL29	BNDEXP, LIBC1, MOVEDV, NXTREF, NXTVAR, PREVAL, SCANSF, TRF2
Loads Library call parameter register to free allocated storage	FREE	TRF2, TRF3
Moves skeleton dope vector for bought temporary	BUYP	TRF2
Buys storage for temporary array	BY14	SCANSF, TRF2
Buys storage for temporary structure	BY13	LIBC4, NXTREF, NXTVAR, SCANSF, TRF2
Places initial value code line for controlled variables	AL15	NXTRF, SCANSF
Skips scan register over initialization statements	SKIPTX	GS1
Generates code to set a pointer to the previous allocation.	PREVAL	TRF2
Searches dimension tables for adjustable bound expressions	ABOUND	SCANSF
Generates code for temporary variables requiring only a dope vector	STMP	LIBC3, TRF2

Table QJ1. Phase QJ Routine/Subroutine Directory

Routine/Subroutine	Function
ABOUND (QK)	Searches dimension tables for adjustable bound expressions.
ALLOC (QK)	Ascertains the type of allocate statement.
AL15	Places initial value code line for controlled variables.
AL20 (QK)	Allocates items not requiring dope vector.
AL27 (QK)	Allocates storage for controlled arrays.
AL29 (QK)	Allocates storage for controlled structures.
BNDEXP	Generates or extracts code to set the adjustable bounds of structures.
BUY	Ascertains the type of buy.
BUYP	Moves skeleton dope vector for bought temporary.
BY13	Buys storage for temporary structure.
BY14	Buys storage for temporary array.
BY15	Buys storage for temporary string.
FREE (QK)	Loads Library call parameter register to free allocated storage.
GS1	General scan of text for ALLOCATE, BUY, and FREE statements.
LIBC1/LIBC2/LIBC4	Places the library calling sequence for controlled storage in sequence in the text.
MOVEDV (QK)	Generates code to move skeleton dope vector into workspace for controlled variables.
NXTREF (QK)	Obtains the next structure base element reference.
NXTVAR (QK)	Obtains the next varying array base element reference.
PREVAL (QK)	Generates code to set a pointer to the previous allocation.
REVPT	Looks ahead to reverse pointers for ALLOCATE statements.
SCANSF	Places second file statement in the line in the text.
SKIPTX	Skips scan register over initialization statements.
STMP (QK)	Generates code to buy storage for temporary variables which only require a dope vector.
TRF1	Transfers input text to output.
TRF2	Adds text skeletons to the output text.
TRF3	Adds the Library Calling sequence to the output text.

Table QU. Phase QU Alignment Processor

Statement or Operation Type	Main Processing Routine	Subroutines Used
Tests pseudo-code instructions for misaligned operands and deduces the correct alignment	ALIGNQ	ALREGQ,MVCMAK,REGENT
Generates a move character (MVC) instruction for a misaligned operand	MVCMAK	ABEOT,NEXREG,OUTEST,PSMOVE,REMOVE, SNEXT,TRANS
Skips a pseudo-code item	T3	TNEXT
Processes the load address (LA) pseudo-code instruction	TLA	TRR
Processes the library calling sequence in the pseudo-code	TLTB	ABEOT,T3
Processes the L pseudo-code instruction	TLL	ALIGNQ,ALREGQ,OUTEST,PSMOVE,REMOVE, SNEXT,TRANS,TRR
Processes pseudo-code instructions, other than L and LA, that may have misaligned operands	THT	ALIGNQ,TRRS
Examines a pseudo-code item and passes control to the appropriate processing routine	TRANS	T3,TABS,TDROP,TEOP,THT,TLA,TLIB, TLL,TRR,TSN

Table QU1. Phase QU Routine/Subroutine Directory

Routine/Subroutine	Function
ABEOT	Outputs termination error message.
ALREGQ	Tests whether or not the register is in the register table.
NEXREG	Gets a symbolic register.
OUTEST	Gets a new output text block if required.
PSMOVE	Fills current output text block and gets a new one.
REGENT	Makes an entry in the register table for a register that has been loaded with the address of a misaligned operand.
REMOVE	Copies text into the output text block.
SNEXT	Accesses next pseudo-code item in the source text.
TABS	Scans absolute code and copies it onto the output text if necessary.
TDROP	Removes dropped registers from the register table.
TEOB	At the end of a source text block, moves out the scanned text and gets the next source text block.
TEOP	At the end of the program, outputs the remaining text, and releases control.
TRR	Deletes an assigned register from the register table.
TSN	Updates the statement number slot in the communications region.

Table QX. Phase QX Print Aggregate Length Table

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scan storage chains in dictionary for aggregate entries	SCANC	ANAGG, PRNTAB
Analyze aggregate dictionary entries and print table entry	ANAGG	ANCOB,EXTENT,FINALA,FIRSTA,FORMAL, GETVO,GETSB,MAKEN,PRHED,SORTEN, VOPLUS

Table QX1. Phase QX Routine/Subroutine Directory

Routine/Subroutine	Function
ANAGG	Analyzes dictionary entries for a major structure or non-structured array.
ANCOB	Finds original major structure dictionary entry for a COBOL major structure.
EXTENT	Calculates length in bytes of a data variable, label, task, event, or area.
FINALA	Calculates address of final basic element of a major structure.
FIRSTA	Calculates address of first basic element of a major structure.
FORMAL	Calculates length of a non-structured array.
GETVO	Gets virtual origin of a dimensioned variable.
GETSB	Sets pointer to BCD in a dictionary entry.
MAKEN	Makes an entry in text block for each aggregate.
PRHED	Prints main heading and sub-heading of table.
PRNTAB	Prints Aggregate Length Table.
SCANC	Scans STATIC, AUTOMATIC and CONTROLLED chains in dictionary for aggregate entries.
SORTEN	Sorts text block entry for aggregate so that the entries are chained in collating sequence order of the aggregate identifiers.
VOPLUS	Calculates address of first or last element of major structure.

REGISTER ALLOCATION PHASE TABLES

Table RA. Phase RA Register Allocation Addressability Analysis

Statement or Operation Type	Main Processing Routine	Subroutines Used
Controls scan of source	LAA	ACT1, ACT2, ACT5, ACT8, ACT9, ACT10, ADCBUF, GETSBF
Processes RX, RS, or SI instructions	ACT3	ADTEST, DRTEST
Processes SS instructions	ACT4	ADTEST, DRTEST
Compiles code for start of PL/I Statement: 1. with label, 2. without label, 3. compiler label	ACT15, ACT14, ACT16	ADCBUF, GENFLP, UPSN
Processes PROCEDURE and BEGIN blocks	ACT6	ADCBUF
Processes END statements on PROCEDURE or BEGIN blocks	ACT7	ADCBUF
Adds text to output string	ADCBUF	GETCBF
Adds text to insertion file	ADIBUF	GETIBF
Obtains new source buffer	GETSBF	None
Obtains next output buffer	GETCBF	None
Obtains next insertion file buffer	GETIBF	None
Examines dictionary reference in source	DRTEST	ADINST, DECOMP, SETBLK
Produces recovery code when literal offset greater than 4095 is met	ADTEST	ADCBUF
Creates coded addressing instructions	ADINST	ADCBUF, ADIBUF

Table RA1. Phase RA Routine/Subroutine Directory

Routine/Subroutine	Function
ACT1	Copies non-special three-byte item to output.
ACT2	Copies non-special five-byte item to output.
ACT3	Processes RX, RS, or SI instructions.
ACT4	Processes SS instructions.
ACT5	End of block routine.
ACT6	Processes PROCEDURE and BEGIN blocks.
ACT7	Processes END statements on PROCEDURE or BEGIN blocks.
ACT8	End of source text routine.
ACT9	Action of start of common block of prologue.
ACT10	Action at end of prologue.
ACT12	Copies absolute code to output stream.
ACT13	Creates ADI instruction at prologue insertion point.
ACT14	Compiles code for start of PL/I statement with label.
ACT15	Compiles code for start of PL/I statement without label.
ACT16	Compiles code for start of PL/I statement compiler label.
ADD/ADD2	Generates store of calculated address.
ADCBUF	Adds text to output string.
ADIBUF	Adds text to insertion file.
ADINST	Creates coded addressing instructions.
ADTEST	Produces recovery code when literal offset greater than 4095 is met.
ATD	Tests whether previous offset is out of bounds.
DECOMP	Decodes dictionary reference.
DRTEST (RB)	Examines dictionary reference in source.
DTY	Scans step table and generates addressing instructions.
GENFLP	Generates code to set bits on and off in a prefix ON-slot.
GETCBF	Obtains next output buffer.
GETIBF	Obtains next insertion file buffer.
GETSBF	Obtains next source buffer.
LAA	Scans input text.
L125	Moves addressing instructions to IN-LINE.
SETBLK	Finds block number of referenced item.
UPSN	Generates code to keep the statement number slot in the DSA up to date.

Table RD. Phase RD Use Determination of all EQUs

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes text blocks for tables	LINIT	None
Scans text	LBUILD	LEQV, LBC, LBAL, LOBR, LEOB, LABS, L3BYT, L5BYT, LVARB, LSTAT, L2BYT, LEOP
Processes EQU items	LEQU	FNDIND
Processes BC items	LBC	FNDIND
Processes BAL items	LBAL	LOBR, L5BYT
Processes any other branch item	LOBR	FNDIND
Skips a 2-byte item	L2BYT	None
Skips a 3-byte item	L3BYT	None
Skips a 5-byte item	L5BYT	None
Skips a variable length item	LVARB	None
Processes a statement number item	LSTAT	None
Processes an EOB item	LEOB	None
Scans absolute code	LABS	None
Finds the indicator byte and text reference of an EQU value	FNDIND	None
Ends table build and passes control to second section	LEOP	LSCAN
Scans tables for optimizable EQUs	LSCAN	LFLAG
Flags EQUs in text as optimizable	LFLAG	None

Table RD1. Phase RD Routine/Subroutine Directory

Routine/Subroutine	Function
FNDIND	Finds indicator byte and text reference of EQU value
LABS	Scans absolute code
LBAL	Process BAL items
LBC	Processes BC items
LBUILD	Scans text
LEOB	Processes EOB items
LEOP	Ends table build and passes control to second section
LEQU	Processes EQU items
LFLAG	Flags EQUs in text as optimizable
LINIT	Initializes text blocks for tables
LOBR	Processes any other branch item
LSCAN	Scans tables for optimizable EQUs
LSTAT	Processes statement number items
LVARB	Skips variable length items
L2BYT	Skips 2-byte items
L3BYT	Skips 3-byte items
L5BYT	Skips 5-byte items

Table RF. Phase RF Register Allocation Physical Registers

Statement or Operation Type	Main Processing Routine	Subroutines Used
Controls scan of text	Z9	ADCBUF, ADIMOV, BR1, BR3, BR4, GETNXT, LBAL, LBALR, LBCTR, LEOB, LEOP, LR1, LR3, LR4, LR6, LR7, LR9, LSHIFT, OBREGS
Processes PROCEDURE or BEGIN statement	LPROC	None
Processes end of PROCEDURE or BEGIN block	LEND	None
Processes requests for registers; allocates physical registers	OBREGS	BRGUSE, LOAD1, STORE1, STORE2, REGUSE
Compiles code to store symbolic registers	STORE2	ADCBUF
Compiles code to store assigned registers	STORE1	ADCBUF
Compiles load of physical registers	LOAD1	ADCBUF
Compiles load register	LOADRG	ADCBUF
Expands coded addressing instructions	ADIMOV	ADCBUF
Adds to output buffer	ADCBUF	None

Table RF1. Phase RF Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ADCBUF	Adds to output buffer.
ADIMOV	Expands coded addressing instructions.
BRGUSE	Tabulates use of base register in look-ahead.
BR1 (RH)	Processes RX branch instructions.
BR3 (RH)	Processes BCT instructions.
BR4 (RH)	Processes RR branch instructions.
FRTEST	Scans list of free registers to make even-odd pair.
GETNXT	Obtains next block.
LAD1 (RH)	Processes AD1 (addressing) instructions.
LB (RH)	Constructs and puts out completed instruction.

Table RF1. Phase RF Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
LBAL (RH)	Processes BAL instructions.
LBALR (RH)	Processes BALR instructions.
LBCTR (RH)	Processes BCTR instructions.
LDROP (RH)	Processes DROP pseudo-instruction.
LEND (RH)	Loads end of PROCEDURE or BEGIN block.
LEOB (RH)	Processes end-of-block marker.
LEOP	Processes end-of-program marker.
LOAD1	Compiles load of physical registers.
LOADRG	Compiles load register.
LPROC (RH)	Processes PROCEDURE or BEGIN statement.
LR1 (RH)	Processes instructions in which first and second operands require loading, and the first is altered, e.g., AR.
LR3 (RH)	Processes floating-point instructions.
LR4 (RH)	Processes SS instructions.
LR6 (RH)	Processes instructions where a load of first operand is required, no operands are changed, e.g., ST.
LR7 (RH)	Processes SI instructions.
LR9 (RH)	Processes instructions in which no load of first operand is needed, and it is changed, e.g., LA.
LSHIFT (RH)	Processes shift instructions.
OB560(RG)	Tests whether all registers are available.
OB630 (RG)	Generates stores of registers if branch in or out.
OB895 (RG)	Generates code to load registers.
P9INIT (RH)	Main text scan.
OBREGS (RG)	Processes requests for registers; allocates physical registers.
REGUSE	Tabulates use of registers in look ahead.
STORE1	Compiles code to store assigned registers.
STORE2	Compiles code to store symbolic registers.
W4 (RH)	Extracts ADIs at prologue insertion point.
Z9 (RH)	Controlling scan of text.

FINAL ASSEMBLY PHASE TABLES

Table TF. Phase TF Final Assembly Pass 1

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	IL0024	None
Assigns offsets to labels	IL0019	FINEQ1, NEXTSL
Increments location counter for machine instructions	IL0014	None
Determines code for instructions which refer to labels	IL0020	FINEQ1
Initializes location counter at start of procedure	IL0010	None
Stores size of procedure and resumes containing procedure	IL0011	None

Table TF1. Phase TF Routine/Subroutine Directory

Routine/Subroutine	Function
FINEQ1	Locates label number table entries.
IL0000	Entry point from compiler control.
IL0003	Entry point to scan from initialization routine.
IL0010	Initializes location counter at start of procedure.
IL0011	Stores size of procedure and resumes containing procedure.
IL0014	Increments location counter for machine instructions.
IL0015	Processes the start of prologues.
IL0017	Releases control.
IL0019	Assigns offsets to labels.
IL0020	Determines code for instructions which refer to labels.
IL0022	Processes end-of-block pseudo-code item.
IL0024	Scans text.
NEXTSL	Determines multiple statement label entries in dictionary.

Table TJ. Phase TJ Final Assembly Optimization

Statement or Operation Type	Main Processing Routine	Subroutines Used
Controls phase	IL0000	OPTIMA
Maintains location counter for machine instructions	IL0014	None
Assigns offsets to labels	IL0019	COMRTN, FINEQ1, NEXTSL
Determines code for instructions which refer to labels	IL0020	FINEQ1
Initialize location counter at start of procedure	IL0010	None
Stores size of procedure for machine instructions	IL0011	None
Reduces number of MVC instructions	IL0027	OFFSET, OSMRTN
Determines offset from a given dictionary reference	OFFSET	None

Table TJ1. Phase TJ Routine/Subroutine Directory

Routine/Subroutine	Function
COMRTN	Determines whether further optimization is possible.
FINEQ1	Locates label number table entries.
IL0000	Controls phase.
IL0003	Entry point to scan loop from initialization.
IL0010	Initializes location counter at start of procedure.
IL0011	Stores size of procedure and resumes containing procedure.
IL0012	Processes machine instructions, etc.
IL0014	Maintains location counter for machine instructions.
IL0019	Assigns offsets to labels.
IL0020	Determines code for instructions which refer to labels.
IL0024	Gets pseudo-code item length and updates text pointer.
IL0027	Elides MVC instructions.
IL1001	Evaluates new ADCON needs. Sets location counter to zero.
IL1101	Restores content of containing procedure.
NEXTSL	Looks for equivalent statement labels.
OFFSET (TK)	Determines offset from a given dictionary reference.
OPTIMA	Scans text.
OSMRTN	Scans ahead for literal offsets.

Table TO. Phase TO Final Assembly External Symbol Dictionary

Statement or Operation Type	Main Processing Routine	Subroutines Used
Constructs first six standard ESD entries	LG401	MOVE, NAME, ERROR
Constructs entries for external procedure labels	LG001	MOVE, ERROR
Constructs PR type entries for each block and procedure	LG030	MOVE, NAME
Constructs entries for external variables and external entry names	LG050	MOVE, ERROR
Constructs entries for controlled variables and task names	LG090	MOVE, NAME, ERROR
Constructs entries for Library conversion modules	IL0200	IHEINC

Table TO1. Phase TO Routine/Subroutine Directory

Routine/Subroutine	Function
ERROR	Truncates over-length external identifier, generates error message.
LG001	Constructs entries for external procedure labels.
LG030	Constructs PR type entries for each block and procedure.
LG050	Constructs entries for external variables and external entry names.
LG055	Processes ON-conditions and external variables.
LG080	Processes external entry names.
LG085	Processes FILE constants.
LG090	Constructs entries for controlled variables and task names.
LG093	Inserts name in ESD entry for CONTROLLED.
LG401	Constructs first six standard ESD entries.
MOVE	Moves ESD entries to card buffers, and puts out buffer when full.
NAME	Generates names for pseudo-registers.
IHEINC (TQ)	Constructs a string of Library module names.

Table TT. Phase TT Final Assembly Pass 2

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans text	IL0002	None
Generates text for RR instructions	IL0012	GENTXT
Generates Text for RX non-branch instructions LM, STM, and SI Types	IL0013	EOBRTN, GENTXT, OFFSET
Generates text for shift instructions	IL0027	GENTXT
Generates Text for SS instructions	IL0014	EOBRTN, GENTXT, OFFSET
Sets up trace information and numbers compiler labels	IL0019	GENTXT
Generates text for branch and load address instructions	IL0020	FINEQ1, GENTXT, OFFSET
Initializes location counter at start of procedure	IL0010	PUNCHT
Resumes containing procedure at end of procedure	IL0011	PUNCHT
Moves Text into card image	GENTXT	PUNCHT
Punches cards ensuring that RLD cards follow related TXT card	PUNCHT	CARDOU
Generates text for compiler subroutine	INCLUD	GENTXT

Table TT1. Phase TT Routine/Subroutine Directory

Routine/Subroutine	Function
CARDOU	Directs card image to load file or punch file.
EOBRTN	Chains to next input text block.
FINEQ1	Locates label number table entries.
GENTXT	Moves text into card image.
IL0002	Scans text.
IL0003	Entry point to scan from initialization routines.
IL0010	Initializes location counter at start of procedure.
IL0011	Resumes containing procedure at end of procedure.
IL0012	Generates text for RR instructions.
IL0013	Generates text for RX non-branch branch instructions, LM, STM, and SI type.
IL0014	Generates text for SS instructions.
IL0015	Processes the start of prologues.
IL0016	Processes the end of prologues.
IL0017	End-of-text routine.
IL0019	Sets up trace information and numbers compiler labels.
IL0020	Generates text for branch and load address instructions.
IL0022	End-of-block routine.
IL0027	Generates text for shift instructions.
OFFSET (TU)	Determines offset and relocation pointer from given dictionary reference.
PUNCHT	Punches cards ensuring that RLD cards follow related TXT card.

Table UA. Phase UA Final Assembly Initial Values, Pass 1

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans STATIC chain to beginning of external section	UA001	UA200, UA220, UA230
Initializes scalar variables	UA200	TXTMOV
Initializes BCD for label	UA220	RLDMOV, TXTMOV
Initializes DED for temporary	UA230	TXTMOV
Initializes address constants.	UA010	UA401, UA403, UA404, UA405, UA406
Initializes symbol table entries	UA080	RLDMOV, TXTMOV
Initializes address slots for external variables	UA403	RLDMOV, TXTMOV
Initializes address slots for functions and programmer-defined ON-condition names	UA401	RLDMOV, TXTMOV
Initializes address slots for label constants	UA404	RLDMOV, TXTMOV
Initializes address slots for entry labels	UA405	RLDMOV, TXTMOV
Initializes file attribute entries and files	UA406	RLDMOV, TXTMOV
Initializes constants pool	UA014	RLDMOV, TXTMOV
Initializes dope vector skeletons	UA021	TXTMOV
Initializes argument lists	UA025	RLDMOV, TXTMOV

Table UA1. Phase UA Routine/Subroutine Directory

Routine/Subroutine	Function
OUTPUT (UB)	Moves card images to load file.
RLDMOV (UB)	Moves RLD entries to card buffer.
TXTMOV (UB)	Moves TXT entries to card buffer
UA0000	Entry point from compiler control.
UA001	Scans STATIC chain to start of external section, to initialize scalar variables.
UA0015	Return point for branches taken in first scan.
UA010	Initializes address constants.
UA013	Return point for branches taken in second scan.
UA014 (UC)	Initializes constants pool.
UA021	Initializes dope vector skeletons.
UA0215 (UC)	Produces text for dope vector skeleton.
UA025	Initializes argument lists.
UA033	Return point for branches taken in last scan.
UA080 (UC)	Initializes symbol table entries.
UA100 (UC)	Initializes one-word CSECT 'IHEMAIN'.
UA100A	Exit from UA to compiler control and UD.
UA200	Initializes scalar variables.
UA220 (UC)	Initializes BCD for label.
UA225 (UC)	Entry to label routines for label variable BCDs.
UA230 (UC)	Initializes DED and FED for temporary.
UA401	Initializes address slots for functions and programmer-defined ON-condition names.
UA403	Initializes address slots for external variables.
UA404	Initializes address slots for label constants.
UA405	Initializes address slots for entry labels.
UA406	Initializes DECLARE control blocks for files and file attributes entries.
UA407	Makes text for file attributes entry.
UCINIT (UC)	Initializes array variables.
UCUPDT (UC)	Initializes arrays of varying strings.
UC0080 (UC)	Initializes bit arrays.
TIDY (UC)	Completes packing of bit strings in structures or arrays.

Table UD. Phase UD Final Assembly Pseudo-Code Static DSA's

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans STATIC DSA chain	A1	AUTO
Scans STATIC DSA's AUTOMATIC chain	AUTO	DAT,LAB,STRUC
Initializes dope vectors for data items and label variables (unstructured)	DATLAB	TXTMOV(UB),RLDMOV(UB)
Initializes dope vectors for structures	STRUC	TXTMOV(UB),TLDMOV(UB)

Table UD1. Phase UD Routine/Subroutine Directory

Routine/Subroutine	Function
A1	Scans STATIC DSA chain.
AUTO	Scans STATIC DSAs AUTOMATIC chain.
DATLAB	Initializes dope vectors for data items and labels.
STRUC	Initializes structure dope vectors.
UD000	Entry point
UDEND	Releases control.

Table UE. Phase UE Final Assembly Initial Values, Pass 2

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans STATIC chain to beginning of external section	UA001	UA200, UA220, UA230
Initializes scalar variables	UA200	TXTMOV (UB)
Scans STATIC chain to initialize internal dope vectors	UA003	UA300, UA320, UA340, UA360, UA365
Initializes dope vectors for internal strings	UA300	RLDMOV (UB), TXTMOV (UB)
Initializes dope vectors for internal data arrays	UA320	RLDMOV (UB), TXTMOV (UB)
Initializes dope vectors for arrays of varying strings	UA340	TXTMOV (UB), UCUPDT (UC)
Initializes dope vectors for internal label arrays	UA360	RLDMOV (UB), TXTMOV (UB)
Initializes dope vectors for internal structures	UA365	UA300, UA320, UA360
Initializes arrays	UA030	RLDMOV (UB), TXTMOV (UB), UCINIT (UC)
Initializes structures	UA040	TXTMOV (UB), UA200, UC0800 (UC), TIDY (UC)
Initializes one word CSECT 'IHEMAIN'	UA100	OUTPUT, RLDMOV, TXTMOV (all in UB)
Initializes CSECT for STATIC external variables	UA1005	OUTPUT (UB), UA030, UA200, UA300, UA320, UA360, UA365, UA401, UA406
Makes up END card and terminates phase	UA120	OUTPUT (UB)
Initializes array variables	UCINIT (UC)	TXTMOV (UB), UC0080 (UC), TIDY (UC)

Table UE1. Phase UE Routine/Subroutine Directory

Routine/Subroutine	Function
AREA	Initializes AREA variables.
EVENT	Initializes EVENT variables.
TASK	Initializes TASK variables.
UA000	Entry point from UA and compiler control.
UA001	Scans STATIC chain to start of external section, to initialize scalar variables.
UA0015	Return point for branches taken in first scan.
UA003	Scans STATIC chain to initialize all dope vectors for internal variables.
UA021	Start of scan for arrays and structures.
UA030	Initializes arrays.
UA031	Produces RLD entry for label array virtual origin.
UA033	Return point for branches taken in array scan.
UA034	Produces RLD entry for data array virtual origin.
UA040	Initializes structures.
UA100 (UC)	Initializes IHEMAIN CSECT.
UA105	Return point for branches taken in external scan.
UA120	Makes up END card and terminates phase.
UA200	Initializes scalar variables.
UA207	Lists label variables.
UA300	Initializes dope vectors for internal strings.
UA320	Initializes dope vectors for internal data arrays.
UA340	Initializes dope vectors for arrays of varying strings.
UA360	Initializes dope vectors for internal label arrays.
UA365	Initializes dope vectors for internal structures.
UA401	Initializes address slots for functions and programmer-defined ON-condition names.
UA406	Initializes DECLARE control blocks for files and file attributes entries.
UA1005	Initializes CSECTs for STATIC external variables.

Table UF. Phase UF Final Assembly Object Listing

Statement or Operation Type	Main Processing Routine	Subroutines Used
Scans Text	IL0002	None
Lists RR instructions	IL0012	PRINIT, RRRTN
Lists RX non-branch instructions	IL0013	BXRTN, PRINIT, PRNTOU, PRNTVF, SECOND
Lists SS instructions	IL0014	EOBRTN, PRINIT, PRNTOU, SSRTN
Lists shift instructions	IL0026	PRINIT, PRNTOU, PRNTVF
Lists LM and STM	IL0027	PRINIT, PRNTOU, PRNTVF, SECOND
Lists SI instructions	IL0028	CHARVF, PRINIT, PRNTOU, PRNTVF SECOND, SSRTN
Lists branch and load address instructions	IL0020	IL0013, NAMEIT, NAMEQU, PRINIT, RRRTN
Lists labels	IL0019	NAMEVF, NEXTEL, NEXTSL, PRNTLC, PRNTOU, PRNTVF, STATMN
Lists procedure names	IL0010	NAMEVF, NEXTEL, PRNTOU, STATMN
Lists ends of procedures	IL0011	NAMEVF, NEXTEL, PRNTOU
Scans ahead for literal offsets; inserts second instruction byte into print image	SECOND	EOBRTN
Generates listing of text for base offset pair	SSRTN, BXRTN	ABSOFF, ADDEND, NAMEIT, NAMEQU, PRNTVF
Names generated label number	NAMEQU	DECINT, FINEQ1
Inserts location counter value, and hexadecimal and mnemonic operation codes in print line	PRINIT	PRNTLC
Moves variable length item into variable field part of print line	PRNTVF	PRNTOU
Lists statement numbers	STATMN	STATNO
Determines name and offset from dictionary reference	NAMEIT	DECINT, HEXINT
Generate listing of compiler subroutine	IL0017	PRNTLC,PRNTVF,PRNTOU

Table UF1. Phase UF Routine/Subroutine Directory (Part 1 of 2)

Routine/Subroutine	Function
ABSOFF	Appends literal offsets to operands in variable part of print line.
ADDEND	Appends signed literal offsets to operands.
BXRTN/SSRTN	Generate listing of text for base offset pair.
CHARVF (UG)	Places one character in variable field of print line image.
DECINT (UG)	Converts binary to externally coded decimal.
EOBRTN	Chains to next input block.
FINEQ1	Locates label number table entries.
HEXINT (UG)	Converts binary to externally coded hexadecimal.
IL0000	Entry point from compiler control.
IL0002	Scans text.
IL0003	Entry to scan from initialization routines.
IL0010 (UG)	Lists procedure names.
IL0011 (UG)	Lists ends of procedures.
IL0012	Lists RR instructions.
IL0013	Lists RX non-branch instructions.
IL0014	Lists SS instructions.
IL0015	Processes the start of prologues.
IL0016	Processes the end of prologues.
IL0017 (UI)	End-of-text routine, and compiler subroutine listing.
IL0018	Processes compiler generated label numbers.
IL0019 (UG)	Lists labels.
IL0020	Lists branch and load address instructions.
IL0026	Lists shift instructions.
IL0027	Lists LM and STM.
IL0028	Lists SI instructions.
IL0032	Processes SS decimal instructions.
IL1003 (UG)	Prints "*PROCEDURE" followed by entry names and statement number.
IL2005	Identifies operands.
NAMEIT	Determines name and offset from dictionary entry.
NAMEQU	Names generated label number.
NAMEVF (UG)	Places a variable name in the print line.
NEXTEL (UG)	Scans dictionary for multiple entry labels.

Table UF1. Phase UF Routine/Subroutine Directory (Part 2 of 2)

Routine/Subroutine	Function
NEXTSL (UG)	Scans dictionary for multiple statement labels.
NM0003 (UH)	Common return point in naming routine.
PRINIT (UG)	Prints location counter value, hexadecimal, and mnemonic op codes.
PRNTLC (UG)	Converts location counter to hexadecimal; places it in print image.
PRNTOU (UG)	Prints a line.
PRNTVF (UG)	Moves variable length item into variable field part of print line.
RRRTN	Generates RR format listing of text.
SECOND	Scans ahead for literal offsets; inserts second instruction byte into print image.
STATMN (UG)	Lists statement numbers.
STATNO (UG)	Converts statement number to decimal.

ERROR EDITOR PHASE TABLES

Table XA. Phase XA Error Message Editor

Statement or Operation Type	Main Processing Routine	Subroutines Used
Determines whether error messages are to be printed	XA	None
Scans error message text skeletons and prints them out	XA8	XA50, XA70, XA90, XA110, ZUPL

Table XA1. Phase XA Routine/Subroutine Directory

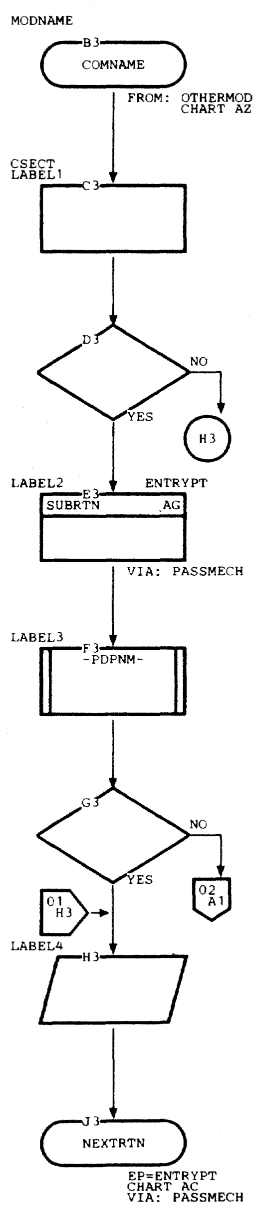
Routine/Subroutine	Function
XA	Determines whether error messages are to be printed.
XA0	Sets severity code.
XA01	Establishes which message types to suppress.
XA1	Counts number of error chains to be processed.
XA2	Puts out messages if there are no diagnostics.
XA4	Prints out "COMPILER DIAGNOSTIC MESSAGES".
XA7	First scan of message chains.
XA8	Scans error message text skeletons and prints them.
XA9 (XB)	Scans to head of next non-empty chain.
XA12A	Selects and prints header for messages of given severity.
XA30 (XB)	Gets next entry in message chain.
XA32 (XB)	Builds up first part of message in buffer.
XA35 (XB)	Accesses message skeleton.
XA40 (XB)	Puts out completed message.
XA50 (XB)	Moves message text to print buffer.
XA70 (XB)	Converts binary statement number to character representation, and moves it to print buffer.
XA90 (XB)	Converts binary numeric value to character representation and moves it to print buffer.
XA110 (XB)	Moves identifier from dictionary entry to the print area.
ZUPL	Prints a line on PLILIST data set.

FLOWCHART CONVENTIONS

The flowcharts in this manual were produced by the IBM System/360 Flowchart Program (FL/I), using ANSI symbols. Following is a description of the ANSI symbols and flowchart conventions.

SYMBOL	DEFINITION	COMMENTS
B1 TERMINAL BLOCK	INDICATES AN ENTRY OR TERMINAL POINT IN A FLOWCHART; SHOWS START, STOP, HALT, DELAY, OR INTERRUPTION. MAY ALSO INDICATE RETURN TO THE CALLING PROGRAM.	B3: MODNAME IS THE LOAD MODULE OR LIBRARY NAME OF THE ROUTINE DESCRIBED BY THIS FLOWCHART. COMNAME IS THE COMMON NAME OF THE ROUTINE. OTHERMOD INDICATES THE MODULES PASSING CONTROL TO THIS MODULE AND THEIR FLOWCHARTS.
C1 PROCESS BLOCK	INDICATES A PROCESSING FUNCTION OR A DEFINED OPERATION CAUSING CHANGE IN VALUE, FORM OR LOCATION OF INFORMATION.	C3: CSECT IS THE CSECT NAME OR OTHER ENTRY POINT AT WHICH PROCESSING BEGINS. LABEL1 IS THE LABEL OF THE FIRST INSTRUCTION.
D1 DECISION BLOCK	INDICATES A DECISION OR SWITCHING-TYPE OPERATION THAT DETERMINES WHICH OF A NUMBER OF ALTERNATE PATHS SHOULD BE FOLLOWED.	D3: PROGRAM EXECUTION CONTINUES WITH BLOCK H3 WHEN THE DECISION IS NO, OR BLOCK E3 WHEN THE DECISON IS YES.
E1 SUBROUTINE BLOCK	INDICATES A SUBROUTINE OR MODULE THAT IS DESCRIBED IN THIS MANUAL	E3: LABEL2 IS THE LABEL OF THE SECTION OF CODE IN THIS ROUTINE FROM WHICH CONTROL IS PASSED TO THE SUBROUTINE. CONTROL RETURNS TO THE NEXT INSTRUCTION FOLLOWING THE SUBROUTINE CALL. ENTRYPT IS THE ENTRY POINT. SUBRTN IS THE COMMON NAME OF THE SUBROUTINE IN FLOWCHART AG. VIA: PASSMECH INDICATES HOW CONTROL PASSES FROM COMNAME TO SUBRTN.
F1 PREDEFINED PROCESS BLOCK	INDICATES A SUBROUTINE OR MODULE THAT IS INCLUDED IN THE FLOWCHARTS OF ANOTHER MANUAL.	F3: LABEL3 IS THE LABEL OF THE SECTION OF CODE FROM WHICH CONTROL IS PASSED TO THE PREDEFINED PROCESS PDPNM, WHICH IS DOCUMENTED IN ANOTHER PUBLICATION (-PDPNM- MAY ALSO BE USED IN A PROCESSING BLOCK).
G1 INPUT/OUTPUT BLOCK	INDICATES GENERAL I/O FUNCTIONS, SUCH AS GET, PUT, READ, WRITE, SIO, AND DEVICE-CONTROL MACRO INSTRUCTIONS.	G3: EXECUTION CONTINUES WITH BLOCK H3 WHEN THE DECISION IS YES, OR WITH BLOCK A1 ON PAGE 2 OF THIS SET OF FLOWCHARTS WHEN THE DECISION IS NO. THE OFFPAGE CONNECTOR MARKED 01H3 INDICATES THAT EXECUTION CONTINUES WITH BLOCK H3 FROM ANOTHER PAGE OF THIS SET OF FLOWCHARTS. THIS CONNECTOR IS ALSO PAIRED WITH THE ONPAGE CONNECTOR FROM BLOCK D3.
H1 PREPARATION BLOCK	INDICATES A PROCESS THAT CHANGES SYSTEM OPERATION, FOR EXAMPLE, SETS A SWITCH, MODIFIES AN INDEX REGISTER, OR INITIALIZES A ROUTINE.	H3: LABEL4 IS THE LABEL OF A SECTION OF CODE OF THIS ROUTINE THAT INITIATES I/O.
ONPAGE CONNECTOR	INDICATES ENTRY TO OR EXIT FROM ANOTHER BLOCK ON THE SAME FLOWCHART PAGE.	J3: NEXTRTN IS THE COMMON NAME OF THE ROUTINE THAT EXECUTES AFTER THIS ROUTINE. ENTRYPT IS THE ENTRY POINT OF NEXTRTN, WHICH IS DESCRIBED IN CHART AC. VIA: PASSMECH INDICATES HOW CONTROL PASSES FROM COMNAME TO NEXTRTN.
OFFPAGE CONNECTOR	INDICATES ENTRY TO OR EXIT FROM A BLOCK ON ANOTHER PAGE OF THE SAME SET OF FLOWCHARTS.	

EXAMPLE



Program Logic Manual

GY28-2051-0

PL/I Compiler

Flowcharts on pages 221-360 were not scanned.

RESIDENT TABLES

There are three resident tables: the dictionary, the keyword tables, and the phase directory. The dictionary is resident through part of the compilation; the formats of the dictionary entries are fully described in this section. The keyword tables are resident during the read-in logical phase, and the phase directory throughout the compilation.

ORGANIZATION OF KEYWORD TABLES

The read-in phase is divided into five passes containing the modules shown in Figure 10.

Modules CA and CC contain routines which are common to all five passes. The keyword tables are held in separate modules (CE, CK, CN, and CR) which must each be less than 1,024 bytes (1K) long. In this way it is possible to hold in storage only those keywords which are required for any one pass. The keyword tables are constructed in the following manner.

For ease of searching and modifying a keyword table, it is organized into two levels and by keyword length, as shown in Figure 11.

The KEYWD routine is called by one of the statement scanning routines, and is supplied with a parameter which enables it to decide which set of keywords to look at (e.g., statement identifier, ON condition, miscellaneous). It does this by using the parameter to extract the required relative address (R(A),etc.) from the first level directory. The second level directory provides the KEYWD routine with the means of reaching a table containing keywords of correct length; the KEYWD routine calls the KEYID routine, which scans the next significant item in the source text to obtain the length used in this look-up.

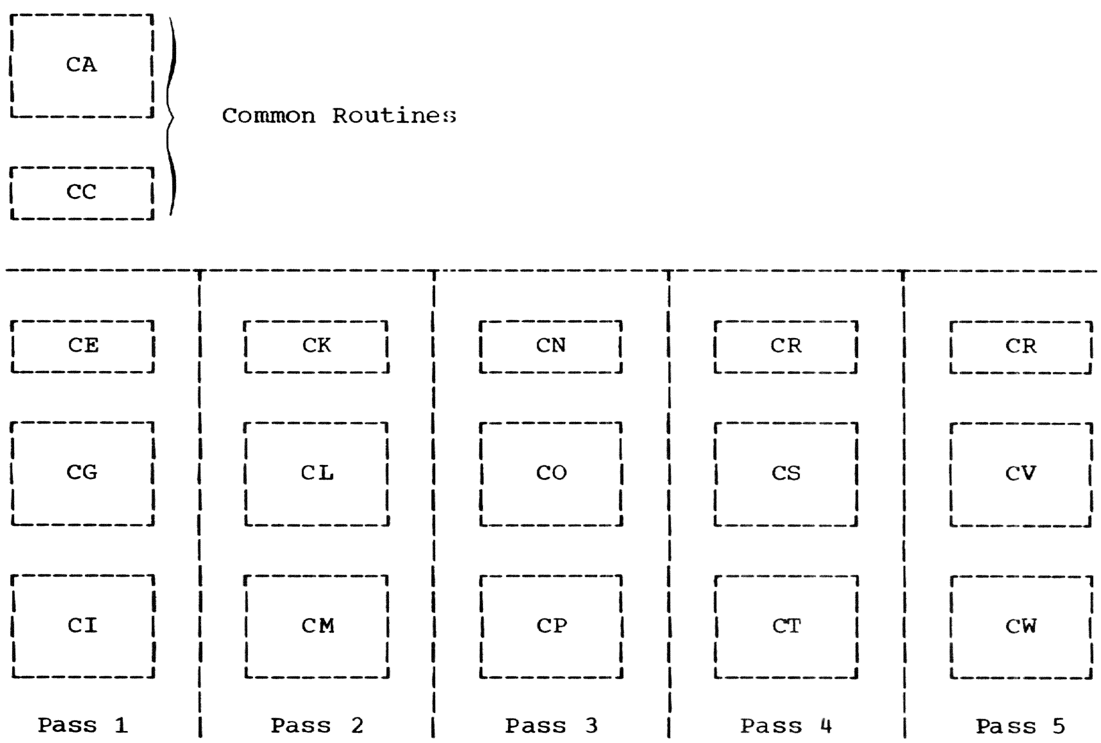


Figure 10. Organization of Read-In Phase

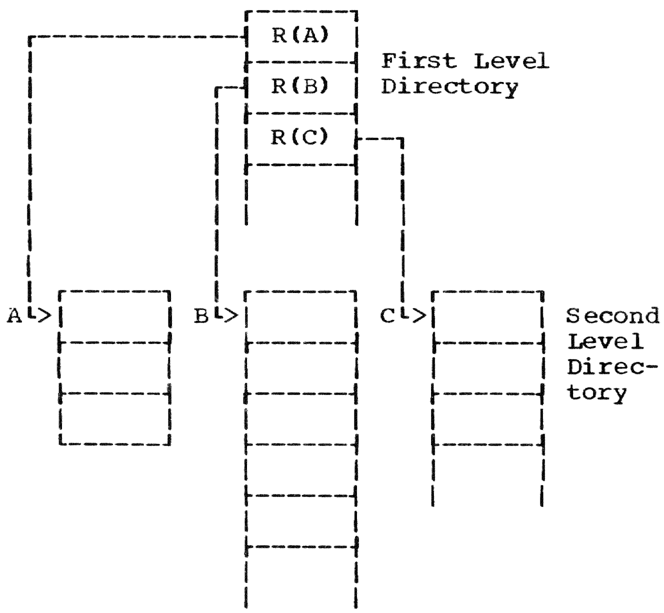


Figure 11. Organization of Keyword Table

Format of First Level Directory

```
FSTLVL DC AL2(STATID - FSTLVL)

       DC AL2(ONID - FSTLVL)
```

Format of Second Level Directory

The second level tables contain relative addresses, which enable the KEYWD routine to reference a third level table containing keywords of the correct length. If one of these entries should contain zero, then KEYWD will interpret this as meaning that no keywords of this length exist in this table.

```
STATID DC FL2'm' where m is smallest
            length in table

       DC FL2'n' where n is largest
            length in table

       DC AL2(STLm-STATID)

       DC AL2(STLn-STATID) where the
            symbols beginning STL are
            the symbolic addresses of
            the corresponding keyword
            tables
```

Format of Third Level Tables

The third level tables have a prefix byte containing the number of entries in this particular table followed by keyword entries. These consist of the keyword in internal code plus the replacement character (keywords recognised as such are replaced by a single code byte).

```
STLm DC FL1'x' where x is number of
          keywords in this table

     DC X'112315' keyword in internal
          code

     DC X'55' replacement in internal
          code

     DC X'393839'

     DC X'5A'
```

Some keywords are not represented by one word (e.g., GO TO, BY NAME) and clearly, the mechanism must be modified to cope with the second word. This modification is achieved by placing a 1-bit into the first bit of the first level by using the OR logical operation. The presence or absence of this bit is tested by the KEYWD routine before the suspected keyword is compared. If the bit is absent, the pass through the routine is quick, as there is no possibility of an extra level search. If the bit is present, the keyword must be compared after the additional bit has been removed by the AND logical operation. If the comparison is equal, the two bytes following the replacement character are used as a relative address to reach the next level table.

Format of Entry Requiring Additional Comparisons

```
DC X'9726' GO + X'1000'

DC X'40'

DC AL2(N XTLVL-*) Relative address
     of next level table
```

The format of these extra level tables is similar to that for the third level. In this way, it is possible for national language keywords to replace single words by two or more words, if so desired.

PHASE DIRECTORY

The phase directory is a list maintained in module AA. Each entry in this list is 8 bytes long. The first two bytes contain the module name. The remaining 6 bytes are initially blank. When an output module is loaded, the address in virtual memory of each module within that output module is slotted into the last 4 bytes of the relevant entry in the phase directory. This directory is used by the phase linkage routines in module AA to locate the compiler modules in virtual memory.

The format of a phase directory entry is as follows:

Byte Number	Description
0 - 1	Module name
2-7	Initially blank; however, when output module is loaded, bytes 4-7 contain address of individual module in output module.

INTERNAL FORMATS OF DICTIONARY ENTRIES

The following description of the formats of dictionary entries during the compilation of a source program is organized in this manner:

1. Dictionary Entry Code Bytes
2. Dictionary Entries for ENTRY Points
3. Code Bytes for ENTRY Dictionary Entries
4. Dictionary Entries for DATA, LABEL, and STRUCTURE Items
5. Code Bytes for DATA, LABEL, and STRUCTURE Dictionary Entries
6. Format of Variable Information
7. Other Dictionary Entries
8. Dimension table
9. Dictionary Entries for Initial Values

1. DICTIONARY ENTRY CODE BYTES

The dictionary is used to communicate a complete description of every element of the source program, the compiled object program, and the compiler diagnostic messages between phases of the compiler; the text describes the operations to be carried out on the elements.

Each type of element has a characteristic dictionary entry, which is identified by a code occupying the first byte of the entry. In general, each type of element has a different code byte, but in order to permit rapid identification of dictionary entries, the code bytes have been allocated on the following basis:

First Half Byte

Bit Position	Bit Value	Meaning
0	0	entry has BCD
	1	entry has no BCD
1*	0	entry is to be chained
	1	entry not to be chained
2	0	not a member of structure
	1	member of structure
3	0	not dimensioned
	1	dimensioned

*This bit only applies to Phase FT which constructs the storage class chains by a sequential scan of the dictionary; later in the compiler, items with this bit on are added to the storage class chains.

Second Half Byte

In the second half byte, the following codes have the meanings shown, unless the first half byte is X'C':

X'7'	means	label variable
X'C'	means	task identifier
X'D'	means	event variable
X'E'	means	structure
X'F'	means	data variable

The second and third bytes of every dictionary entry contain the length, in bytes, of the entry. If the entry has BCD (i.e., the first bit of the entry is zero), this length count does not include the BCD; instead, the BCD, which follows the main body of the entry, is preceded by a single byte containing one less than the number of characters of BCD.

Using this general scheme, the code bytes allocated for dictionary entries appear in the following table. Code bytes in the table which have no corresponding description are not allocated.

X'00'	Statement label constant
01	Procedure or entry label
02	GENERIC entry label
03	External entry label (entry type 4)
04	Built-in function, e.g., DATE
05	Temporary variable and controlled allocation workspace
06	Built-in GENERIC label, e.g., SIN
07	Label variable
08	File constant
09	
0A	
0B	
0C	Task identifier
0D	Event variable
0E	
0F	Data variables (not dimensioned or a structure member)
10	
11	
12	
13	
14	
15	

16	
17	Dimensioned label variable
18	
19	
1A	
1B	
1C	Dimensioned task identifier
1D	Dimensioned event variable
1E	
1F	Dimensioned data variable
20	
21	
22	
23	
24	
25	
26	
27	Label variable in structure
28	
29	
2A	
2B	
2C	Task identifier in structure
2D	Event variable in structure
2E	Structure item
2F	Data variable in structure
30	
31	
32	
33	
34	
35	
36	
37	Dimensioned and structured label variable
38	
39	
3A	
3B	
3C	Dimensioned task identifier in structure
3D	Dimensioned event variable in structure
3E	Dimensioned structure item
3F	Dimensioned and structured data variable
40	Formal parameter type 1
41	
42	
43	
44	
45	
46	
47	
48	
49	
4A	
4B	
4C	
4D	ON CONDITION entry
4E	
4F	
80	ENTRY type 1 -- from a PROCEDURE statement
81	BEGIN statement entries -- entry type 1
82	ENTRY statement -- entry type 1
83	Entry type 5
84	Entry type 3
85	Entry type 2
86	Entry type 6
87	Label variable formal parameter or temporary
88	Constant
89	File formal parameter or file temporary
8A	
8B	
8C	Task identifier formal parameter
8D	Event variable formal parameter
8E	
8F	Data variable formal parameter or temporary
90	Invocation count dictionary entry
91	
92	
93	
94	
95	
96	
97	Dimensioned variable formal parameter or temporary
98	File attribute entry
99	
9A	
9B	
9C	Dimensioned task identifier formal parameter
9D	Dimensioned event variable formal parameter
9E	
9F	Dimensioned data variable formal parameter or dimensioned temporary
A0	
A1	
A2	
A3	
A4	
A5	
A6	
A7	Structured label variable temporary
A8	
A9	
AA	
AB	
AC	Structured task identifier temporary
AD	Structured event variable temporary
AE	Temporary or formal parameter structure
AF	Structured data variable temporary
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	Dimensioned and structured label variable temporary

B8	
B9	
BA	
BB	
BC	Dimensioned and structured task identifier temporary
BD	Dimensioned and structured event variable temporary
BE	Dimensioned structure formal parameter or temporary
BF	Dimensioned and structured data variable temporary
C0	String dope vector for temporary
C1	DED2 entry
C2	Internal library function, e.g., conversion routines
C3	Compiler label
C4	Prefix ON list item
C5	Parameter lists
C6	Dope vector skeletons
C7	Symbol table entry or DED entry
C8	Error message, table entry, workspace requirement, STATIC DSA, etc.
C9	Record dope vector (RDV) entry
CA	Workspace requirement entry
CB	Select a member from a generic family
CC	AUTOMATIC chain delimiter or Dope Vector Descriptor (DVD) entry
CD	ON condition entry
CE	Label BCD entry
CF	End of information in dictionary block

2. DICTIONARY ENTRIES FOR ENTRY POINTS

Entry type 1 for PROCEDURE, BEGIN, and ENTRY statements

The format of an entry for a PROCEDURE statement is as follows:

Byte Number	Description
1	Code byte X'80'
2-3	Length
4	Level
5	Count
6-7	Dictionary reference to the entry type 1 of the containing block
8-9	Dictionary reference of the dictionary entry for the first label that was attached to the PROCEDURE statement
10-11	Dictionary reference to the entry type 1 of the next PROCEDURE or BEGIN statement in the source program
12-13	The start of the chain of all AUTOMATIC variables
14-15 16-17 18-19	Dictionary references to three dictionary entries indicating storage requirements for workspace
20-21	Dictionary reference of CHECK list set by phase FO. Phase QU re-uses this slot and sets it to the offset from register 9, at which the register allocator workspace is to start.
22-23	Dictionary reference of NOCHECK list
24-25	Dictionary reference of the first symbol table entry for this block
26-28	Size of the DSA for this block, set in storage allocator phase Note: If this procedure has a static DSA, a "C8 static DSA" entry is made by phase PA and the dictionary reference of this entry is put in bytes 27 and 28. Phase MA moves, into bytes 27 and 28, the dictionary reference of a temporary describing a table to be built at execution time for the TRANSLATE or VERIFY function.
29-31	Offset of the eight words in the DSA used for addressing the DSA
32-34	Offset of the storage used for the parameter list necessary in an ALLOCATE- FREE statement
35-37	Offset of the two-byte switch which is set on entry to a procedure and tested at a RETURN (expression)
38-40	Offset of the four-byte slot which will contain the address of the first approximation of the target field (the address of the implied parameter)
41-42	Dictionary reference of the entry type 1 of the first ENTRY statement of the procedure. The entry type 1 for PROCEDURE and ENTRY statements of any one procedure form a circular chain. If there are no ENTRY statements in a procedure this slot will contain the dictionary reference of the PROCEDURE's entry type 1, i.e., of the entry in which the slot occurs
43	OPTIONS code byte
44-57	Eight 2-byte dictionary references to dictionary entries

	for prefix options. Only those prefix options which are changed within the procedure have a dictionary reference. The remainder are zero. The order of the options in this list is the same as in the options byte. (See "Options Code Byte" in Section 3 below)
60	Options change byte. This byte contains a one bit for each prefix option which is changed within the procedure. Its format is identical with the normal options byte
61-63	Offset of workspace used in BUY statement
64	Optimization byte
65	2*n where n is the number of parameters at this entry point
66 onwards	N dictionary references of formal parameter type 1 entries

The format of an entry for a BEGIN statement is similar to the above for the first 34 bytes. The initial code byte is X'81', and the dictionary reference in bytes 8 and 9 is that of the first label on the original BEGIN statement, if any. If there was no statement label, then the statement number occupies this slot. The presence of a statement number or statement label is indicated by a flag byte in position 35. This is set to SN for a statement number, or to SL for a statement label. Bytes 36-56 contain the same as bytes 44-64 in a PROCEDURE entry type 1.

The format for the entry type 1 derived from an ENTRY statement is as follows:

Byte Number	Description
1	Code byte X'82'
2-3	Length
4	Level
5	Count
6-7	Dictionary reference of the next member in the circular PROCEDURE-ENTRY chain
8-9	Dictionary reference of the dictionary entry for the first label on the original ENTRY statement
10-12	The offset of the apparent entry point
13	2*n where n is the number of parameters
14	n dictionary references to the formal onwards parameter type 1 entries

The labels on a PROCEDURE or ENTRY statement will be placed in the dictionary according to the following format:

Byte Number	Description
1	Code byte X'01'
2-3	Length
4-5	Hash chain(STATIC chain)
6-8	Pointer to transfer vector
9-10	Statement number. If the label is mentioned in an ON CHECK list, this slot contains the dictionary reference of Label BCD Entry (X'CE'), which has the statement number in bytes 4 and 5
11	Other 1 code byte. (See "First code byte - other 1" in Section 5 below.) The last bit will always be set to one, unless the label is the last label for a particular statement, in which case the last bit will be set to zero.
12-13	Pointer to entry type 2
14-16	Spare bytes for final assembly. The pseudo-code phase dealing with RETURN (expression) will insert into these bytes a code which must be stored in a specific slot in the DSA whenever the procedure is entered via this label. The code is used by the prologue construction phase. Byte 16 in the first label for each PROCEDURE or ENTRY statement will contain the number of labels associated with that statement
17	Block level
18	Block count
19	Count of containing block
20	BCD length-1
21	BCD of label

Entry Type 2

An entry type 2 describes the data attributes of an entry point. The format is as follows:

Byte Number	Description
1	Code byte X'85'
2-3	Length.
4-5	Dictionary reference of entry type 3
6-8	Offset, i.e., the position of the string dope vector in the DSA of the block to which the entry belongs. This will be zero if the item is not a string.
9	DATA byte (see "DATA Byte" in Section 5 below).
10-12	Data information, which is: 1. with numeric data, the precision and scaling, left justified 2. for strings of fixed maximum length, the binary version of the string length in the two leftmost bytes of the data information 3. for strings of adjustable length, the text reference of a second file statement giving the expression for the string length
13-14	Picture table reference, if required. The storage allocation phase will change this to the dictionary reference of a DED entry, the picture table reference being moved into this reference if necessary

Entry Type 3

Entry type 3 dictionary entries are constructed either from an explicit declaration or from implicit and default rules. Their format is as follows:

Byte Number	Description
1	Code byte X'84'
2-3	Length of entry.
4-5	Dictionary reference of entry type 1 of PROCEDURE or ENTRY statement.
6-7	Dictionary reference of entry type 2. This describes the value returned when the label associated with this entry type 3 is invoked as a function.
8-10	The offset in the DSA of the containing block of the first approximation of the storage for the value returned by this entry point, when it is invoked as a function.
11	The entry code byte. (See "Entry Code Byte" in Section 3 below)
12-13	The dictionary reference of an item in the AUTOMATIC chain of the containing block. Entry type 3 entries feature in the AUTOMATIC chain of the containing block.
14-15	Switch bytes. The pseudocode phase dealing with RETURN (expression) inserts into these bytes the bit pattern of the code which will signify that entry to the procedure was by the label associated with this particular entry type 3. Phase QF will use this to create MVI instructions.
16-17	Dictionary reference of a SETS list. This will be zero if the attribute SETS was not specified. The format of a SETS list is given at the end of this section.
18-19	Dictionary reference of the dictionary entry for the label belonging to this entry type 3.
20	Status byte. This byte will contain X'00' or X'F0'. X'00' indicates that the entry was constructed from an ENTRY declaration which had parameter descriptions. X'F0' indicates the entry was constructed either artificially or from an ENTRY declaration which did not have parameter descriptions.
21	2*n where n is the number of parameters. This is zero if the status byte is X'FF'
22 onwards	If the status byte is X'00' there are n two-byte references of parameter descriptions. A parameter description is a dictionary entry for the particular type of item but without a BCD. If one particular parameter is not described, i.e. if there are two adjacent commas in the ENTRY attribute, then the dictionary reference is zero. When the status byte is X'F0' then an entry type 3 is only 23 bytes long.

22+2n- 23+2n	DECLARE statement number

SETS List Format

Byte Number	Description
1	Code byte X'C8'
2-3	Overall length of original BCD entry
4-5	2*n1 where n1 is the number of identifiers in the SETS list. If * was specified, these bytes contain 2*n1+1.
6-5+2*n1	Dictionary references of the identifiers in the SETS list.
6+2*n1	n2, the number of para- meters in the SETS list.
7+2*n1 onwards	n2 numbers of one byte each. These are the parameter numbers and will be in ascending order.

Entry Type 4

Entry type 4 dictionary entries describe external entry points. Their format is as follows:

Byte Number	Description
1	Code byte X'03'
2-3	Length
4-5	Hash chain, later used as the STATIC chain
6-8	Offset of the load constant in STATIC
9-11	Offset in the DSA of the declaration block of the storage for the first approximation of the value returned.
12-13	The dictionary reference of an item in the AUTOMATIC chain of the declaring block. Entry type 4 entries are members of the AUTOMATIC chain of the declaring block.
14	The Entry code byte. (See "ENTRY Code Byte" in Section 3 below).
15	The DATA byte. (See "DATA Byte" in Section 5 below)
16-18	Data information which is: a) with numeric data, the precision and scaling, left justified b) for strings of fixed maximum length, the binary version of the string length in the two leftmost bytes of the data information c) for strings of adjustable length, the text reference of a second file statement giving the expression for the string length
19-20	Picture table address if required.
21-22	Dictionary reference of a SETS list
23	Status byte. If this byte is X'00' the meaning is the same as the status byte in an entry type 3. If the byte is X'FF' it is implied that no parameters were described
24	2*n where n is the number of parameters. This is zero if the status byte is X'FF' parameters. This is zero if the status byte is X'FF'
25	n dictionary references to parameter descriptions as in an entry type 3
25+2*n	Level
26+2*n	Count
27+2*n	BCD length-1
28+2*n onwards	BCD of identifier

Entry Type 5

Entry type 5 dictionary entries describe the entry points which are formal parameters. They have the same format as entry type 4 except that:

Byte 1 is X'83'

Bytes 4 and 5 contain the address of the formal parameter type 1 entry

Bytes 6 to 8 contain the offset in the DSA of the declaring block of the address slot associated with a formal parameter

No BCD is contained in the entry

GENERIC Entry Point

The format for a GENERIC entry point is as follows:

Byte Number	Description
1	Code byte X'02'
2-3	Length
4-5	Hash chain
6-8	Offset 1 Slot
9-10	DECLARE statement number
11	2n, where n is the number of two-byte addresses following
12-11+2n	Pointers to entry type 4 or 5, ENTRY labels, or BUILTIN entries. These entries are made when an identifier is given the attribute GENERIC. The pointers are to the entries which contain specifications of the various possible attributes
12+2n	Level
13+2n	Count
14+2n	BCD length-1
15+2n onwards	BCD

3. CODE BYTES FOR ENTRY DICTIONARY ENTRIES

ENTRY Code Byte

This code byte is used in entry type 3, 4, and 5 dictionary entries. The format is as follows:

Byte Number	Description
0	IRREDUCIBLE
1	REDUCIBLE
2	USES
3	SETS
4	SECONDARY
5	RECURSIVE
6	Has data attribute
7	Not used

Options Code Byte

This code is used in entry type 1 dictionary entries for PROCEDURE statements. The format is as follows:

Byte Number	Description
0	REENTRANT
1	ON Block
2	MAIN
3	TASK
4	RECURSIVE
5	OPTIONS
6	Contains RETURN (expression) statement
7	ENTRY name is passed as argument

Optimization Byte

This code byte is used in entry type 1 dictionary entries.

Format of the Optimzation Byte:

Byte Number	Description
0	Not eligible for DSA in library.
1	Eligible for DSA in STATIC storage.
2	Needs invocation count.
3	Needs current file slot.
4	Contains asynchronous CALL.
5	Indicates ORDER or REORDER option 0 = ORDER, 1 = REORDER. Default is ORDER
6	Not used.
7	Not used.

4. DICTIONARY ENTRIES FOR DATA, LABEL, AND STRUCTURE ITEMS

Label Variables - Obtained from DECLARE Statement

Byte Number	Description
1	Code byte may be X'07', X'17', X'27', X'37', X'87', X'97', X'A7', X'B7'. The last four cases apply when the item occurred in a parameter list in a PROCEDURE or ENTRY statement. In this case, bytes 4 and 5 will contain the dictionary reference of the corresponding formal parameter type 1 entry. In the

	first four cases, bytes 4 and 5 initially contain the hash chain. After the scan of the dictionary, this slot will be re-used to form another chain, e.g., AUTOMATIC or STATIC chain
2-3	Length
4-5	Initially contains the hash chain. After the dictionary scan, this is re-used to form another chain, e.g., AUTOMATIC or STATIC chain
6-8	Offset inserted by storage allocation phase (as for a data item)
9-10	DECLARE statement number
11	'Other 1' code byte (See "First Code Byte - Other 1" in Section 5 below)
12	'Variable' code byte (See "Variable Byte" in Section 5 below)
13	'Other 2' code byte (See 'Second Code Byte - Other 2" in Section 5 below)
14	'Other 3' code byte (See "Third Code Byte - Other 3" in Section 5 below)
15	'Other 4' code byte (See "Fourth Code Byte - Other 4" in Section 5 below)
16 onwards	Content determined by variable code byte.
	After variable information
2 bytes	Symbol slot
1 byte	Level
1 byte	Count
1 byte	BCD length-1
	BCD

With the exception of the 2-byte symbol slot, the general format is the same as for a structure.

Dictionary Entries for Data Items

The format is as follows:

Byte Number	Description
1	Code byte may be X'0F', X'1F', X'2F', X'3F', X'8F', X'9F', X'AF', or X'BF'. The last four cases apply when the item occurred in a parameter list in a PROCEDURE or ENTRY statement. In this case, bytes 4 and 5 will contain the dictionary reference of the corresponding formal parameter type 1 entry. In the first four cases, bytes 4 and 5 initially contain the hash chain. After the scan of the dictionary this slot will be re-used to form another chain, e.g., AUTOMATIC or STATIC chain
2-3	Length
4-5	See above
6-8	Offset. (See "Format of Variable Information" in Section 6 below)
9-10	DECLARE statement number. If the variable has not been explicitly declared, this number is zero; otherwise, it is the statement number assigned to the DECLARE statement from which the variable was obtained.
11-16	Six code bytes. These are: other 1, variable, other 2, other 3, other 4, and data. (See "Code bytes" in Section 5 below for a description of these bytes.)
17-19	Data information, which is: 1. with numeric data, the precision and scaling, left justified 2. for strings of fixed maximum length, the binary version of the string length in the two leftmost bytes of the data information 3. for strings of adjustable length, the text reference of a second file statement giving the expression for the string length 4. Bit 0 of byte 19 indicates that the temporary is used as an argument for halfword binary temporaries

Note: In the case of a temporary data variable for an argument, the first bit of byte number 19 is set to 1 by phase GP. It is set to 0 by phase IM for temporary arguments to built-in functions and pseudo-variables. It is examined by phase QU.

20-21	Symbol slot, containing either zero, or one of the following: 1. If the SYMBOL and DED bits are not on, and the data item has a picture, these bytes contain the dictionary reference of

the picture table entry
2. If the DED bit is on and the SYMBOL bit off, this slot points at a DED entry. If the item has a picture, the DED entry will contain the picture table address
3. If the SYMBOL bit is on, the slot will point at a SYMBOL entry. This again will contain the picture address, if specified

22	Variable information. The contents of these bytes are determined by the variable code byte. (See "Format of Variable Information" in Section 6 below)
1 byte	Level
1 byte	Count
1 byte	BCD length-1
	BCD

<u>Major and Minor Structure Entries</u>

These entries do not include base elements, i.e., they do not have any data attributes or LABEL. Their format is:

Byte Number	Description
1	Code byte may be X'2E', X'3E', X'AE', or X'BE'. The last two indicate that there is no BCD attached. When the identifier occurs in the parameter list of a PROCEDURE or ENTRY statement, bytes 4-5 contain the dictionary reference of the formal parameter type 1 entry. In the case of the first two code bytes, bytes 4-5 of the entry initially contain the hash chain. This is later modified by Phase FT
2-3	Length
4-5	See byte number 1
6-8	These bytes are used by the storage allocator; they will finally contain one of the following offsets: 1. For structures which are parameters, or are dynamically defined, the offset from the start of the major structures dope vector or the minor structures dope vector. 2. For major structures, the offset from the start of AUTOMATIC or STATIC of the address slot which will point at the structure dope vector 3. For CONTROLLED structures, only that specified for minor structures in 1 , above 4. For structures in STATIC EXTERNAL the contents depend on the setting of the "dope vector required" bit in the "other 3" code byte. If this bit is off and the item is a major structure, the slot contains the offset from the start of STATIC of the slot which will contain the address of the first byte of the structure. If the dope vector bit is on, the slot contains the offset from the start of STATIC of the address slot which will point at the structure dope vector. The offset slot is not used in either of the above cases for minor structures
9-10	DECLARE number, i.e., the statement number of the DECLARE statement which produced the structure
11-15	Five code bytes. These are: other 1, variable, other 2, other 3, and other 4
16	Variable information. The content is determined by the variable code byte, and will always include the information required for structure members. (The format is described under "Format of Variable Information" in Section 6 below)
	After variable information:
1 byte	Level
1 byte	Count
1 byte	BCD length-1
	BCD

5. CODE BYTES FOR DATA, LABEL, AND STRUCTURE DICTIONARY ENTRIES

The First Code Byte - Other 1

Bit No.	Description	Set By
0	Symbol or requires load constant if label constant	Phase EL, FT, or NU
1	Defined on	Phase EL
2	Mentioned in CHECK list	Phase FO
3	Needs DVD	Various
4	Last member in structure	Phases EL or EW
5	Variable dimensions	Phase EL
6	* dimensions	Phases EL and FT
7	* string length for data item	Phases EL and FT
	--More labels follow for a label constant	Phase EG
	---Major Structure - no member of the structure has a dimension or length attribute which is not *	Phase EY

The Second Code Byte - Other 2

Bit No.	Description	Set by
0	Dynamically defined	Phase EL
1	CONTROLLED major structure with varying strings	Phase EY
2	NORMAL = 0, ABNORMAL = 1	Phases EI and FT
3	Reserved	
4	Formal Parameter	Phase EI
5	INTERNAL = 0, EXTERNAL = 1	Phase EI
6 and	00 = AUTOMATIC or DEFINED or simple parameter	Phase EL
7	01 = STATIC	Phase EL
	11 = CONTROLLED	Phase EL

The Third Code Byte - Other 3

Bit No.	Description	Set by
0	Needs dope vector	Phases EK and EY if variable dimension entries, variable string length, or in CONTROLLED storage; Phase NU when item appears in an argument list
1	Needs DED	Phase NU
2	Needs no storage for the item itself	Phase GP
3	Correspondence defined	Phase FV
4	Chameleon	Phase GP
5	Data Variable UNSAFE	Set by phase KE 1= UNSAFE 0= SAFE This use is local to the K phases
	Sign bit for first offset	Phase PH for STATIC and Phase PT for AUTOMATIC
6	Indication of the state of the value in the first offset 0 = rubbish 1 = good value	Phase PH for STATIC and Phase PT for AUTOMATIC
7	As above but for second address slot	Phase PH

The Fourth Code Byte - Other 4

Bit No.	Description	Set by
0	Usage (i): An explicit alignment declaration has been made Usage (ii): A constant has been produced for this structure or array	Phase EL (for EW) Phase JK
1 and 2	00 = Not temporary 01 = Temporary type 2 10 = Temporary not sold 11 = COBOL temporary	Phase GP, HF, HK, IM, or LB
3	Member of defined structure	Phase FV
4	Packed = 0 Aligned = 1	Phase EL
5	Major structure	Phase EL
6	No dope vector initialization	Phase GP
7	A temporary type 2 which has been incorporated in workspace 1 or RDV required. For COBOL temporaries this bit means RDV required	Phase OB

Variable Byte

Bit No.	Description
0	Second address slot
1	Dimensioned
2	Member of structure
3	Value list for label variables or POS for defined items
4	Initial value if not a structure or LIKE if a structure
5	EXTERNAL slot
6	Defined slot
7	CONTROLLED from ALLOCATE statement
Note: For a detailed explanation of the significance of these bits and a description of the extra slots associated with them, see "Format of Variable Information" in this section.	

Data Byte

BIT	0	1	2	3	4	5	6	7	
CAD or NUMERIC FIELD	1	POINTER/ OFFSET *	Sterling NON STERLING	Long Short/ OFFSET	Cad. Numeric Field	Binary Decimal	Float Fixed	Complex Real	1 0
STRINGS	0	Adjustable Length String	Aligned Unaligned	Varying	No Picture Picture	Char Bit	AREA VARI- ABLE*	Not Used	1 0

*AREA, POINTER, and OFFSET data byte settings are:
AREA: X'02' superimposed on the non-pictured CHAR string data byte entry
POINTER: X'40' superimposed on the FIXED BIN data byte entry
OFFSET: X'50' superimposed on the FIXED BIN data byte entry

Note: The LONG bit X'10' is set to 1 for halfword binary and 0 for fullword binary

6. FORMAT OF VARIABLE INFORMATION

Data items, labels, and structures require pointers to various tables if they have certain attributes; for example, if they are dimensioned or defined on a base. Space will be left for information only if the attribute is present. This leads to an addressing problem of how to find the position of the information when the presence of other attributes alter its address.

The problem is resolved by collecting, into one byte, all the attributes which require more than one bit to describe them. This has taken the second place in all the collections of attribute bytes. The presence of a bit in this byte indicates the presence of further information. The offset of this information from the start of the variable information is given by the presence of the bits to the left of the one of interest. Each bit will have a value associated with it. The sum of the values of the bits present and to the left of the one of interest will give the value of the offset. This is achieved in the coding by moving the code byte, masking off the bits to the right of the one being tested and the bit itself, and translating the byte.

The information produced by the presence of the following bits in the variable byte is as follows:

Bit number 0: The second offset slot is 4 bytes long. The contents of this slot are described in this appendix. The decision to include a second offset slot in a dictionary entry is based on questions about the nature of the identifier. Refer to Figure 12.

$\boxed{Y}$ implies that a second offset slot will be given,

$\boxed{N}$ that it will not.

Bit number 1: The dimensioned bit. The slot produced by this is three bytes long. The first byte will contain the number of dimensions, the next two the dictionary reference of the dimension (multiplier) table.

Bit number 2: Member of a structure bit. This slot is ten bytes long and has the following format:

Byte Number	Description
1	Declared level number
2	True level number
3-4	Dictionary reference of the containing structure
5-6	Dictionary reference of the next member in the structure
7	Alignment
8-10	Element length

Bit number 3: POS for defined items. The two-byte slot will contain the POS value as a binary integer.

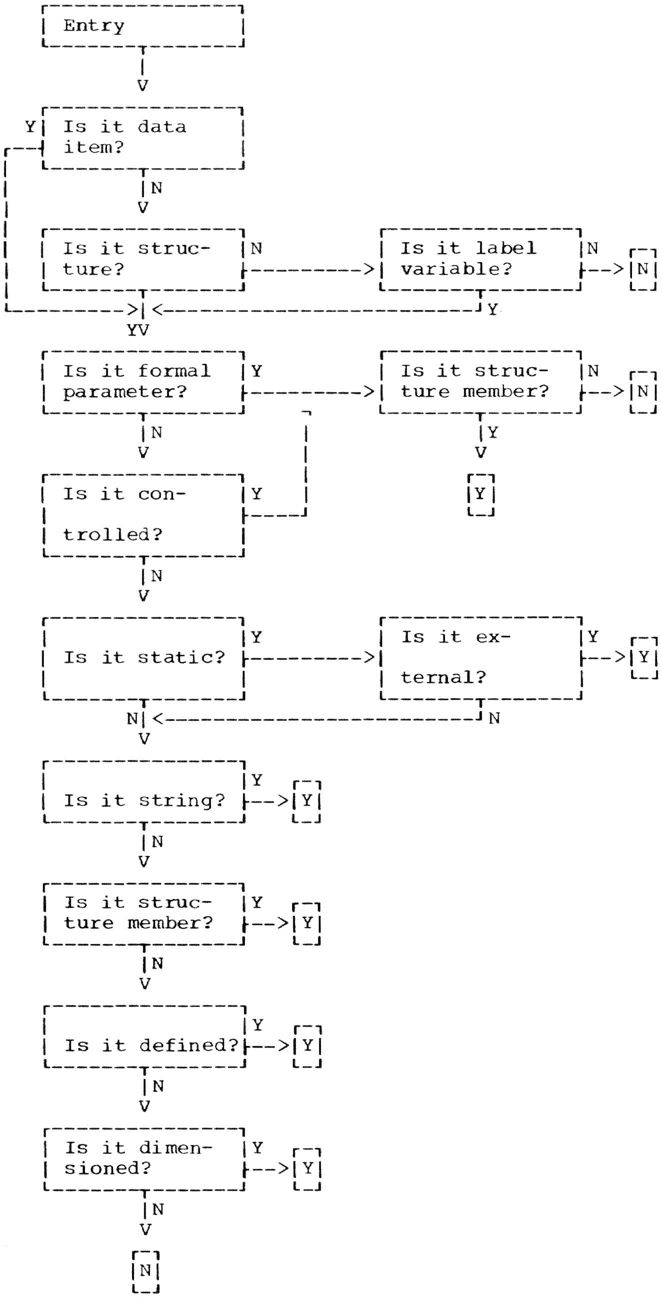


Figure 12. Decision to Include a Second Offset Slot

<u>Bit number 4</u>: The initial value or LIKE bit is a four-byte slot.

1. For normal initial value. The first two bytes contain the dictionary reference of the associated 'Initial Value' dictionary entry. The fourth byte contains X'F0'

2. For INITIAL CALL. The first three bytes contain the text reference of a second file statement. The fourth byte contains X'0F'.

3. For initial labels. The first three bytes contain the text reference of a set of second file statements. The fourth byte contains X'FF'. If there is an initial slot but no initial values the fourth byte contains X'00'

4. For LIKE. The first two bytes contain the LIKE chain. The third and fourth bytes contain the dictionary reference of the likened structure

<u>Bit number 5</u>: The EXTERNAL bit. This 2-byte slot contains the ESD number.

<u>Bit number 6</u>: The DEFINED bit. This 7-byte slot contains the following:

Byte Number	Description
1-2	Defined chain.
3-4	Dictionary reference of base
5-7	The text reference of a second file statement. After the dictionary these bytes will contain X'FFFFFF' if the base is unsubscripted.

<u>Bit number 7</u>: The CONTROLLED from ALLOCATE bit. This bit is on for dictionary entries for level 1 CONTROLLED data specified in ALLOCATE statements. The two-byte slot contains the dictionary reference of the dictionary entry for the data constructed from the DECLARE statement.

<u>Uses of the OFFSET1 and OFFSET2 Slots in Data, Label, and Structure Dictionary Entries</u>

The OFFSET1 slot is in bytes 6-8 of the dictionary entry and the OFFSET2 slot is part of the variable information.

STATIC INTERNAL Structures

Major and minor structure entries: OFFSET1 slot not used. OFFSET2 slot contains offset of structure dope vector from start of STATIC INTERNAL control section (if there is a dope vector)

Basic elements: OFFSET1 slot contains offset of virtual origin (in the case of dimensioned items) or offset of item (when not dimensioned) from start of STATIC INTERNAL control section. OFFSET2 slot contains offset of dope vector (if there is one) from start of STATIC INTERNAL control section

AUTOMATIC Structures

Constant dimensions: as for STATIC INTERNAL except that all offsets are relative to start of DSA.

Adjustable dimensions: major and minor structure entries: OFFSET1 slot not used. OFFSET2 slot contains offset of structure dope vector from start of DSA (if there is a dope vector)

Basic elements: OFFSET1 slot not used. OFFSET2 slot contains offset of element's dope vector (if there is one) from the start of the DSA

STATIC EXTERNAL and Parameter Structures

Major structure entry: OFFSET1 slot contains offset of address slot from start of data region. OFFSET2 slot contains size of EXTERNAL control section. (Offset of major structure dope vector = 0.)

Minor structure entries: OFFSET1 slot not used. OFFSET2 slot contains offset of structure's dope vector from start of major structure dope vector.

Basic elements: OFFSET1 slot not used. OFFSET2 slot contains offset of element's dope vector from the start of the EXTERNAL control section

CONTROLLED Structures

Major and minor structures: OFFSET1 slot not used. OFFSET2 slot contains offset of structure dope vector from point to which pseudo register points. (In the case of the major structure, this value will be zero.)

Basic elements: OFFSET1 slot not used. OFFSET2 slot contains offset of element's dope vector relative to address in pseudo-register.

Non-Structured Arrays in STATIC INTERNAL

OFFSET1 slot contains offset of vertical origin of the array relative to start of data region. OFFSET2 slot contains offset of dope vector (if there is one) from the start of the data region.

Non-Structured Arrays in AUTOMATIC

Constant dimensions: as for STATIC INTERNAL

Adjustable dimensions: OFFSET1 slot not used. OFFSET2 slot contains offset of dope vector from start of data region.

STATIC EXTERNAL, CONTROLLED or Parameter Array

OFFSET1 slot contains offset of address slot which contains a pointer to the arrays dope vector. (Not used in the case of CONTROLLED.) OFFSET2 slot is not present.

Non-Structured Scalar Strings in STATIC INTERNAL

OFFSET1 slot contains offset of datum from start of data region. OFFSET2 slot contains offset of dope vector (if there is one) from start of data region.

Non-Structured Scalar Strings in AUTOMATIC

Constant length: as for STATIC INTERNAL

Adjustable length: OFFSET1 slot not used. OFFSET2 slot contains offset of dope vector from start of data region.

Non-Structured Scalar Strings in STATIC EXTERNAL, CONTROLLED or Parameter

OFFSET1 slot contains offset of address slot which points to string dope vector (not used in the case of CONTROLLED). OFFSET2 slot not present.

Non-Structured Non-String Scalars in AUTOMATIC or STATIC INTERNAL

OFFSET1 slot contains offset of datum from start of data region. OFFSET2 slot not present.

Non-Structured Non-String Scalars in STATIC EXTERNAL, CONTROLLED or Parameter

OFFSET1 slot contains offset of address slot which points to datum (not used in the case of CONTROLLED). OFFSET2 slot not present.

7. OTHER DICTIONARY ENTRIES

Label Constants - Extracted by the Read-In Phase

Byte Number	Description
1	Code byte X'00'
2-3	Length up to BCD length-1 byte
4-5	Hash chain - STATIC chain
6-8	Offset
9-10	Statement number (except when the label is mentioned in an ON CHECK list, in which case it gives the chain to the label BCD dictionary entry, code byte X'CE')
11	Other 1 Code Byte (See "First Code Byte - Other 1" in Section 5 above)
12-14	Second Offset Slot
15-16	Spare for Final Assembly
17	Level
18	Count
19	Count of Containing Block
20	BCD Length-1
21 etc.	BCD

Compiler Labels

The format is identical to that of a label constant, except for the omission of the BCD. The code byte is X'C3'.

Formal parameter type 1 entry

These entries are derived from the PROCEDURE and ENTRY statements, and do not contain any information other than that the identifier is a formal parameter. The format is as follows:

Byte Number	Description
1	Code byte X'40'
2-3	Length
4-5	Hash chain
6-7	These bytes will point to a full description of the identifier after Phase EK, or Phase FA, or Phase FE. These full descriptions are dictionary entries for the type of item they are describing. They do not contain the BCD of the identifier, but in the slot for the hash chain there is the dictionary reference of the corresponding formal parameter type 1 entry.
8	Level
9	Count
10	BCD length-1
11	BCD

For a description of the types of entry pointed to, see "Dictionary entry for parameter descriptions."

Dictionary entry for FILE

For attributes specified in OPEN statement the format is as follows:

Byte Number	Description
1	Code byte X'98'
2-3	Length
4-5	STATIC chain
6-8	OFFSET1
9-10	DECLARE statement number
11 onwards	String of second level markers (without preceding 'C8' code bytes) one for each attribute other than FILE, TITLE and IDENT.

This entry is created by the read-in phase and is referred to only as the argument of an ATTRIBUTES marker.

FILE Constants

Code X'08' is used for file constant entries, which have the following format:

Byte Number	Description
1	Code byte X'08'
2-3	Length
4-5	Hash chain, subsequently EXTERNAL or STATIC chain depending on whether FILE is EXTERNAL or INTERNAL
6-8	OFFSET1 (STATIC or transfer vector offset)
9-10	Declare statement number

11-12	Dictionary reference of attributes entry (zero if none)
13	Code byte (similar to the "other 2" code byte. Only internal/ external bit used)
14-15	Dictionary reference of environment string (zero if none)
16	Level
17	Count
18	BCD length-1
19 onward	BCD

FILE Parameters and Temporaries

Code X'89' is used for file parameters and for file temporaries. The format of the entry will be the same as that for label variables.

FILE Environment Entries

Code X'C8' is used for the environment string.

Byte Number	Description
1	Code byte X'C8'
2-3	Length
4 onwards	Internally coded form of argument of ENVIRONMENT option

Code X'C8' is also used for attributes collected from the DECLARE statement.

Byte Number	Description
1	Code byte X'C8'
2-3	Length
4 onwards	String of second level markers (without preceding code bytes X'C8'), one for each attribute other than FILE, ENVIRONMENT, EXTERNAL, or INTERNAL

Dictionary Entries from Constants

The format is:

Byte Number	Description
1	Code byte X'88'
2-3	Length
4-5	Hash chain
6	DATA byte
7	Data Precision*
8	Scale Factor*
	*These are the apparent precision and factor derived from the BCD of the constant (see Note 2)
9	Type (see Note 1)
10	DATA byte (2)
11	Data Precision (2)**
12	Scale Factor (2)**
	**These bytes are inserted by the phase requesting conversion. If a picture is required, these bytes are used to contain a picture table reference (see Note 3)
13-14	Dictionary reference - used when a phase requires a constant to be converted into a specific location in storage
15	BCD

Notes:

1. The type byte has the following meaning:

   First and second bits:

   00 - normal BCD constant. The first offset slot must be relocated by the storage allocation phase, to contain the offset of the converted constant from the start of STATIC storage, rather than from the start of the constants pool

   11 - the BCD is replaced by the internal form of the constant. The first offset slot is treated in the same way as for the code 00

   10 or 01 - the constant is required to be converted into a specific location in storage. The second code implies the converted constant should be made negative before being stored

   Sixth bit: 1 indicates that the constant requires a DED.

   Seventh bit: 1 indicates that the constant requires a dope vector.

   Eighth bit: 1 indicates that no conversion is required.

2. After the constants processor the bytes 6 through 8 will contain the offset of the constant from the start of the pool of constants. If a dope vector is requested then the offset of this from the start of the constants pool is eight less than that of the converted constant.

3. Should a DED be required, this will be constructed by Phase PL. The two bytes, precision(2) and scale factor(2), will contain a dictionary reference of a DED dictionary entry. If the constant requires a dope vector then Phase OS will make a dictionary entry for it, and the dictionary reference preceding the BCD will be the dictionary reference of this.

Task Identifiers and EVENT Data

The format of the dictionary entries for task identifiers and EVENT data is, apart from the initial code byte, the same as that for a label variable.

Dictionary Entries for Built-in Functions

The format is:

Byte Number	Description
1	Code byte X'04'
2-3	Length
4-5	Hash chain - later becomes the STATIC chain
6-8	Offset - gives the position in STATIC storage of the load constant for Library routine
9-10	Code bytes - the first code byte contains a value which identifies the built-in function and also provides information about it. It is used mainly by phases IM and MD-MM inclusive. The second code byte contains further information about the built-in function (See "Second Code Byte.")
11-12	DECLARE statement number
13	Level
14	Count
15	BCD length-1
16	BCD

Second Code Byte

The second code byte contains the following information:

Byte Number	Description
0	May be passed as an argument
1	May have an array as an argument
2	Must have an array as an argument
3	Is a pseudo-variable
4	Indicates to which of the two tables the offset refers
5	May have an array (or structure) as an argument, but will return a scalar result

Internal Library Functions

Library routines, other than built-in or GENERIC functions, are known as Internal Library Functions. Their dictionary entry format is as follows:

Byte Number	Description
1	Code byte X'C2'
2-3	Length
4-5	Hash chain
6-8	Offset
9	Library Code - identifies the particular Library routine required
10	Not used
11-12	Code Bytes - the first code byte contains a value used by phase MG to pick up complete information about the library function. The second code byte contains further information about the function
13	Level
14	Count

BCD Entries

BCD entries are used when the LIKE or DEFINED attributes are used. A short dictionary entry with the format given below is used. This is pointed at by the dictionary entry with the attribute.

Byte Number	Description
1	Code byte X'40'

2-3	Length
4	BCD length-1
5	BCD

Dictionary Entry for Parameter Descriptions

Dictionary entries for parameter descriptions are identical with the normal entry for data variable, label variable, structure, file, or entry points, except for the following details:

Hash chain contains pointer to formal parameter type 1. After Phase FT this pointer is moved to the bytes containing level and count

No BCD is present

No block identification is present for ENTRY or FILE

The code byte for an entry point - referred to as entry type 6 - is X'86'

ON Statements

Entries for ON statements are made by Phase FO, and contain the following:

Byte Number	Description
1	Code byte X'CD'
2-3	Length
4-5	AUTOMATIC chain
6-8	Offset
9	Code byte as supplied by the Read-In Phase
10	Block level
11	Block count
12	n
13 onwards	n dictionary references of variables or ON condition entries

ON Condition

This entry is made by Phase FO:

Byte Number	Description
1	Code byte X'4D'
2-3	Length
4-5	Hash chain later used as AUTOMATIC chain
6-8	Offset
9	Code byte as supplied by the read in phase
10	Block level
11	Block count
12	BCD length-1
13 onwards	BCD

CHECK List Entry

This entry is made by Phase FO:

Byte Number	Description
1	Code byte X'C8'
2-3	Length
4	n where n is the number of dictionary references following
5 onwards	Dictionary references (2n bytes)

PICTURE Entry

The format of an entry in the picture table in the dictionary.

Byte Number	Description
1	Code byte X'C8'
2-3	Length = L+13
4-5	Contains address of next entry in picture chain
6-8	Usage (1) (Before Phase FQ) Dictionary reference of associated declare or format statement, right adjusted Usage (11) Offset in STATIC storage
9	Code Byte (after Phase FQ) (See Code Byte description)
10	P - the number of digit positions in field in numeric picture.
11	Q - the number of digit positions after V character in numeric picture. Code X'80' represents 0, X'7F' represents -1, and X'81' represents +1.
12	W - apparent length of picture. - length of picture following. (For a non-numeric picture the

length is obtained in bytes 12-13.)

14 onwards	Picture.

Byte 9 - Code Byte

Bit Number	Description
0	0 string 1 numeric
1	0 correct 1 error
2	0 not sterling 1 sterling
3	0 short 1 long
4	Not used
5	0 decimal 1 binary
6	0 fixed 1 floating
7	Not used

Dictionary Entry for Workspace Requirement

The format for a dictionary entry for workspace requirement is:

Byte Number	Description
1	Code byte X'C8' or X'CA'
2-3	Length = 8
4-5	Total workspace required
6-8	Offset

If the code byte is C8 this is the temporary workspace used by pseudo-code (temporary type 1).

Dictionary Entry for Parameter Lists

Dictionary entries for parameter lists have the following format:

Byte Number	Description
1	Code byte X'C5'
2-3	Length
4-5	STATIC chain
6-8	STATIC offset
9-10	Assembled length
11 onwards	Contains DCA's

Dictionary Entries for Dope Vector Skeletons

Byte Number	Description
1	Code byte X'C6'
2-3	Length
4-5	STATIC chain
6-8	Offset in STATIC
9-10	Dictionary reference or DECLARE number
11 onwards	Bit pattern of skeleton dope vector

This entry is constructed by Phase PD

Symbol Table Entry

Symbol table entries are made by Phase PL.

Byte Number	Description
1	Code byte X'C7'
2-3	Length
4-5	STATIC chain
6-8	Offset in STATIC of DED
9-11	Actual DED if not pictured. If a picture is involved, the last two bytes are the dictionary reference of the picture table entry
12-13	Offset in STATIC storage of symbol table entry
14-15	Dictionary reference of next item in the symbol table for this block
16-17	Dictionary reference of item requiring entry in symbol table

Dictionary Entry for AUTOMATIC Chain Delimiter

An entry for AUTOMATIC chain delimiter is made by Phase PP.

Byte Number	Description
1	Code byte X'CC'
2-3	Length

4-5	AUTOMATIC chain
6-7	Pointer to first second file entry
8-9	Pointer to second second file entry

DED Dictionary Entry

An entry for a DED is created by Phase PL.

Byte Number	Description
1	Code byte X'C7'
2-3	Length
4-5	STATIC or AUTOMATIC chain
6-8	Offset
9-10	Dictionary reference of variable
11-18	Eight bytes of RDV text
19-20	DECLARE number

The LONG bit X'10' of byte 9 is set to 1 for halfword binary and 0 for fullword binary.

If the DED requires a picture, the last two bytes contain the dictionary reference of the picture table entry.

This entry has the same format as the first eleven bytes of a symbol table entry. No item will require both types of entry. The type required will be chained from the symbol slot in an item.

DED2 Entries

These entries are generated when a DED is required for the conversion of a temporary result.

Byte Number	Description
1	Code byte X'C1'
2-3	Length = 11
4-5	STATIC chain
6-8	Offset
9-11	Actual DED

Dictionary Entry for FED - Format Element Descriptor

The entry for a FED is made by Phase NV.

The entry is identical with a DED2 entry but with a length of 12, instead of 11. The storage allocated will be word-aligned.

Byte Number	Description
1	Code byte X'C1'
2-3	Length = 12
4-5	STATIC chain
6-8	STATIC offset
9-12	Actual FED

Label BCD Entries

Label BCD entries are made by Phase FO.

Byte Number	Description
1	Code byte X'CE'
2-3	Length
4-5	DECLARE number
6-7	Offset of the BCD in STATIC
8-9	Dictionary reference of item requiring BCD

These entries are constructed when a statement label or a PROCEDURE or ENTRY label is mentioned in an ON CHECK list. These entries are also made for EVENT and TASK variables. Phase PD will allocate storage in STATIC for the BCD of the label, and place the offset of this in the above entry.

Dope Vector Entries for Temporaries

This entry is constructed to indicate that a dope vector is required for a temporary result. At this stage the bytes in the entry contain the following:

Byte Number	Description
1	Code byte X'C0'
2-3	Length
4-5	AUTOMATIC chain
6-8	Offset in the temporary type 1 stack. After Phase QJ this will contain the offset from the start of the DSA
9-10	Dictionary reference of dope vector skeleton entry
11-12	Length of string

Record Dope Vector Entry

This entry is constructed when a variable requires a record dope vector.

Byte Number	Description
1	Code byte X'C29'
2-3	Length
4-5	STATIC or AUTOMATIC chain
6-8	Offset
9-10	Dictionary reference of variable
11-18	Eight bytes of RDV text
19-20	DECLARE number

Dope Vector Descriptor Entry

This entry is constructed for a structure which requires a dope vector descriptor.

Byte Number	Description
1	Code byte X'CC'
2-3	Length
4-5	STATIC chain
6-8	Offset
9-10	Dictionary reference of structure
11-12	Chain to RDV entry or DECLARE number
13...	DVD text set up by Phase JK

Format of a Second File Dictionary Entry

Byte Number	Description
1	Code byte X'C8'
2-3	Length of entry
4-5	Statement number of the DECLARE or other statement giving rise to the second file statement
6-7	Dictionary reference of the entry type 1 of the block from which the second file statement was extracted
8-9	Dictionary reference of a three-byte slot in the dictionary.
10	Type of second file statement, i.e., the function it performs. This is the second byte of the dictionary reference used to designate the function in the actual second file statement

Dictionary Entry for a STATIC DSA

This entry is made by phase PA (whenever a block has its DSA in STATIC). The "size of DSA" slot (right-hand two bytes) in the Entry Type 1 is used to contain the dictionary reference in this entry.

Byte Number	Description
1	Code byte X'C8'
2-3	Length = 20
4-5	Dictionary reference of the next STATIC DSA entry, or zero if this is the last entry.
6-7	Offset address slot in STATIC for the DSA (set by phase PD).
8-10	Size of DSA (set by PT and amended by RF).
11-12	Dictionary reference of the Entry Type 1 of the block.
13	Code byte to be put into first byte of DSA.
14-16,	Offset of start of DSA in STATIC (set by phase TO).
17-18	Head of block's automatic chain.
19-20	Not used

Dictionary Entry for an Error Message

Byte Number	Description
1	Code byte X'C8'
2-3	Length
4-5	Chain
6-7	Messages number
8	Flags: Bit 0 on if text to be inserted in message 1 on if statement number to be inserted 2 on if a numeric parameter to be inserted 3 on if a dictionary reference to be inserted 4-7 Severity: X'0' Termination X'4' Severe X'8' Error X'C' Warning
9	Variable information in the format shown below:

2 bytes Statement number (if present)
2 bytes Numeric parameter (if present)
2 bytes Dictionary reference (if present)

If text is to be inserted it is contained in a second dictionary entry immediately following the main entry for the message. The format of this second entry is:

Byte Number	Description
1	Code byte X'C8'
2-3	Length
4	Flag: X'00' Text that follows has not been truncated and is 10 bytes or less X'01' Text that follows is not to be truncated and is greater than 10 bytes X'FF' Text that follows has been truncated and is to be printed within quotes.
5	Variable length of text (in compiler internal code).

8. DIMENSION TABLE

Each entry containing dimension information will result in a table being set up. This table is shown in Figure 13.

9. DICTIONARY ENTRIES FOR INITIAL VALUES

The declaration of a variable with an INITIAL attribute produces these entries:

An INITIAL dictionary entry

and

One or more dictionary entries for constants

and perhaps

A second File Statement for any iteration expression contained in the INITIAL specification.

	Code Byte C8	Two-byte length	Flag Byte
	Zero byte	No. of dimensions (n)	Two-byte chain address
	VIRTUAL ORIGIN WORD		
	One-byte marker	Not used	Lower bound (halfword)
	One-byte marker	Not used	Upper bound (halfword)
			nth upper bound
n multipliers			

Note: The one-byte marker is:

00 if bound is fixed point constant; bound is a two-byte binary constant, right-adjusted.

FF if bound is an expression; bound is a three-byte pointer to a second file statement in text.

7F if the bound is inherited and has an MTF function.

3F if the bound is inherited and is covered by a previous MTF function.

FO if the bound is specified by an *.

Figure 13. Dimension Table

The INITIAL dictionary entry contains pointers to the constant entries and any Second File Statements, and is of the following format:

Byte Number	Description
1	Code byte X'C8'
2-3	Length of entry
4	Prefix options byte
5	INITIAL code byte X '79'
6	Left parenthesis
7 onwards	INITIAL value list (see below)
Final	Right parenthesis

INITIAL Value List

The INITIAL value list contains references to Second File Statements and dictionary entries which are created to correspond to the value in the input text.

The list contains the following code bytes to identify each associated dictionary reference:

X'F1' Constant iteration factor. This is followed by X'00' and the dictionary reference of the constant iteration factor.

X'F3' INITIAL value item. This is followed by X'00' and the dictionary reference of the constant. (The BCD of the constant is expanded by any imposed string replication factor).

X'F5' EOB marker. This is followed by X'00' and the dictionary reference of the next entry on the chain. (This will occur when the scratch core storage allocated for building the entry is not sufficient, and a chain of entries is constructed).

X'F7' Variable iteration factor. This is followed by the text reference of the Second File Statement containing the expression.

INTERNAL FORMATS OF TEXT

The following is a description of the internal formats of text at various points during the compilation of a source program. It is organized in this manner:

1. Text Code Bytes after the Read-In Phase
2. Text Formats after the Read-In Phase
3. Text Code Bytes on Entry to the Translator Phases
4. Format of Triples
5. Text Code Bytes in Pseudo-Code
6. Text Formats in Pseudo-Code
7. Text Formats in Absolute Code
8. Second File Statements, and the Formats of Compiler Functions and Pseudo-Variables
9. Pseudo-Code Phase Temporary Result Descriptors (TMPDs)
10. Library Calling Sequences
11. Descriptions of Terms and Abbreviations used in Text During a Compilation

Note: The internal formats of text during compile-time processing are described in Appendix F.

1. TEXT CODE BYTE AFTER THE READ-IN PHASE

First Level Table (00 to 7F)

	0	1	2	3	4	5	6	7
0	0	@	#	$	BLANK			
1	1	A	J		,	{	DO EQUALS	}
2	2	B	K	S	'		--->	
3	3	C	L	T	(			-
4	4	D	M	U				C '
5	5	E	N	V	)		<= ¬>	+
6	6	F	O	W	.			
7	7	G	P	X	ASSIGN	MULTIPLE ASSIGN	>= ¬<	/
8	8	H	Q	Y	:			REFER
9	9	I	R	Z	||		¬=	*
A	_			PSEUDO-VARIABLE	%			'
B					|		=	PREFIX -
C				FUNCTION				↑'
D					&		>	PREFIX +
	SUBSCRIPT					LITERAL CONSTANT		↑'
F					¬		<	**

|<-Digits->|<-----Letters----->|<--------------Operators-------------->|

First Level Table (80 to FF)

	8	9	A	B	C	D	E	F
0	TO	LINE	A	HYBRID QUAL		SN		FL DEC IMAG
1	ALLOCATE		CALL	ENTRY		ASSIGN BY NAME		FL DEC REAL
2	BY		B			SL		FL BIN IMAG
3	FREE		RETURN	PROC		SL'	ON PROC	FL BIN REAL
4	WHILE		P	CHECK		CN		FIX DEC IMAG
5		DISPLAY	GOOB	BEGIN		GET		FIX DEC REAL
6	SNAP	COL	R			CL		FIX BIN IMAG
7		SIGNAL	GO TO	ITDO	WRITE	PUT	END DO	FIX BIN REAL
8	SYSTEM	E		NO CHECK	2nd LEVEL MARKER		END ITDO	INTEGER
9	WAIT	REVERT		DO	READ	UNLOCK	END	STG DEC REAL
A	THEN	F		DATA LIST DO				
B	DELAY		INIT LABEL	IF	LOCATE	REWRITE	END PROG	ON
C	CONTROL VARIABLE			SN2				ARRAY CROSS SECTION
D	EXIT	NULL	DECLARE	ELSE	DELETE	OPEN	END BLOCK	CHAR CONSTANT
E		C	X	NO SNAP				iSUB
F	STOP	ASSIGN		FORMAT		CLOSE	;	BIT CONSTANT

Second Level Table (00 to 7F) (Preceded by Second Level Marker Byte C8)

	0	1	2	3	4	5	6	7
0		FILE			DECIMAL	OPTIONS	EXTERNAL	AREA
1					BINARY	IRREDUCIBLE	INTERNAL	POINTER
2		LIST			FLOAT	REDUCIBLE	AUTOMATIC	EVENT
3		EDIT	EVENT[1]		FIXED	RECURSIVE	STATIC	TASK
4	TITLE	DATA	PRIORITY		REAL	ABNORMAL[2]	CONTROLLED	CELL
5	ATTRIBUTES	STRING	REPLY		COMPLEX	NORMAL[2]	SECONDARY	BASED
6	PAGESIZE	SKIP			PRECISION 1	USES[2]		OFFSET
7	IDENT	LINE			PRECISION 2	SETS[2]		
8	LINESIZE	PAGE			VARYING	ENTRY	INITVAR 1	
9		COPY			PICTURE(NUM)	GENERIC	INITIAL	INITVAR 2
A	INTO	KEYTO			BIT ATTRIBUTE	BUILTIN	LIKE	
B	FROM	TASKOP			CHAR ATTRIBUTE		DEFINED	
C	SET		IN		DIMS(INTEGERS)		ALIGNED	
D	KEY				LABEL	ORDER	UNALIGNED	
E	NOLOCK	KEYFROM				REORDER	PACKED[2]	
F	IGNORE	FORMAT LIST		BY NAME	DIMS (NON-INTEGER)	RETURNS	POS	PICTURE (CHAR)

[1]The EVENT built-in function and pseudo-variable are known externally by the equivalent name COMPLETION.
[2]Obsolete attribute. The second level marker is used only to ensure correct warning message transmission in Read-In, and does not appear in text at any time.

Second Level Table (80 to FF)

	8	9	A	B	C	D	E	F
0	BUFFERED			MAIN		OVERFLOW	CONVERSION	CONDITION
1	UNBUFFERED							
2	EXCLUSIVE			REENTRANT		UNDERFLOW	STRING-RANGE	NAME
3	KEYED							
4	STREAM			SECONDARY		ZERODIVIDE	AREA	TRANSMIT
5	RECORD						PENDING	
6	BACKWARDS			TASK		FIXED OVERFLOW	ENDFILE	CHECK
7	SEQUENTIAL							
8	DIRECT			ON-BLOCK		SUBSCRIPT RANGE	ON RECORD	
9	PRINT							
A	ENVIRONMENT					ERROR	END PAGE	
B	INPUT							
C	OUTPUT					FINISH	KEY	NOCHECK
D	UPDATE							
E	TRANSIENT					SIZE	UNDEFINED FILE	
F								

2. TEXT FORMATS AFTER THE READ-IN PHASE

In the following statement formats the code bytes SN, SL, SL', POS, and OB have the following meanings:

SN statement number

SL statement label

SL' initial label

POS following SN is a 2-byte statement number
following SL is a 2-byte dictionary reference of statement label or entry type 1

OB prefix options byte, specifying ON conditions enabled for the statement as follows:

BIT	ON CONDITION
0	OVERFLOW
1	UNDERFLOW
2	ZERODIVIDE
3	FIXEDOVERFLOW
4	SUBSCRIPTRANGE
5	SIZE
6	CONVERSION
7	STRINGRANGE

The abbreviation SQUID means an identifier, possibly subscripted and/or qualified.

PROCEDURE Statement

The format of a PROCEDURE statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	PROCEDURE
6	Block level
7	Block count
8-10	PROCEDURE-BEGIN chain
11-13	DECLARE chain
14-16	ENTRY chain
17	Left parenthesis - optional
18...	Format parameter list - optional
	Right parenthesis - optional
	Attribute marker - optional
	Attribute code - optional
	Attribute list - optional
	Statement terminating semicolon

ENTRY Statement

The format of an ENTRY statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	ENTRY
6-8	ENTRY chain
9	Block level
10	Block count
11	Left parenthesis - optional
12...	Formal parameter list - optional
	Right parenthesis - optional
	Attribute marker - optional
	Attribute code - optional
	Attribute List - optional
	Statement terminating semicolon

BEGIN Statement

The format of a BEGIN statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	BEGIN
6	Block level
7	Block count
8-10	PROCEDURE-BEGIN chain
11-13	DECLARE chain
14	Statement terminating semicolon

END Statement

The format of an END statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	END1, END2, or END3 - END1 ends a PROCEDURE or BEGIN block; END2 ends an iterative DO block; END3 ends a non-iterative DO block
6	Block level for the containing block
7	Block count for the containing block
8	Statement terminating semicolon

IF Statement

The format of an IF statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS

4	OB
5	IF
6...	Expression
	THEN
	Statement or Group
	ELSE - optional
	Statement or Group optional

Note: The semicolon preceding the ELSE has been deleted.

DO Statement

The format of a DO statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	DO or ITDO
6	CV
7	BKC
8...	Squid
	DO equals
	Expression
	TO
	Expression
	BY
	Expression
	WHILE
	Expression
	Statement terminating semicolon

ON Statement

The ON statement takes one of the following formats:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	ON
6	ON Condition
7	SNAP or NOSNAP
8	Statement or block

-or-

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	ON
6	ON Condition
7	System
8	SNAP or NOSNAP

ASSIGN Statement

The format of the ASSIGN statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	ASSIGN or ASSIGN BY NAME
6...	Squid
	Comma - optional, may be repeated
	Squid - optional, may be repeated
	Variable number of bytes - optional, may be repeated
	ASSIGN
	Expression
	Statement terminating semicolon

WAIT Statement

The WAIT statement has the following format:

Byte Number	Description
1	Code byte SN or SL

2-3	POS
4	OB
5	WAIT
6	Left parenthesis
7...	Identifier
	Left parenthesis - optional
	Expression - optional
	Right parenthesis - optional
	Comma
	Further optional parentheses and expressions
	Right parenthesis
	Left parenthesis - optional
	Expression - optional
	Right parenthesis - optional
	Statement terminating semicolon

CALL Statement

The CALL statement has the following format:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	CALL
6-8	CALL chain
9	Identifier
10	Left parenthesis
11	Expression
12...	Right parenthesis
	Left parenthesis
	Argument List
	Right parenthesis
	Statement terminating semicolon

GO TO Statement

The format of the GO TO statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	GO TO
6...	Squid
	Statement terminating semicolon

SIGNAL and REVERT Statements

The SIGNAL and REVERT statements have the following format:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	SIGNAL or REVERT
6	ON Condition
7	Statement terminating semicolon

DISPLAY Statement

The format of the DISPLAY statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	DISPLAY
6	Left parenthesis
7...	Expression
	Right parenthesis
	Left parenthesis - optional
	Squid - optional
	Right parenthesis - optional

Statement terminating semicolon

DELAY Statement

The format of the DELAY statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	DELAY
6	Left parenthesis
7...	Expression
	Right parenthesis
	Statement terminating semicolon

RETURN Statement

The format of the RETURN statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	RETURN
6	Left parenthesis - optional
7...	Expression - optional
	Right parenthesis - optional
	Statement terminating semicolon

STOP, EXIT, and Null Statements

The format of STOP, EXIT and Null statements is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	Statement identifier
6	Statement terminating semicolon

INITIAL Label DECLARE Statements

The format of INITIAL label DECLARE statements is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	INITIAL Label DECLARE
6-8	DECLARE chain
9...	INITIAL label
	Statement terminating semicolon

DECLARE and ALLOCATE Statements

The format of DECLARE and ALLOCATE statements is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	DECLARE or ALLOCATE
6-8	DECLARE chain or ALLOCATE chain
9...	Declaration list
	Statement terminating semicolon

FORMAT Statements

The format of the FORMAT statement is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	FORMAT
6...	Format list
	Statement terminating semicolon

Format items are replaced by one-byte codes.

OPEN and CLOSE Statements

The format of OPEN and CLOSE statements follows.

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	OPEN or CLOSE
6...	File group list
	Statement terminating semicolon

READ, WRITE, GET, PUT, REWRITE, UNLOCK, and DELETE Statements

The format of READ, WRITE, GET, PUT, REWRITE, UNLOCK, and DELETE statements is as follows:

Byte Number	Description
1	Code byte SN or SL
2-3	POS
4	OB
5	Statement identifier
6...	Option list Statement terminating semicolon

3. TEXT CODE BYTES ON ENTRY TO THE TRANSLATOR PHASES

	0	1	2	3	4	5	6	7
0	DICT. REF.	FILE		COMPILER FUNCTION		FILE'		COMPILER FUNCTION'
1					COMMA	{	DO EQUALS	}
2		LIST		COMPILER FUNCTION CALL	FCOMMA	LIST'		COMPILER FUNCTION CALL'
3		EDIT	EVENT		(	EDIT'		-
4	TITLE	DATA	PRIORITY	COMPILER PSEUDOVAR	COMPILER FUNCTION COMMA	DATA'		COMPILER PSEUDOVAR'
5	ATTRIBUTES	STRING	REPLY		)	STRING'	≤	+
6	PAGESIZE	SKIP		ERROR	COMPILER ASSIGN			NDX
7	IDENT	LINE	BUY CHAMELEON	BUY ASSIGN	ASSIGN	MULTIPLE ASSIGN	≥ ¬<	/
8	LINESIZE	PAGE		ARCO	DROP	TMPD	LEFT	OFS
9		COPY			||	LD	¬=	*
A	INTO	KEYTO		PSEUDOVAR	BUYB CALSEQ	TT		PSEUDOVAR'
B	FROM	TASK	LIST MARK	END LIST MARK	|	JMP	=	PREFIX -
C	SET	RPL	IN	FUNCTION	CNVA	RPL'		FUNCTION'
D	KEY			ARGUMENT MARK	&		>	PREFIX +
E	NOLOCK	KEYFROM	DEFINED SUBSCRIPT	SUBSCRIPT	CNVB	LITERAL CONSTANT	DEFINED SUBSCRIPT'	SUBSCRIPT'
F	IGNORE	FORMAT LIST			¬	FORMAT LIST'	<	**

	8	9	A	B	C	D	E	F
0	TO	LINE	A		TO'	SN		
1	ALLOCATE		CALL				CALL'	EIO
2	BY	PAGE	B		BY'	SL		
3	FREE		RETURN	PROC				PROC'
4	WHILE	SKIP	P		WHILE'	CN	P'	
5		DISPLAY	GOOB	BEGIN	SORT	GET		BEGIN'
6	SNAP	COL	R		SNAP'	CL		
7		SIGNAL	GOTO	ITDO	WRITE	PUT	END DO	ITDO'
8	SYSTEM	E			SYSTEM'	E'	END ITDO	
9	WAIT	REVERT		DO	READ	UNLOCK	END	DO'
A	THEN	F	G			F'	G'	
B	DELAY			IF	LOCATED	REWRITE	END PROG	IF' OR ON
C	CV		SELL	SN2	CV'			ARRAY CROSS SECTION
D	EXIT	NULL	BUY	ELSE	DELETE	OPEN	END BLOCK	
E		C	X	NOSNAP		C'	END PROG 2	NOSNAP'
F	STOP	ASSIGN	BUYS	FORMAT		CLOSE	;	FORMAT

4. FORMAT OF TRIPLES

The triples produced as output from the translator phase each consist of five bytes, an operator followed by two two-byte fields. Each of the two-byte fields may be occupied by an operand, which may be a dictionary reference, a code byte or code bytes, or a numeric parameter. Two zero bytes in place of a dictionary reference operand imply that the operand is the result of previous operations, and that its type and location are described in a TMPD in the text.

The number of operands and the fields which they occupy depend upon the type of triple. The following table contains this information for all the triples used in the compiler.

TRIPLE TYPE	HEX CODE	FIELD 1	FIELD 2
KEYED	03	-	-
TITLE	04	-	OPERAND
ATTRIBUTES	05	-	OPERAND
PAGESIZE	06	-	OPERAND
IDENT	07	-	OPERAND
LINESIZE	08	-	OPERAND
INTO	0A	-	OPERAND
FROM	0B	-	OPERAND
KEY	0D	-	OPERAND
IGNORE	0F	-	OPERAND
FILE	10	-	OPERAND
LIST	12	-	-
EDIT	13	-	-
DATA	14	-	-
STRING	15	-	OPERAND
SKIP	16	-	OPERAND
LINE	17	-	OPERAND
PAGE	18	-	-
COPY	19	-	-
KEYTO	1A	-	OPERAND
TASK	1B	-	-
RPL	1C	-	-
IN	1D	-	OPERAND
KEYFROM	1E	-	OPERAND
FORMAT LIST	1F	-	-
UP	20	-	OPERAND
GIVING	21	-	OPERAND
DOWN	22	-	OPERAND
EVENT	23	-	OPERAND
PRIORITY	24	-	-
REPLY	25	-	OPERAND
BUY CHAMELEON	27	-	OPERAND
MSA	28	OPERAND 1	OPERAND 2
MTA	29	OPERAND 1	OPERAND 2
DEFINED SUBSCRIPT	2E	OPERAND	-
NULL-FUNCTION	2F	-	-
COMPILER FUNCTION	30	OPERAND	-
COMPILER FUNCTION CALL	32	OPERAND	-
COMPILER PSEUDO-VARIABLE	34	OPERAND	-
BUY ASSIGN	37	OPERAND 1	OPERAND 2
ARCO	38	-	-
SUBO	39	OPERAND 1	OPERAND 2
PSEUDO-VARIABLE	3A	OPERAND	-
SSUB	3B	OPERAND1	OPERAND2
FUNCTION	3C	OPERAND	-
SSB'	3D	OPERAND	-
SUBSCRIPT	3E	OPERAND	-
NOP	3F	-	-
PTCH	40	OPERAND 1	OPERAND 2
COMMA	41	-	*

*This triple may have two operands in format lists.

FUNCTION COMMA	42	-	OPERAND
COMPILER FUNCTION COMMA	44	-	OPERAND
ACT	45	OPERAND 1	OPERAND 2
COMPILER ASSIGN	46	OPERAND 1	OPERAND 2
ASSIGN	47	OPERAND 1	OPERAND 2
DROP	48	-	OPERAND
CONCATENATE	49	OPERAND 1	OPERAND 2
BUY B	4A	-	OPERAND
OR	4B	OPERAND 1	OPERAND 2
AND	4D	OPERAND 1	OPERAND 2
NOT	4F	-	OPERAND
LIST'	52	-	-
EDIT'	53	-	-
DATA'	54	-	-
STRING'	55	-	-
STMPD	56	OPERAND 1	OPERAND 2
MULTIPLE ASSIGN	57	OPERAND 1	OPERAND 2
TMPD	58	OPERAND 1	OPERAND 2
JMP	5B	OPERAND 1	OPERAND 2
RPL'	5C	-	-
LITERAL CONSTANT	5E	-	OPERAND
FORMAT LIST'	5F	-	-
UP'	60	-	-
DO EQUALS	61	OPERAND 1	OPERAND 2
DOWN'	62	-	-
ERROR	63	-	-
UPSIDE-DOWN COMMA	64	OPERAND 1	OPERAND 2
LESS/EQUAL	65	OPERAND 1	OPERAND 2
GREATER/EQUAL	67	OPERAND 1	OPERAND 2
LEFT	68	OPERAND 1	OPERAND 2
NOT EQUAL	69	OPERAND 1	OPERAND 2
EQUAL	6B	OPERAND 1	OPERAND 2
GREATER	6D	OPERAND 1	OPERAND 2

DEFINED SUBSCRIPT'	6E	OPERAND	-
LESS	6F	OPERAND 1	OPERAND 2
COMPILER FUNCTION'	70	OPERAND	-
COMPILER FUNCTION CALL'	72	OPERAND	-
MINUS	73	OPERAND 1	OPERAND 2
COMPILER PSEUDO-VARIABLE'	74	OPERAND	-
PLUS	75	OPERAND 1	OPERAND 2
COMR	76	-	OPERAND
DIVIDE	77	OPERAND 1	OPERAND 2
OFS	78	OPERAND	-
MULTIPLY	79	OPERAND 1	OPERAND 2
PSEUDO-VARIABLE'	7A	OPERAND	-
PREFIX MINUS	7B	-	OPERAND
FUNCTION'	7C	OPERAND	-
PREFIX PLUS	7D	-	OPERAND
SUBSCRIPT'	7E	OPERAND	-
EXPONENTIATE	7F	OPERAND 1	OPERAND 2
TO	80	-	-
ALLOCATE	81	-	OPERAND
BY	82	-	-
FREE	83	-	OPERAND
WHILE	84	OPERAND	-
*CV	85	-	OPERAND
SNAP	86	-	OPERAND
DELAY	8B	-	OPERAND
CV	8C	OPERAND 1	OPERAND 2
EXIT	8D	-	-
STOP	8F	-	-
LINE	90	-	OPERAND
END ALLOCATE	91	-	-
PAGE	92	-	-
SKIP	94	-	OPERAND

DISPLAY	95	-	OPERAND
COLUMN	96	-	OPERAND
SIGNAL	97	-	OPERAND
E	98	-	-
REVERT	99	-	OPERAND
F	9A	-	-
C	9E	-	-
A	A0	-	OPERAND
CALL	A1	-	OPERAND
B	A2	-	OPERAND
RETURN	A3	-	OPERAND
P	A4	-	OPERAND
GO OUT OF BLOCK	A5	-	OPERAND
R	A6	-	OPERAND
GO TO	A7	-	OPERAND
GOLN	A8	-	OPERAND
BUYT	A9	-	OPERAND
BUYX	AA	-	OPERAND 2
HSELL	AB	OPERAND 1	OPERAND 2
SELL	AC	-	OPERAND
BUY	AD	-	OPERAND
X	AE	-	OPERAND
BUYS	AF	-	OPERAND
PROC	B3	-	OPERAND
BEGIN	B5	-	OPERAND
ITERATIVE DO	B7	OPERAND	-
DO	B9	OPERAND	-
IF	BB	OPERAND 1	OPERAND 2
SN2	BC	-	OPERAND
NOSNAP	BE	-	OPERAND
FORMAT	BF	-	OPERAND
TO'	C0	-	OPERAND
BY'	C2	-	OPERAND

WHILE'	C4	OPERAND 1	OPERAND 2
WRITE	C7	-	-
READ	C9	-	-
CV'	CC	OPERAND 1	OPERAND 2
STATEMENT NUMBER	D0	OPERAND 1	OPERAND 2
CLN1	D1	-	OPERAND
STATEMENT LABEL	D2	OPERAND 1	OPERAND 2
CLN2	D3	-	OPERAND
COMPILER NUMBER	D4	-	OPERAND
GET	D5	-	-
COMPILER LABEL	D6	-	OPERAND
PUT	D7	-	-
E'	D8	-	-
UNLOCK	D9	-	-
F'	DA	-	-
REWRITE	DB	-	-
OPEN	DD	-	-
C'	DE	-	-
CLOSE	DF	-	-
CALL'	E1	-	-
P'	E4	-	-
END PROG	EB	-	-
END BLOCK	ED	-	-
END PROG 2	EE	-	-
END I/O	F1	-	-
PROC'	F3	-	OPERAND
BEGIN'	F5	-	OPERAND
ITERATIVE DO'	F7	-	OPERAND
DO'	F9	-	OPERAND
IF' OR ON	FB	-	OPERAND
PREFMT	FD	-	-
FORMAT'	FF	-	-

5. TEXT CODE BYTES IN PSEUDO-CODE

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	DCV0	OSM1	BGPE	BLBS	LCR	LCDR	LCER	LM	BCTA'	LH	LA	CLI	CLC	TR		INST
1	DCV1	OSM2	EOB	BLBS'	BCR	SPM	CLR	SLA	BC	CH	CL	MVI	MVC	TRT		MVCL
2	DCV2	ALLOC	PCC	BUYS	HER	LTR	ALR	SLDA	DCF	AH	AL	NI	MVN	PACK		
3	DCV3	DCA3	CHSM	PINS	HDR	LTER	SLR	SLDL	BCTA	SH	SL	OI	MVO	UNPK		
4	DCV4	DCA4	ADR	RWA	BCTR	LTDR		SLL	BCT	MH	STC	SSM	MVZ	IGNORE		
5	DCV8	FREE	SN3	APRM	NR	LNR	LPR	SRA	N	STH	ST	TM	NC			
6	DROP	BUY	BCIN	USNG	OR	LNER	LPER	SRDA	O	QLA	EX	XI	OC	CONV		
7	EQU	SELL	STOP	EDIT	XR	LNDR	LPDR	SRDL	X	STD	STE	LA'	XC	CONV'		
8	PROC	PROC'	BGNP	FMT LST	LR	LDR	LER	SRL	L	LD	LE	DCF2	ZAP	USSL		
9	BEGIN	BEGIN'	BGNP'	FMT	CR	CDR	CER	STM	C	CD	CE	BCT'	CP	DRPL		
A	PASS	ADV	DROB'	FMT'	AR	ADR	AER	BXH	A	AD	AE	MDRP	AP	CNVA		
B	EOP	PLBS	PLBS'	EDIT'	SR	SDR	SER	BXLE	S	SD	SE		SP	SINL		
C	EOP2	PCBS	PSLD	ERROR	MR	MDR	MER	SL1	M	MD	ME	SN2	MP	CNVC1		IGN2
D	IPRM	IPRM'	ABS	PFMT	DR	DDR	DER	SN	D	DD	DE	OSM3	DP	CNVC2		IGN4
E	EPRM	EPRM'	ABS'		SVC	AWR	AUR	CL1	IC	AW	AU	EQU'	ED	CNVC3		IGN5
F	ITDO	ITDO'	ALIGN		BALR	SWR	SUR	CN	BAL	CVB	CVD	BSW	EDMK	CNVC4		IGN8

6. TEXT FORMATS IN PSEUDO-CODE

Pseudo-code Design

Pseudo-code is essentially a symbolic representation of machine code, designed in such a way that it is possible to directly transform it into executable machine code by an assembly process.

A unit consists of a one-byte operation code followed by, normally, a two or four-byte field and on the other occasions by a variable length field. The bit pattern of the operation code indicates the type of unit which it heads.

By having most units either three or five bytes long, the scanning of pseudo-code is a fairly straightforward process.

The format of the various pseudo-code units is as follows:

Three-byte unit: this consists of a one-byte operation code followed by a two-byte literal offset, and it appears immediately after the symbolic representation of the instruction to which if refers.

Five-byte unit: there are four basic five-byte units which have the following formats.

Bytes 0	1	3
OP CODE	1 R1 0	R2 DICT REF

Bytes 0	1	3
OP CODE	R3	OFFSET

Bytes 0	1	3
OP CODE	R1	R2

Bytes 0	1	3
OP CODE	1 HEX 0	R2 DICT REF

Using these units with, if necessary, a three-byte unit, it is possible to symbolically represent any possible RR, RX, RS or SI instruction.

Variable length unit: the format of this is:

Bytes 0	1	3
OP CODE	FLAG	VARIABLE

With a specially designed variable field described by a two-byte flag, it is possible to represent any SS instruction with this unit.

The first byte of the two-byte flag indicates the format of the variable field and the second gives the length of the total unit.

RX Instructions

The following examples illustrate the basic forms of an RX instruction and the way in which they are represented in pseudo-code.

L R1,JOE

Bytes 0	1	3
L	0 R1	DR(JOE)

L R1, JOE+24

Bytes 0	1	3
L	0 R1	DR(JOE)
OSM1	24	

L R1,JOE(R3)

Bytes 0	1	3
L	0 R1	DR(JOE)
OSM2	R3	

L R1,JOE+24(R3)

Bytes 0	1	3
L	0 R1	DR(JOE)
OSM3	24	R3

Alternatively, JOE might be a base register in which case the dictionary reference would be replaced by a symbolic register. The two forms are distinguished by setting the flag bit of the first symbolic register equal to one when a base register is intended.

L R1,0(R3,R2)

Bytes 0	1	3
L	1 R1	R2
OSM2	R3	

When a branch instruction is generated which branches to a compiler generated EQU value, bit two of the second byte is set to one to indicate that the second field is in fact an EQU value.

Bytes 0	1	2	3
BC	01	HEX	EQU VALUE

Bytes 0	1	2	3
BC	1 0 0	HEX	R2 / DICT REF

RS Instructions

The following examples illustrate the basic forms of an RS instruction and the way in which they are represented in pseudo-code:

BXH R1,R2,ALPHA

Bytes 0	1	3
BXH	0 R1	DR(ALPHA)
OSM2	R2	

BXH R1,R2,ALPHA+24

Bytes 0 1 3

BXH	0 R1	DR(ALPHA)

OSM3	24	R2

Alternatively, ALPHA might be a base register in which case the dictionary reference would be replaced by a symbolic register as in the RX instruction.

SLA R1,6

Bytes 0 1 3

SLA	R1	6

RR Instructions

The following example illustrates the form of an RR instruction and the way in which it is represented in pseudo-code.

AR R1,R2

Bytes 0 1 3

AR	R1	R2

SI Instructions

The following examples illustrate the basic forms of an SI instruction and the way in which they are represented in pseudo-code:

CLI BETA,X'FF'

Bytes 0 1 3

CLI	0 FF	DR(BETA)

CLI BETA+4,X'FF'

Bytes 0 1 3

CLI	0 FF	DR(BETA)

OSM1	4

Alternatively, BETA might be a base register in which case the dictionary reference would be replaced by a symbolic register.

SS Instructions

Basically, an SS instruction consists of two base registers and a length byte. Since this does not conform to the format of other items of pseudo-code, it is necessary to represent an SS instruction with a variable length field, the length of which is specified in the second of two flag bytes immediately following the operation code.

This variable form of pseudo-code will be used to convey items of information internally between compiler phases, at the same time maintaining the items in the guise of pseudo-code.

Variable Length Item FLAG

The first bit of the FLAG indicates whether or not the unit represents a machine instruction. In the former case, the format of the instruction is:

Bytes
0 1 2 3 4 6 8 10

OP CODE	FLAG	LENGTH	F1	F2	F3	F4	F5

The format of the FLAG is:

Bit	Zero	One
0	Always zero	
1	F2=dict. ref.	F2=sym reg.
2	F3=dict. ref.	F3=sym reg.
3	F4 not present	F4 present
4	F5 not present	F5 present
5-7	Not used	

The FI field is identical to the length field in the SS machine instruction. The field contains one or two lengths which are each one less than the corresponding lengths used in Assembler Language. The F4 and F5 fields contain literal offsets.

Compiler Function (Bit 1=1)

In compiler functions, the format of the FLAG depends on the operation code. Thus:

CONV	FLAG	LENGTH	F1	F2

The format of the FLAG is:

Bits	Both Zero	Both One
0		Always one
1 and 2	F1=dict. ref.	F1=TMPD operand
3 and 4	F2=dict. ref.	F2=TMPD operand
5-7	Not used	

The FLAG in the IGNORE item does not contain any information.

The following examples illustrate the basic forms of an SS instruction and the ways in which they are represented in pseudo-code.

MVC ALPHA(7),BETA

Bytes 0	1	2	3	4
MVC	X'00'	8	6	DR(ALPHA)

6
DR(BETA)

ZAP ALPHA+3(14), BETA(6)

Bytes 0	1	2	3	4
ZAP	X'10'	10	13 5	DR(ALPHA)

6	8
DR(BETA)	3

MVC ALPHA(6), BETA+11

Bytes 0	1	2	3	4
MVC	X'08'	10	5	DR(ALPHA)

6	8
DR(BETA)	11

AP ALPHA+3(14),BETA+11(6)

Bytes 0	1	2	3	4
AP	X'18'	12	13 5	DR(ALPHA)

6	8	10
DR(BETA)	3	11

Alternatively, ALPHA and/or BETA might be base registers, in which cases, the dictionary references would be replaced by symbolic registers and the FLAG byte would be set accordingly:

MVO ALPHA(14), 11(6,R1)

Bytes 0	1	2	3	4
MVO	X'28'	10	13 5	DR(ALPHA)

6	8
R1	11

Pseudo-Code Format Between IEMTRA and IEMTRF

Fields that may hold a dictionary reference or register number have, at this time, the possibility of holding a literal offset. The presence of an offset is indicated by the first bit of the field being set to one, and earlier flags being set to 'register.'

7. TEXT FORMATS IN ABSOLUTE CODE

Where a standard set of assigned registers is to be used for a section of code, e.g. in the construction of prologues, or during the generation of addressing instructions, it is possible to generate instructions with registers in absolute code, instead of the normal pseudo-code two-byte symbolic registers (see "Text Formats in Pseudo-Code" in this section).

Sections of absolute code are preceded by ABS markers and followed by ABS' markers. The operation codes are the same as the normal pseudo-code instructions (see "Text Code Bytes in Pseudo-Code" in this section), but the instruction formats differ, as shown in the following examples:

RR Instructions

Bytes 0	1
OP CODE	R1 R2

RX Instructions

Bytes 0	1	2
OP CODE	R1 X2	B2

3
DICT REFCE

0	1
OSM2	OFFSET

SS Instructions

Bytes

0	1	2	3
OP CODE	LENGTH	B1	DICT REFCE

5	6
B2	DICT REFCE

0	1
OSM1	OFFSET

0	1
OSM2	OFFSET

RS Instructions

Shift Instructions

Bytes

0	1	2
OP CODE	R1 0	OFFSET

Other Instructions

Bytes

0	1	2
OP CODE	R1 R2	B2

3
DICT REFCE

0	1
OSM2	OFFSET

SI Instructions

Bytes

0	1	2	3
OP CODE	MASK	B1	DICT REFCE

0	1
OSM2	OFFSET

Note that the OSM1/OSM2 markers and their following offsets are all optional; note also that the OSM2 byte does not have a register following it, as in normal pseudo-code, but a literal offset.

The first bit (bit 0) of the byte containing the base is used as a flag. If this bit is a one, the following two bytes contain, in their low order position, a twelve bit offset, instead of a dictionary reference.

After Phases RA and RF all instructions in the text will be in absolute code.

8. SECOND FILE STATEMENTS, AND THE FORMATS OF COMPILER FUNCTIONS AND PSEUDO-VARIABLES

Second File Statements

Any expression occurring in an attribute must be put into a form which is acceptable to the translator phase. This means that it must look like a source statement. To comply with this, all expressions dealing with array bounds, string lengths, DEFINING, and INITIAL value iteration factors are converted into assignments to function references. These functions have a special meaning. They are not entered in the dictionary, and their dictionary references are to a region in the communications area. The pseudo-code physical phase dealing with each particular function generates in-line code instead of a function reference.

All the statements of this type are generated in the source text after the end of the original source program. They form a second program and are referred to later as the "second file."

The statements generated have the following overall format:

Byte Number	Description
1	Code byte SN2
2-3	Dictionary reference
4	Options byte
5	Statement type markers
6 onwards	Statement body

The dictionary reference is the reference of a second file dictionary entry. It is described in this section under "Internal Formats of Dictionary Entries." The options byte is that for the options operative in a prologue, i.e., no interruptions are accepted.

Array Bounds

The format of the second file statement for array bounds is as follows:

Byte Number	Description
1	Assignment statement marker
2	Code Byte X'00'

3-4	ADV code X'0002'
5	
5	Compiler pseudo-variable
6	Left parenthesis
7	Code byte X'00'
8-9	Dictionary reference of array
10	Triple operator code byte X'44'
11	Code byte X'5E'
12	Code byte X'00'
13	Code byte X'00' for lower bound, X'01' for higher bound
14	Number of the dimension whose bound is referenced
15	Right parenthesis
16	Triple operator code X'46'
17...	Expression for bounds Statement terminating semicolon

Multiplier Function

Muitiplier function statements are used to denote copying of a section of one dope vector into another. The format is:

Byte Number	Description
1	Assignment statement marker
2	Code byte X'00'
3-4	MTF code bytes X'0010'
5	Compiler call marker
6	Left parenthesis
7	Code byte X'00'
8-9	Dictionary reference 1
10	Triple operator code byte X'44'
11	Code byte X'00'
12-13	Dictionary reference 2
14	Triple operator code byte X'44'
15	Code byte X'5E'
16	Code byte X'00'
17-18	Offset 1
19	Triple operator code byte X'44'
20	Code byte X'5E'
21	Code byte X'00'
22-23	Offset 2
24	Triple operator code byte X'44'
25	Code byte X'5E'
26	Code byte X'00'
27-28	Length
29	Right parenthesis
30	Statement terminating semicolon

This statement requires the number of bytes specified by the length to be moved from the dope vector of the item at dictionary reference 2, starting at an offset of offset 2, to the dope vector of the item at dictionary reference 1, starting at an offset of offset 1.

String Length Statement

The string length statement is used to initialize the maximum length slot in a string dope vecoor. The format is:

Byte Number	Description
1	Assignment statement marker
2	Code byte X'00'
3-4	SDV code X'0004'
5	Compiler pseudo-variable
6	Left parenthesis
7	Code byte X'00'
8-9	Dictionary reference
10	Right parenthesis
11	Triple operator Code X'46"
12...	Expression Statement termination semicolon

The dictionary reference is that of the item whose dope vector is being initialized. If the expression is defining the length of a string being returned by an internal functoon, then the dictionary reference is that of the entry type 2 belonging to the label. In Figure 9 the reference is to B or C depeiding on whether the statement appeared in a PROCEDURE/ ENTRY statement, or an ENTRY attribute. If the item is a data item, an external procedure, or a formal parameter entry point, then the dictionary reference of that particular item appears in the statement.

INITIAL Value Statements

INITIAL value statements are used to initialize a vector of storage used to contain iteration factors. It is implied that the value of the expression must be converted to type integer. The format is as follows:

Byte Number	Description
1	Assignment statement marker
2	Code byte X'00'
3-4	IDV code
5	Compiler pseudo-variable
6	Left parenthesis
7	Code byte X'00'
8-9	Dictionary reference
10	Right parenthesis
11	Triple operator code X'46'
12...	Expression Statement terminating semicolon

The dictionary reference is to the item being initialized. The integer is the number of assignment statements of this type, and for this variable, that have been generated before this one.

Second File Statements for BASED and OFFSET

A statement is generated for a variable which is declared BASED with its associated pointer given, and for a variable which is an OFFSET with its associated base given. The format of the statement is similar to that of the INITIAL value statement except:

Byte Number	Description
3-4	PEXP code (BASED) X'0016' or BVEXP code (OFFSET) X'0017'
12	Expression (which must be a single dictionary reference of the associated pointer or base)

The dictionary reference in bytes 8-9 is that of the BASED or OFFSET variable.

Second File Statements for DEFINED

Second file statements are generated when an expression is associated with DEFINED, but the expression does not contain any iSUBs. The format is:

Byte Number	Description
1	Compiler assignment statement marker
2	Code byte X'00'
3-4	ADF code; X'0011' for base only X'0012' for subscripted base X'0013' for base with iSUB's
5	Pseudo-variable marker
6	Left parenthesis
7...	Base and subscript list Right parenthesis Statement terminating semicolon

9. PSEUDO-CODE PHASE TEMPORARY RESULT DESCRIPTORS (TMPD)

Temporary Description Stack

All information on temporary results is contained in this stack. Each item in the stack consists of 10 bytes. A maximum of 200 items is allowed.

Byte Number	Description
1	Flag 1: describes the addressing method contained in bytes 5 through 10. 2 bits in this byte are also used during the release of temporary results
2	Code 2 describes the radix, scale, mode, string type etc. of the temporary result. The format of this byte is identical to the similar byte in the dictionary and the DED used by the Library subroutines (see "Data Byte" in this section).
3-4	P,Q: describes the precision and scaling of arithmetic type results
5-6	BASE in one of the following forms: 1. "Reg by value" register containing the result - no index or offset is allowed. 2. "Reg by value" register containing the base address of the result stack 3. Offset from beginning of current temporary storage for results held in the temporary storage stack 4. Dictionary reference which specifies the base address of the result of a subscript calculation

7-8 NDX in one of the following forms:

1. Symbolic indexing register for BASE type 2 and 4.
2. The number of bytes required in the temporary core stack for BASE type 1

9-10 OFS: which is a literal offset to be inserted in the base address. When used with BASE type 1 the actual temporary offset is the sum of the offsets and the number of bytes required in the stack is the sum of the contents of OFS and NDX

Strings are described in the following ways:

If the string is of fixed length less than 256 bytes, it is given storage in the core stack. This type of string has a dictionary entry if it is passed to a subroutine.

If the string is of variable length or longer than 256 bytes, the storage is bought and sold when required. This type of string always has a dictionary entry.

If the string has no dictionary entry, it is described by the usual CODE bytes, the temporary core offset in BASE, and the byte length in NDX.

If the string has a dictionary entry, it is described by the usual CODE bytes and the dictionary reference IN BASE. The dictionary entry describes the location of the string which may be either the temporary area offset and size for the first type, or a BUY statement for the second type.

The 'top' of the stack is indicated by two pointers: PSTK and LSTK. PSTK points to the 'physical' top of the stack, which is the last item added. LSTK points to the 'logical' top of the stack, which is the next item to be released. The difference is necessary because the temporary storage stack may not be released in the same order as the description stack. When an item in the description stack is released, the corresponding temporary storage may not be at the top of the stack storage. As the storage stack is always released in order, the description is flagged and the LSTR is reduced by 1 item. When the corresponding temporary core is released from the top of the storage stack, the description is completely removed from the 'physical' stack.

Temporary Descriptions in Pseudo-Code

Descriptions are passed between pseudo-code phases using two or three TMPD triples, with the following formats:

TMPD	FLAG	F2	F3	F4
TMPD	F5		F6	
TMPD	F7			

1. FLAG describes the use of fields F5, F6, and F7.

2. CODE contains the data byte (describing type, radix, scale, mode, etc.)

3. F3 and F4 contain:
   a. Precision and scale factor of coded arithmetic type data
   b. String length for coded non-adjustable strings (maximum length for varying strings)
   c. Picture dictionary reference for data with picture

Bit Number	Value	Meaning
0 and 1	00	F5 contains a dictionary reference
	11	F5 contains a temporary workspace offset
	01	F5 contains symbolic register with address of item
	10	F5 contains register with value of item
2	0	F6 does not contain index register
	1	F6 contains index register
3	0	Two TMPD triples are used
	1	Three TMPD triples are used, and F7 contains an offset
4	0	Normal setting. String utility STRUT2 drops symbolic register in F5 if used for input
	1	String utility STRUT2 does not drop symbolic register
5	0	Normal setting
	1	Result of an invocation of SUBSTR or REPEAT
6	0	No SELL is required
	1	User of this description must SELL dictionary reference in F5. Set by string utilities for adjustable string result
7	0	F6 does not contain a dictionary reference
	1	F6 contains a dictionary reference

FLAG	F5	F6	Whether F7 applicable	Comments
X'00'	Dictionary reference	-	Yes	
X'02'	Dictionary reference	-	No	STRUT2 output -- must SELL dictionary ref.
X'04'	Dictionary reference	-	No	REPEAT function result.
X'05'	Dictionary reference[1]	Dictionary reference[2]	No	SUBSTR function result.
X'20'	Dictionary reference	Index register	Yes	Arithmetic subscript, or SDV for varying string subscript.
X'41'	Symbolic register	Dictionary reference	Yes	Non-adjustable fixed string subscript, with DROP in STRUT2.
X'49'	Symbolic register	Dictionary reference	Yes	Non-adjustable fixed string subscript, without DROP in STRUT2.
X'80'	Register	-	No	Item in register -- F7 cannot exist.
X'C0'	Workspace offset	-	Yes	
X'C1'	Workspace offset	Dictionary reference	Yes	SDV for adjustable fixed string subscript.
X'C5'	Workspace offset	Dictionary reference	No	SUBSTR pseudo-variable result.

<u>Notes</u> 1. Since F6 cannot be used for both an index register and a dictionary reference, bits 2 and 7 of the FLAG byte cannot both be 1.

2. Many other bit configurations in the FLAG byte are meaningful and could be used for future applications.

Figure 14. Temporary Descriptions in Pseudo-Code -- Use of TMPD Triple Fields F5 and F6

4. F5 and F6 are at present used as shown in Figure 14.

5. F7 can be used by adding X'10' to the FLAG byte in all cases which give a meaningful result (see Figure 14).

10. LIBRARY CALLING SEQUENCES

<u>Internal library routines</u> are used for such things as data type conversion, where there is no explicit reference to the routine in the PL/I source program. The arguments are handed to the routines in registers. In pseudo-code form, assigned registers are used, and special markers, IPRM and IPRM' are used to indicate the calling sequence to the register allocator phase. Internal library calls appear in pseudo-code as:

```
IPRM
L        1,(ARGUMENT1)
L        2,(ARGUMENT2)
--------------------
L        15,IHE----(Routine Name)
BALR     14,15
IPRM'
```

The second byte of the IPRM item is used as a flag byte. The settings are as follows:

Bit 0 Must be zero

Bit 1 END, or RETURN statement not in BEGIN block calling sequence

Bit 2 END statement calling sequence

<u>External library routines</u> calls correspond to explicit references to functions or I/O statements in the PL/I source program. The arguments to the routines are placed in workspace, and register 1 is set to point to the first argument. For pseudo-code form the calling sequence is preceded by an EPRM marker and followed by an EPRM' marker. Thus, the library calling sequence appears as:

```
MVC      WSP(N),(ARGUMENT1)
---------------------------
EPRM     1,WSP
L        15,IHE----(Routine Name)
BALR     14,15
EPRM'
LA       1,WSP
```

The second byte of the EPRM is used as a flag byte. The setting is as follows:

Bit 0 A calling sequence to a PL/I procedure

DESCRIPTIONS OF TERMS AND ABBREVIATIONS USED IN TEXT DURING A COMPILATION

The table in this Appendix gives first, the term or abbreviation; second, the phase in which the term is used; and third, a brief description of the meaning of the term or abbreviation. The key to the code used is:

- R After the Read-In Phase
- PS During the Pseudo-Code Phase
- T A triple or translator input code byte

Term or Abbreviation	Used In Phase	Description
A	R,T	Character string format item
ABS	PS	Indicates the start of absolute code (Section 4, Text Formats in Absolute Code)
ABS'	PS	Indicates the end of absolute code
ADI	PS	Addressing instruction (used between phases RA and RF)
ADR	PS	The two byte operand contains a register for use by final assembly for addressing branch destinations beyond 4096 bytes from the program base
ADV	PS	Used in 2nd file assignment statements to indicate that the expression has been calculated and that the following code is only concerned with assignment to the variable, or its dope vector, which is the subject of the second file statement
ALIGN	PS	Indicates that 4 byte alignment is required in the code at this point
ALLOCATE	R,T,PS	Replaces the keyword ALLOCATE
APRM	PS	Indicates the library calling sequence for VDA or controlled storage
ARCO	T,PS	Provides space to allow insertion of argument conversion triple
AREA	R	Replaces keyword AREA
ARGUMENT MARK	R	Marker used by phases GK and GP to indicate he start of a function argument
ARRAY CROSS SECTION	R,T	Replaces the PL/I '*' used to specify an array cross section
ASSIGN	R,T	Marker which precedes an assignment statement
ASSIGN BYNAME	R	Precedes an assignment statement with the BY NAME option
ATTRIBUTES	R,T	Marker which precedes a dictionary entry containing the attributes which have been specified on an OPEN or CLOSE statement
AUTOMATIC	R	Replaces the keyword AUTOMATIC
B	R,T	Bit string format item
BACKWARDS	R	Replaces keyword BACKWARDS BEGIN
BASED	R	Replaces keyword BASED
BEGIN'	T,PS	Triple which terminates the BEGIN block triples
BGPE	PS	Indicates the end of the complete prologue for a begin block
BGNP	PS	Indicates the start of code for a BEGIN block with no prologue

BGNP'	PS	Indicates the end of code for a begin block with no prologue
BIT ATTRIBUTE	R	Replaces the keyword BIT
BIT CONST	R	Marker preceding a BIT string constant
BINARY	R	Replaces the keyword BINARY
BLBS	PS	Indicates the start of the prologue for a BEGIN block
BLBS'	PS	Indicates the end of the prologue for a BEGIN block
BUFFERED	R	Replaces keyword BUFFERED
BUILTIN	R	Replaces the keyword BUILTIN
BUY	T,PS	Code byte or triple which indicates that a temporary variable is required
BUY ASSIGNMENT	T	Triple which indicates assignment to a temporary variable, and which implies that the workspace for the temporary variable must be obtained before the assignment
BUYB	T	Triple or code byte which indicates that a scalar temporary is required for an aggregate argument to a generic scalar built in function
BUY CHAMELEON	T	Marker which indicates that workspace is required for a temporary variable of chameleon data type i.e. the data type is taken from the expression assigned to the variable
BUYS	T,PS	Code byte or triple which indicates that a temporary variable is required, and that initialization code exists between this triple and the BUY triple
BY	R,T	Replaces the keyword BY
BY'	T	Triple which indicates the end of a BY expression
BY NAME	R	Replaces the keyword BY NAME
C	R,T	Complex decimal format item
C'	T	Triple which indicates the end of a C format item
CALL	R,T	CALL statement marker
CALL'	T	Triple internal to phase IA which marks the end of a CALL statement
CELL	R	Replaces the keyword CELL
CHAR ATTRIBUTE	R	Replaces the keyword CHARACTER
CHAR CONSTANT	R	Marker preceding a character string constant
CHECK	R	Replaces the keyword CHECK
CHSM	PS	A special offset marker. Used only in absolute code to indicate that the offset may require changing
CL	R,T,PS	Compiler label marker
CLN1	T	Compiler label number triple, referred to once only in the current statement
CLN2	T	Compiler label number triple, referred to at any point
CLOSE	R,T	Replaces the keyword CLOSE
CN	R,T,PS	Compiler statement number. Can precede compiler inserted statements
CNVC1,--4	PS	Convert compiler functions 1=Drop all registers

		2=Drop target register 3=Drop source register 4=Do not drop register
COL	R,T	Replaces the keyword COLUMN
COMA	T	Triple indicating an individual subscript in a subscript list
COMPLEX	R	Replaces the keyword COMPLEX
COMPILER ASSIGN	T	Code byte or triple indicating assignment
COMPILER FUNCTION	T	Code byte or triple used to indicate the start of a compiler function call argument list
COMPILER FUNCTION'	T	Triple indicating the end of a compiler function argument list
COMPILER FUNCTION CALL	T	Code byte or triple used to indicate the start of a compiler function call argument list
COMPILER FUNCTION CALL'	T	Triple indicating the end of a compiler function call argument list
COMPILER FUNCTION COMMA	T	Triple used to indicate the argument of compiler function, or Pseudo-Variable
COMPILER PSEUDO-VARIABLE'	T	Triple indicating the end of a compiler pseudo-variable argument list
COMPILER PSEUDO-VARIABLE	T	Code byte or triple used to indicate the start of a compiler pseudo-variable argument list
COMR	T	Triple indicating an individual subscript held in a register
CONDITION	R	Replaces the keyword CONDITION
CONTROLLED	R	Replaces the keyword CONTROLLED
CONTROL VARIABLE	R,T	Marker which indicates the control variable of a DO loop
CONVERSION	R	Replaces the keyword CONVERSION
COPY	R,T	Replaces the keyword COPY
CONTROL VARIABLE'	T	Triple which indicates the end of a control variable expression
DATA	R,T	Replaces the keyword DATA
DATA'	T	Triple indicating the end of a data directed I/O list
DATA LIST DO	R	Replaces the keyword DO in an iterative clause in a data list
DCF2	PS	Causes output of 'ERROR STOP' and the four bytes following in the pseudo-code item
DECIMAL	R	Replaces the keyword DECIMAL
DECLARE	R	Replaces the keyword DECLARE
DEFINED	R	Replaces the keyword DEFINED
DEFINED SUBSCRIPT	T	Marker which precedes the parenthesized iSUB subscript list of a defined array
DELAY	R,T	Replaces the keyword DELAY
DELETE	R,T	Replaces the keyword DELETE
DICTIONARY REFERENCE	T	Marker indicating that the following two bytes contain a symbolic dictionary reference
DIRECT	R	Replaces the keyword DIRECT
DISPLAY	R	Replaces the keyword DISPLAY
DO	R,T	Replaces the keyword DO, in a non-iterative DO group

DO EQUALS	R,T	Marker which replaces the PL/I'=' in the iterative DO statement (DO I=)
DROB	PS	Indicates to the register allocation phases that a base register used for addressing a controlled variable should be dropped
DROP	T	Triple used in optimization indicating the drop of an index register
DROP	PS	Indicates that a symbolic or assigned register in the operand field of the instruction is no longer required
DRPL	PS	Indicates the end of the use of a list of symbolic registers which have appeared in an USSL item
E	R,T	Floating decimal format item
EDIT	R,T	Replaces the keyword EDIT
EDIT'	T	Triple indicating the end of an edit directed I/O list
EIO	T	Code byte or triple which indicates the end of an I/O statement
ELSE	R,T	Replaces the keyword ELSE
END	R,T	Replaces the END keyword at the end of a BEGIN or PROCEDURE block
END BLOCK	R,T,	Indicates the end of a text block
END DO	R,T	Replaces the END keyword at the end of a non-iterative DO group
ENDFILE	R	Replaces the keyword ENDFILE
END ITDO	R,T	Replaces the END keyword at the end of an iterative DO loop
END LIST MARK	R	Marker used by phases GK and GP to indicate the end of a function argument list
END PROG	R,T,PS	Marks the end of program
END PROGRAM2	T,PS	Triple which marks the end of the second file text i.e. prologue initialization text, which is placed after the source program text
ENTRY	R	Replaces the keyword ENTRY
EPRM	PS	Indicates the start of an external library calling sequence (Section 4)
EPRM'	PS	Indicates the end of an external library calling sequence
EQU	PS	Indicates that the two byte operand field contains a label. The label is considered to be attached to the following pseudo-code item
EQU'	PS	As for EQU, but indicates that control may enter from a different statement.
ERROR	R	Replaces the keyword ERROR
ERROR	T	Code byte or triple which marks the position of an erroneous source statement which has been deleted
ERROR	PS	Indicates the presence of a source program error
EVENT	R,T	Replaces the keyword EVENT
EXCLUSIVE	R	Replaces keyword EXCLUSIVE
EXIT	R,T	Replaces the keyword EXIT
EXTERNAL	R	Replaces the keyword EXTERNAL

F	R,T	Fixed decimal format item
F'	T	Triple which indicates the end of an F format item
F COMMA	T	Triple used to indicate the arguments of a function or pseudo variable
FILE	R,T	Replaces the keyword FILE
FILE'	T	Triple indicating the end of a file list
FINISH	R	Replaces keyword FINISH
FIXED	R	Replaces the keyword FIXED
FIX BINARY IMAGINARY	R	Marker which precedes a fixed binary imaginary constant
FIX BINARY REAL	R	Marker which precedes a fixed binary real constant
FIX DECIMAL IMAGINARY	R	Marker which precedes a fixed decimal imaginary constant
FIX DECIMAL REAL	R	Marker which precedes a fixed decimal real constant.
FIXED OVERFLOW	R	Replaces keywords FIXED OVERFLOW
FLOAT	R	Replaces the keyword FLOAT
FLOAT BINARY IMAGINARY	R	Marker which precedes a float binary imaginary constant
FLOAT BINARY REAL	R	Marker which precedes a float binary real constant
FLOAT DECIMAL IMAGINARY	R	Marker which precedes a float decimal imaginary constant
FLOAT DECIMAL REAL	R	Marker which precedes a float decimal real constant
FORMAT	R,T	Replaces the keyword FORMAT
FORMAT'	T	Triple which marks the end of a remote format statement
FORMAT LIST	R,T	Precedes a format list
FORMAT LIST'	T	Triple indicating the end of a format list
FREE	R,T,PS	Replaces the keyword FREE
FROM	R,T	Replaces the keyword FROM
FUNCTION	T	Code byte or triple indicating the start of a function argument list
FUNCTION	R	Marker which precedes the parenthesized argument list (if present) of an entry name in a function reference or CALL statement
GENERIC	R	Replaces the keyword GENERIC
GET	R,T	Replaces the keyword GET
GOOB	R,T	GOTO out of block statement marker
GOLN	T	Indicates a branch to a label number
GOTO	R,T	GOTO in block statement marker
IDENT	R,T	Replaces the keyword IDENT
IF	R,T	Replaces the keyword IF
IF'	T	Triple which terminates an IF expression
IGN 2..8		Ignore markers used by Final Assembly when code has been made redundant. The final digit indicates length to be ignored.
IGNORE	R,T	Replaces the keyword IGNORE
IGNORE	PS	Pseudo-code item which indicates that the number of bytes appearing in the length count must be ignored

IN	R/T	Replaces the keyword IN
INITIAL	R	Replaces the keyword INITIAL
INITIAL LABEL	R	Marker which precedes elements of arrays of labelvariables which are initialized by being attached to statements
INITVAR	R	Replaces the keyword INITIAL(iteration factors)
INPUT	R	Replaces keyword INPUT
INST	PC	Defines a store generated by register allocator which may be deleted by phase TF if unused
INTEGER	R	Marker which precedes an internal binary integer constant
INTERNAL	R	Replaces the keyword INTERNAL
INTO	R,T	Replaces the keyword INTO
IPRM	PS	Indicates the end of an internal library calling sequence
ITDO	R,T,PS	Replaces the keyword DO in an iterative DO loop
ITDO'	T,PS	Triple which terminates an iterative DO expression
JMP	T	Triple indicating the presence of pseudo-code. The number of bytes of pseudo-code is specified in the first operand
KEY	R,T	Replaces the keyword KEY
KEYED	R	Replaces keyword KEYED
KEYFROM	R,T	Replaces the keyword KEYFROM
KEYTO	R,T	Replaces the keyword KEYTO
LABEL	R	Replaces the keyword LABEL
LEFT	T	Triple indicating a temporary result for a pseudo-variable
LIKE	R	Replaces the keyword LIKE
LINE	R,T	Replaces the keyword LINE
LINESIZE	R,T	Replaces the keyword LINESIZE
LIST	R,T	Replaces the keyword LIST
LIST'	T	Triple indicating the end of a list directed I/O list
LIST MARK	T	Marker used by Phases GK and GP to indicate the start of function argument list
LITERAL CONSTANT	R,T	Indicates that the following two bytes contain a fixed binary constant
LOCATE	R,T	Replaces the keyword LOCATE
MAIN	R	Replaces keyword MAIN
MDRP	PC	Defines a register which will be multiply dropped. Phase RA no-ops all DROP's for this register except the last
MULTIPLE ASSIGN	R,T	Marker indicating multiple assignment (Replaces PL/I',')
MVCL	PC	Defines a character move greater than 256 bytes. This is expanded by phase QF
NAME	R	Replaces the keyword NAME in the context of ON NAME
NEW PAGE	R	Replaces the keyword NEW PAGE
NOCHECK	R	Replaces the keyword NOCHECK
NO SNAP	R,T	Replaces the keyword NOSNAP

NOSNAP'	T	Triple which indicates the end of a NOSNAP list
NULL	R,T	Null statement marker
NULL-FUNCTION	T	Enables TMPD's to be passed in text by phases LB and LG before the evaluation phase IS
OFFSET	R	Replaces the keyword OFFSET
OPEN	R,T	Replaces the keyword OPEN
OFS	T	Triple indicating offset used in optimization of DO loops
ON	R,T	Replaces the keyword ON
OPTIONS	R	Replaces the keyword OPTIONS
ON RECORD	R	Replaces the keyword RECORD in the context ON RECORD
OSM1	PS	Indicates that the two byte operand field contains an index register
OSM2	PS	Indicates that the two byte operand field contains a literal offset
OSM3	PS	Indicates the presence of a literal offset and an index register
OUTPUT	R	Replaces keyword OUTPUT
OVERFLOW	R	Replaces keyword OVERFLOW
P	AR,T	Picture format item
P'	T	Triple which indicates the end of a P format item
PAGE	R,T	Picture format item
PAGESIZE	R,T	Replaces the keyword PAGESIZE
PASS	PS	POINTER Assignment
PCBS	PS	Indicates the end of the complete prologue for a procedure block
PCC	PS	Follows a PROC or BEGIN marker. Used to carry the prefix change byte for the block.
PFMT	PS	PICTURE format
PICTURE	R	Replaces the keyword PICTURE
PINS	PS	Indicates the prologue insertion point
PLBS	PS	Indicates the start of the prologue for a procedure block which is common to all entry points
PLBS'	PS	Indicates the end of the prologue of a procedure block which is common to all entry points
POINTER	R	Replaces the keyword POINTER
PRECISION1	R	Indicates a precision which has been written in the source program as '(10)', which may be either fixed or float
PRECISION2	R	Indicates a precision which has been written in the source program as '(5,2)' which implies fixed
PRINT	R	Replaces keyword PRINT
PRIORITY	R,T	Replaces the keyword PRIORITY
PSEUDO-VARIABLE	R	Marker which precedes the parenthesized argument list to a pseudo-variable
PSEUDO-VARIABLE	T	Code byte or triple indicating the start of a pseudo-variable argument list
PSEUDO-VARIABLE'	T	Triple indicating the end of a pseudo-variable argument list

PSLD	PS	Indicates a pseudo-code instruction for use by the final assembly listing phase
PROC	R,T PS	Replaces the keyword PROCEDURE
PROC'	T.PS	Triple which terminates the procedure block triples
PTCH	T	Patch triple. Used by optimization phase to overwrite a triple in text at a point where code is to be inserted. The code to be inserted and the overwritten triple are held in a table in text blocks
PUT	R,T	Replaces the keyword PUT
R	R,T	Remote format statement marker
READ	R,T	Replaces the keyword READ
REAL	R	Replaces the keyword REAL
RECORD	R	Replaces the keyword RECORD
RECURSIVE	R	Replaces the keyword RECURSIVE
REENTRANT	R	Replaces the keyword REENTRANT
REPLY	R,T	Replaces the keyword REPLY
RETURN	R,T	Replaces statement marker
REVERT	R,T	Replaces the keyword REVERT
REWRITE	R,T	Replaces the keyword REWRITE
RPL	T	Code byte or triple indicating the start of a format list replication factor expression
RPL'	T	Triple indicating the end of a format list replication factor expression
RWA	PS	Indication of an addressing vector for use by the register allocator when the number of symbolic registers in use exceeds the amount of work space which has been allocated
SECONDARY	R	Replaces keyword SECONDARY
SECOND LEVEL MARKER	R	A code byte which immediately precedes all code bytes appearing in the second level table
SELL	T,PS	Code byte or triple which indicates that a temporary variable is no longer required
SET	R,T	Replaces the keyword SET
SETS	R	Replaces the keyword SETS
SEQUENTIAL	R	Replaces the keyword SEQUENTIAL
SIGNAL	R,T	Replaces the keyword SIGNAL
SIZE	R	Replaces the keyword SIZE
SKIP	R,T	Replaces the keyword SKIP
SL	R,T,PS	Statement label marker. Precedes all labelled statements
SN	R,T,PS	Statement number marker. Precedes all unlabelled statements
SN2	R,T,PS	Marker which precedes a second file statement (see Section 4)
SN3	PS	Indicates the start of a second file statement which is concerned with initializing array, or structure, or string dope vectors. Similar to SN2 (Section 4) except that there is no associated entry

SNAP'	T	Triple which indicates the end of a snap list	TITLE	R,T	Replaces the keyword TITLE
STATIC	R	Replaces the keyword STATIC	TMPD	T	Triple indicating a temporary expression result
STERLING DECIMAL REAL	R	Marker which precedes a sterling decimal constant	TO	R,T	Marker replacing TO in the iterative DO statement
STOP	R,T	Replaces the keyword STOP			
STREAM	R	Replaces keyword STREAM	TO'	T	Triple which indicates the end of a TO expression
STRING	R,T	Replaces the keyword STRING			
STRING'	T	Triple indicating the end of a string list used with list directed I/O	TRANSMIT	R	Replaces the keyword TRANSMIT
			UNBUFFERED	R	Replaces the keyword UNBUFFERED
STRINGRANGE	R	Replaces the keyword STRINGRANGE	UNDEFINEDFILE	R	Replaces the keyword UNDEFINEDFILE
SUB	R	Replaces the keyword SUB used in iSUB DEFINING marker preceding a BIT	UNDERFLOW	R	Replaces keyword UNDERFLOW
			UNLOCK	R,T	Replaces the keyword UNLOCK
SUBSCRIPT	R,T	Marker which precedes the parenthesized subscript list of an array	UPDATE	R	Replaces keyword UPDATE
			USES	R	Replaces the keyword USES
SUBSCRIPT'	T	Triple indicating the end of a subscript list	USNG	PS	Indicates the presence of an assigned register
SUBSCRIPT-RANGE	R	Replaces keyword SUBSCRIPTRANGE			
SSB'	T	Supersubs prime triple. Similar to SUBS'. Used by Optimization phases in conjunction with SSUB	USSL	PS	Indicates a list of symbolic registers which need not be saved on branch and branch and link instructions
SSUB	T	Supersubs triple. Similar to SUBS. Used by Optimization phases	VARYING	R	Replaces the keyword VARYING
			WHILE	R,T	Replaces the keyword WHILE
SYSTEM	R,T	Replaces the keyword SYSTEM	WHILE'	T	Triple which indicates the end of a WHILE expression
SYSTEM'	T	Triple which indicates the end of a system list	WRITE	R,T	Replaces the keyword WRITE
TASK	R,T	Replaces the keyword TASK	X	R,T	Spacing format item
THEN	R,T	Replaces the keyword THEN	ZERODIVIDE	R	Replaces the keyword ZERODIVIDE

The communications region is an area specified by the control routines (see Appendix H), and used to communicate necessary information between the various phases of the compiler. The communications region is resident in the first dictionary block throughout the compilation.

Note: The use of the communications region during compile-time processing is described in Appendix F.

The tables below give the following information for each location of the communications region: name of location; offset (i.e., relative address); use (i.e., stages of compilation during which the location is in use); and a description of the contents. Certain locations are used in one capacity during part of the compilation, and then re-used in a different capacity during another part of the compilation. In these cases, one location will have two table entries: details of alternative usage appear in the columns headed $Name_2$, Use_2, etc.

Table 2. Communications Region (Part 1 of 2)

Name	Offset (Dec.)	Use	Description
SAVE0	0	ALL PHASES	Register save area
SAVE1	SAVE0+4	ALL PHASES	Register save area
SAVE2	SAVE0+8 ETC.	ALL PHASES	Register save area
.	.	.	.
.	.	.	.
.	.	.	.
SAVE15	SAVE0+60	ALL PHASES	Register save area
ZTV	64	ALL PHASES	Control phase base
ZTRAN1	68	ALL PHASES	External to internal translate table
ZTRAN2	ZTRAN1+4	ALL PHASES	Internal to external translate table
ZNXTD	76	ALL PHASES	Next available dictionary location
ZERRD	80	ALL PHASES	
ZERRS	ZERRD+4	ALL PHASES	First locations of error chains
ZERRW	ZERRD+8	ALL PHASES	
ZERRC	ZERRD+12	ALL PHASES	
ZDNXT	ZERRD+16	ALL PHASES	
ZSNXT	ZDNXT+4	ALL PHASES	Current ends of error chains
ZWNXT	ZDNXT+8	ALL PHASES	
ZCNXT	ZDNXT+12	ALL PHASES	
ZMYNAM	112	ALL PHASES	Name of last phase entered
DICTP	116	ALL PHASES	
ZCNCHR	118	ALL PHASES	Source column containing control character
ZPROCH	120	ALL PHASES	Chain of created procedures
ZSTAT	124	ALL PHASES	Current statement number
PAR1	128	ALL PHASES	Parameter word 1
PAR2	PAR1+4 ETC.	ALL PHASES	Parameter word 2
.	.	.	.
.	.	.	.
.	.	.	.
PAR8	PAR1+28	ALL PHASES	Parameter word 8
CORLFT	160	ALL PHASES	Amount of core left for compilation
LKNAME	164	PHASE VE	Member name of module produced by compilation
ZOBSAD	172	ALL PHASES	Address of overflow block
TERMSW	176	ALL PHASES	Compilation terminating switch
OFDNAM	178	ALL PHASES	
SPLNAM	180	ALL PHASES	Name of phase in control when spill file is opened
ZOBNUM	182	ALL PHASES	Overflow block number
SCNOP	184	ALL PHASES	Phase directory scan switch

Table 2. Communications Region (Part 2 of 2)

Name	Offset (Dec.)	Use	Description
SCCNF	185	ALL PHASES	On if in second half of compiler
ZDROLF	186	ALL PHASES	Overlay switch
AREA	187	ALL PHASES	Code word for dummy routines (phase AD)
ZM91	188	ALL PHASES	(used for Model 91 systems)
PERRSW	189	ALL PHASES	Print error switch
BERSW	190	ALL PHASES	*process error byte
IOERSW	191	ALL PHASES	I/O error switch
ZPAGE	192	ALL PHASES	Number of lines in page
ZLINE	194	ALL PHASES	Number of characters in a line
ZOPT	196	ALL PHASES	Code word of loading of optimizing phases
PARMLEN	197	ALL PHASES	Length of options field in * - process card
MAXFON	198	ALL PHASES	Number of offset slots in a dictionary block
ZDICTSP	200	ALL PHASES	Useful dictionary block size
ZNXTOF	204	ALL PHASES	Offset of next dictionary entry
FONOF	208	ALL PHASES	Offset of offset slots in dictionary block
FSTDRF	212	ALL PHASES	Dictionary reference of last diction-ary entry
ERCODE	224	ALL PHASES	Error message codes
MCSIZE	228	ALL PHASES	M/CSIZE this run
CCCODE	232	ALL PHASES	Compiler option indicators (see table 5 below)
HDR	236	ALL PHASES	Address of phase directory
TLR	240	ALL PHASES	Timer last read
TRT	244	ALL PHASES	Total run time
ARINT	248	ALL PHASES	Arithmetic interrupt
BR2	252	ALL PHASES	Second base for control phase
STARTX	256	ALL PHASES	Start of text
DICTSZ	260	ALL PHASES	Dictionary block size
TXTSZ	264	ALL PHASES	Space available in text block
RDSIZE	268	ALL PHASES	SIZE of read area
INCOD	272	ALL PHASES	Interrupt code
ARMASK	273	ALL PHASES	Arithmetic error mask
LOCK	274	ALL PHASES	Dictionary lock slot
ZNXTLC	276	ALL PHASES	End of current text
ZSHIFT	280	ALL PHASES	Number of bits in dictionary reference offset
ZMASK	284	ALL PHASES	Mask to remove block numbers
ZMASK1	ZMASK + 2	ALL PHASES	Mask to remove offset
ZSOR	288	ALL PHASES	Input record source
ZMAG	290	ALL PHASES	Input record margin
CCCODEE	292	ALL PHASES	Compiler option indicators
ZCOMM	304	ALL PHASES	

Table 3. Communications Region (Part 1 of 2)

Name	Dec. Offset	Use		Description	Name$_2$	Use$_2$		Description$_2$
		Start	End			Start	End	
ZCALLC	ZCOMM+ 0	Read in	BCD to Dict. Ref.	Start of CALL chain				
ZLABTB	+ 4	Read in	Initial	Start of label chain				
ZATTID	+8			Pointer to attribute tidy-up area	ZPCOP			
ZALLCH	+12	Read in	ALLOCATE + Attribute	Start of ALLOCATE chain				
ZFLAG1	+16	Read in	Dictionary	Flag bytes,				
ZFLAG2	+17			mainly used				
ZFLAG3	+18			for optional				
ZELAG4	+19			phase				
ZFLAG5	+20			marking (see Table 4 below)	ZSYSOT	Pseudo code	Pseudo code	Dict. Ref. SYSOUT
ZFLAG6	+21	FU	QU	Unaligned (see Table 4)				
UNUSED	+21 TO +23							
ZSCRCH	+24	PD	PL	Address of scratch core kept across phases				
UNUSED	+25 TO +27							
ZHASH	+28	Dictionary	Dictionary	Start of hash table	ZINCL	PC.	End	INCLUDE card pointer
	+32	Not used	Not used		ZEQTA3	Final	Assy.	Assigned offset table
ZFATTB	+36	Dictionary	Declare pass 2	Start of fact-ored attribute	ZLCONS ZEOCS	Strge	Alloc	Last constant in STATIC. End of STATIC
ZCDIMC	+40	Dictionary	Pre-translator	Start constant dimension	ZSMREG	Trans-lator	Pseudo code	Current sym-bolic register
Z2FILE	+44	Dictionary	End	Start of second file				
ZDLFST	+48	Dictionary	Storage allocator	Defined storage area	ZFSTEX	Strge alloc	End	First external item
ZDCBLD	+52	Dictionary	Dictionary	Dictionary build area	ZPRSIZ	Final	Assy.	Size of com-piled program
ZMPSTK	+56	Dictionary	Translator	Program map stack	ZSICSZ	Final	Assy.	STATIC INTERNAL size
ZUPIC	+60	Dictionary	Picture processor	Start of picture chain	ZSTALC	Final	Assy.	Storage loc-ation counter
ZPROC1	+64	Dictionary	End	Start of entry type 1 chain				
ZSTACH	+68	Dictionary	End	Start of STAT-IC chain (6)				
ZVDIMC	+74	Dictionary	Translator	Start of vari-able dimension chain				
ZCONCH	+78	Dictionary	ALLOCATE	Start of con-stants chain				
ZDEFCH	+80	Dictionary	Dictionary	Chain of defined items	ZCITEM	Pre trans.	End	Chain of CON-TROLLED items
ZLIKCH	+82	Dictionary	Dictionary	Chain of LIKE items	ZEQMAX	Pseudo code	End	Max. label number

Table 3. Communications Region (Part 2 of 2)

Name	Dec. Offset	Use Start	Use End	Description	$Name_2$	Use_2 Start	Use_2 End	$Description_2$
ZPOLCH	+84	Dictionary	Dictionary	Chain of POOL items	ZOSMRG	Pseudo code	Pseudo code	End of optimization symbolic registers
ZDCOM1	ZCOMM+86	Dictionary	Dictionary	Latest dict. ref.	ZCOBOL	GB	JI	Head of COBOL structure chain
ZDCOM2	+90	Dictionary	Dictionary	Flags for dictionary build interface (8 bytes)				
ZDSA	ZCOMM+94	PA	UD	Head of STATIC DSA chain				
ZCPOFF	ZCOMM +100	PD	UA	Offset of first constants pool within static internal.				
MACRON	ZCOMM +336	BW	TP	Macro reqd count (halfword)				
SOURCN	ZCOMM +338	CI	TP	Source reqd count (halfword)				
STMNTN	ZCOMM +340	CI	TP	Source stmt count (halfword)				
INSTRN	ZCOMM +342	TJ	TP	Object instn count (halfword)				
BYTESN	ZCOMM +344	TO	TP	Object byte count (fullword)				

Note: Bytes ZCOMM+60 to ZCOMM+332 are used internally within the Read-in phase and cannot be assumed to be zero at exit from that phase.
Bytes ZCOMM+336 to ZCOMM+344 are not available for re-use.
Bytes ZCOMM+104 to ZCOMM+131 are used by the generic phase IM. They are cleared before exit from the phase.

Table 4. Communications Region. Bit Usage in ZFLAGS

Byte Name	Offset	Bit (Hex)	Bit Name	Description Bits are set on, on encountering:-
ZFLAG1	ZCOMM+16	80	ZDEFFL	DEFINED attribute
		40	ZAWAFL	ALLOCATE statement
		20	ZSECFL	Second File statement
		10	ZDIMFL	Dimension attribute
		08	ZCHKFL	CHECK/NOCHECK prefix
		04	ZONFL	ON, SIGNAL or REVERT statement
		02	ZSTRFL	Structure
		01	ZDECFL	DECLARE statement
ZFLAG2	+17	80	ZLIKFL	LIKE attribute
		40	ZINTST	STATIC INITIAL
		20	ZOPCFL	OPEN/CLOSE statement
		10	ZGTPFL	GET/PUT statement
		08	ZGOTFL	GO TO statement
		04	ZTEPFL	TASK/EVENT/PRIORITY options, REPLY statement
		02	ZPICFL	PICTURE attribute/format item
		01	ZISBFL	iSUB defining
ZFLAG3	+18	80	ZCONTG	UNALIGNED(NONSTRING) attribute
		40	ZSETFL	SETS attribute
		20	ZOSSFL	DELAY, DISPLAY, WAIT statement
		10	ZARGFL	Argument list
		08	ZINLFL	INITIAL Label
		04	ZDIOFL	DATA directed I/O
		02	ZRECIO	RECORD I/O
		01	ZINTAC	AUTO/CTL initialization
ZFLAG4	+19	80	ZFREE	FREE statement
		40	STM256	More than 256 statements
		20	FILEFL	Files present
		10		Spare
		08	ZPUTFL	PUT DATA
		04	ZGETFL	GET DATA
		02	ZPTRFL	Pointer Qualifier
		01	ZRODFL	STATIC DSA Entry
ZFLAG5	+20	80	ZFTASK	TASK/EVENT/PRIORITY option on a CALL statement
		40	ZDENFL	Set by FT
		20	ALCSLM	ALLOCATE, with second level marker
		10		Spare
		08		Spare
		04		Spare
		02		Spare
		01		Spare
ZFLAG6	+21	80	ZUNAFL	ON for unaligned data: set by FU
		40		Spare
		20		Spare
		10		Spare
		08		Spare
		04		Spare
		02		Spare
		01		Spare

Table 5. Communications Region. Bit Usage in CCCODE. (Part 1 of 2)

Byte	Bit		
0	0	DUMP	1 wanted 0 not wanted
	1		1 abort has occurred
	2	LIST	1 not wanted 0 wanted
	3	LOAD	1 not wanted 0 wanted
	4	DECK	1 not wanted 0 wanted
	5	EXTREF	1 not wanted 0 wanted
	6	XREF	1 not wanted 0 wanted
	7	ATR	1 not wanted 0 wanted
1	0		1 means U-format 0 means F-format records on input
	1		1 if track overflow is present
	2		Severity code
	3		Severity code
	4		Severity code
	5		Severity code where 0000=FLAGW 0001=FLAGE 0010=FLAGS
	6	CHAR 48	1 not wanted 0 wanted
	7	MACRO	1 not wanted 0 wanted

Table 5. Communications Region. Bit Usage in CCCODE. (Part 2 of 2)

Byte	Bit		
2	0	SOURCE	1 not wanted 0 wanted
	1	CHK	1 not wanted 0 wanted
	2	BCD	1 BCD input 0 EBCDIC input
	3	SOURCE2	1 wanted 0 not wanted
	4	OPT	1 wanted 0 not wanted
	5		1 AE required
	6		1 program check has occurred
	7		1 means first record has been read
3	0	STMT	1 not wanted 0 wanted
	1	MACDCK	1 not wanted 0 wanted
	2	COMP	1 not wanted 0 wanted
	3		1 macro phase now running
	4		1 batch record found 0 batch record not found
	5		1 EOF record found 0 EOF record not found
	6		not used requirement
	7	NEST	1 wanted 0 not wanted

APPENDIX C: COMPILER OPTIONS TABLE

Control module IEMTAF consists of a control section, IEMAF, containing a bit string field of fourteen bytes in length, followed by seven fixed-point values aligned on fullword boundaries (see Figure 15). The first five fixed-point values give defaults for the compiler options LINECNT, SIZE, SORMGINL (start), SORMGINR (end), and CRGCNTL (control column), respectively. The remaining two fixed-point values are spare and not currently in use.

The SORMGINL, SORMGINR, and CRGCNTL settings of 9, 108, and 8, respectively, represent actual left and right margins of 1 and 100, and a carriage control column setting of 0. However, the first 8 bytes of each line in TSS must be reserved for a header containing the line number. Any changes in margin settings, including CRGCNTL, should take this fact into account, adding 8 to each setting desired.

Bits 0 to 25, 28 to 46, and 51 to 53 in the string are used to specify the default status of the options. Bits 54 to 102 in the string specify whether an option keyword is to be deleted (see Figure 16). A "1" in the bit string means "yes;" a "0" means "no." The remaining bits in the string are spare bits not currently in use. Figure 17 contains the PL/I defaults for TSS/360.

Note: Bits 28 through 30 are dummy settings for the syntax check option. Because the defaults for these options will differ depending upon the mode the user is in, the defaults are always set in module F1. However, bits 28 through 30 cannot be treated as spare bits (due to a routine check of them by module AB during compilation) and must always be 0.

```
IEMAF        CSECT

             DC B'  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 '     DEFAULT
             DC B'  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 '     SWITCHES
             DC B'  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 '
             DC B'  0 0 0 0 0 0|0 0  0 0 0 0 0 0 0 0 '     DELETE
             DC B'  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 '     SWITCHES
             DC B'  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 '
             DC B'  0 0 0 0 0 0 0|0  0 0 0 0 0 0 0 0 '     SPARE SWITCHES

LINECNT      DC F'50'
SIZE         DC F'999999'
SORMGINL     DC F'9'
SORMGINR     DC F'108'
CRGCNTL      DC F'0'
             DC F'0'
             DC F'0'
```

• Figure 15. The IEMAF Control Section

Bit	Parameter	Bit	Parameter	Bit	Parameter
0	ATR	38	NONEST	75	DELETE=SOURCE 2
1	NOATR	39	COMP	76	DELETE=NOSOURCE2
2	BCD	40	NOCOMP	77	DELETE=OPT
3	EBCDIC	41	M91	78	DELETE=LINECNT
4	CHAR60	42	NOM91	79	DELETE=LINELNG
5	CHAR48	43	MACDCK	80	DELETE=SIZE
6	DECK	44	NOMACDCK	81	DELETE=SORMGIN
7	NODECK	45	EXTDIC	82	Not used
8	EXTREF	46	NOEXTDIC	83	DELETE=STMT
9	NOEXTREF	47	Not used	84	DELETE=NOSTMT
10	FLAGW	48	Not used	85	DELETE=MACRO
11	FLAGE	49	Not used	86	DELETE=NOMACRO
12	FLAGS	50	Not used	87	DELETE=COMP
13	LIST	51	DEFAULT/DELETE	88	DELETE=NOCOMP
14	NOLIST		(BIT always 0)	89	DELETE=M91
15	LOAD	52	LIB=REAL	90	DELETE=NOM91
16	NOLOAD	53	LIB=COMPLEX	91	DELETE=PAGECTL
17	XREF	54	DELETE=ATR	92	DELETE=MACDCK
18	NOXREF	55	DELETE=NOATR	93	DELETE=NOMACDCK
19	SOURCE	56	DELETE=BCD	94	DELETE=EXTDIC
20	NOSOURCE	57	DELETE=EBCDIC	95	DELETE=NOEXTDIC
21	SOURCE2	58	DELETE=CHAR60	96	DELETE=OPLIST
22	NOSOURCE2	59	DELETE=CHAR48	97	DELETE=NOOPLIST
23	OPT=0	60	DELETE=DECK	98	DELETE=NEST
24	OPT=1	61	DELETE=NODECK	99	DELETE=NONEST
25	OPT=2	62	DELETE=EXTREF	100	DELETE=SYNCHKE
26	Not used	63	DELETE=NOEXTREF	101	DELETE=SYNCHKS
27	Not used	64	DELETE=FLAGW	102	DELETE=SYNCHKT
28	SYNCHKE[1]	65	DELETE=FLAGE	103	Not used
29	SYNCHKS[1]	66	DELETE=FLAGS	104	Not used
30	SYNCHKT[1]	67	DELETE=LIST	105	Not used
31	OPLIST	68	DELETE=NOLIST	106	Not used
32	NOOPLIST	69	DELETE=LOAD	107	Not used
33	STMT	70	DELETE=NOLOAD	108	Not used
34	NOSTMT	71	DELETE=XREF	109	Not used
35	MACRO	72	DELETE=NOXREF	110	Not used
36	NOMACRO	73	DELETE=SOURCE	111	Not used
37	NEST	74	DELETE=NOSOURCE		

[1]See Note on preceding page.

Figure 16. Bit Identification Table

Option	TSS Default
Fixed Values	
LINECNT	50
SIZE	999999
SORMGIN - left	9
SORMGIN - right	108
CRGCNTL	8
Alternative Options	
ATR|NOATR	NOATR
BCD|EBCDIC	EBCDIC
CHAR60|CHAR48	CHAR60
DECK|NODECK	NODECK
EXTREF|NOEXTREF	NOEXTREF
FLAGW|FLAGE|FLAGS	FLAGW
LIST|NOLIST	NOLIST
LOAD|NOLOAD	LOAD
XREF|NOXREF	NOXREF
SOURCE|NOSOURCE	SOURCE
SOURCE2|NOSOURCE2	SOURCE2
OPT=0|OPT=1|OPT=2	OPT=1
M91|NOM91	NOM91
MACDCK|NOMACDCK	MACDCK
EXTDIC|NOEXTDIC	EXTDIC
OPLIST|NOOPLIST	OPLIST
STMT|NOSTMT	NOSTMT
MACRO|NOMACRO	NOMACRO
NEST|NONEST	NONEST
Unused	
COMP|NOCOMP	COMP
LIBRARY OPTION, REAL|,COMPLEX	-
SYNCHKE|SYNCHKS|SYNCHKT	SYNCHKS (conversational) SYNCHKT (nonconversational)
Deleted Options	
M91	
NOEXTDIC	

Figure 17. PL/I Defaults-

The mechanism of dynamic storage management is described in the publication IBM System/360 Time Sharing System, PL/I Subroutine Library Program Logic Manual.

Part of the code required to implement the storage management is generated as prologue and epilogue code by the compiler. This Appendix contains annotated examples of prologues and epilogues for PROCEDURE, BEGIN, and ON blocks.

PROLOGUES AND EPILOGUES

Example in PL/I

```
A:I: PROCEDURE(X,Y);
     DECLARE Y CONTROLLED;
       .
       .
       .
     ON OVERFLOW C=0;
       .
       .
       .
B: BEGIN;
       .
       .
       .
   END;
       .
       .
       .
AB:IJK: ENTRY(Y,Z)
       .
       .
       .
     RETURN(EXPRESSION)
       .
       .
       .
   END;
```

```
A        BC    15,16,(0,15)           BRANCH ROUND FOLLOWING CONSTANTS
         DC    A11(1)                 LENGTH OF BCD
         DC    C'A'                   BCD OF ENTRY POINT
SIZDSA   DC    F' SIZE OF DSA'
STATIC   DC    A(STATIC CONTROL       ADDRESS OF STATIC INTERNAL CONTROL SECTION
               SECTION)               (ONLY COMPILED FOR EXTERNAL AND ON PROLOGUES)
         STM   14,11,12,(13)          SAVE STANDARD REGISTERS IN SAVE AREA OF
         *                            CALLER'S DSA
         LR    10,15                  SET UP FIRST PROLOGUE BASE
         BAL   8,GETDSA(0,10)         CALL ROUTINE TO GET DSA
         MVI   SWITCH(13),X'X1'       INSERT RETURN (EXPRESSION) SWITCH
*                                     (ONLY COMPILED IF THERE IS A RETURN (EXP)
*                                     AND THE ENTRY LABELS HAVE DIFFERENT DATA
*                                     ATTRIBUTES)
         BC    15,COPRAM1(0,10)       BRANCH TO COPY OVER PARAMETERS
I        BC    15,10(0,15)            BRANCH ROUND FOLLOWING CONSTANTS
         DC    AL1(1)                 LENGTH OF BCD
         DC    C'I'                   BCD OF ENTRY POINT
ADPRIM   DC    A(A)                   FIRST PROLOGUE BASE ADDRESS
         STM   14,11,12(13)           SAVE STANDARD REGISTERS IN SAVE AREA OF
*                                     CALLER'S DSA
         L     10,ADPRIM(0,15)        SET UP FIRST PROLOGUE BASE
         LA    8,IP(0,10)             SET RETURN REGISTER
```

```
GETDSA   L     11,STATIC(0,10)          SET UP STATIC DATA POINTER (ONLY IN
*                                       EXTERNAL PROCEDURES AND ON PROLOGUES)
         L     0,SIZDSA(0,10)           GR0=SIZE OF DSA
         L     15,32,(0,11)             LOAD GR15 WITH ENTRY POINT OF IHESADA
*                                       (UNLESS DSA IS IN STATIC, WHEN ENTRY POINT OF
*                                       COMPILER'S 'GET DSA' ROUTINE WILL BE LOADED)
         BALR  14,15                    CALL ROUTINE TO GET A NEW DSA
         LR    14,13                    POINT GR14 AT NEW DSA
         LA    0,7,(0,0)                SET LOOPING VALUE = 7
         SR    15,15                    CLEAR INDEXING REGISTER
LOOP     A     14,0(0,11)               BUMP GR14 BY 4096
         ST    14,ADVEC+4(15,13)        STORE GR14 IN ADDRESSING VECTOR
         LA    15,4(0,15)               BUMP INDEX REGISTER
         BCT   0,LOOP(0,10)
         BCR   15,8                     BRANCH ON RETURN REGISTER
IP       MVI   SWITCH(13),X'X2'         INSERT RETURN (EXP) SWITCH
COPRAM1  L     14,0(0,1)                PICK UP FIRST ARGUMENT ADDRESS AND
         ST    14,X(0,13)               STORE IN X IN DSA
         L     14,4(0,1)                PICK UP SECOND ARGUMENT ADDRESS
         LA    0,10(0,0)
         SR    14,0                     POINT GR14 AT PSEUDO-REGISTER OFFSET OF
         LH    14,0(0,14)               ARGUMENT AND PICK IT UP
         ST    14,Y(0,13)               STORE OFFSET IN Y IN DSA
         L     14,8(0,1)                PICK UP ADDRESS OF TARGET FIELD
         ST    14,TARGET(0,13)          AND STORE IN DSA
         L     10,A...A(0,11)           LOAD GR10 FROM TRANSFER VECTOR SLOT
*                                       FOR ENTRY POINT A IN STATIC.
         BAL   8,COMMON(0,10)           BRANCH AND LINK TO COMMON PROLOGUE
*        BCR   15,10                    BRANCH TO THE APPARENT ENTRY POINT
*                                       FOR A
COMMON   MVI   96(13),X'80'             SET DSA TASKING FLAG (ONLY COMPILED
*                                       IF TASKING IN COMPILATION)
         BALR  10,0                     SET UP COMMON PROLOGUE BASE
         LA    9,ADDAREA(0,13)          SET GR9 TO POINT TO ADDRESSING AREA
*                                       AT END OF DSA
         ST    9,ADVEC(0,13)            AND STORE IN ADDRESSING VECTOR.


*****
*
*              THE FOLLOWING CODE APPEARS
*        ONLY IN THE CASE OF RECURSIVE PROCEDURES
*
         L     14,PR...A(12)            LOAD GR14 WITH THE CURRENT DISPLAY
*                                       VALUE FOR A
         ST    14,92(0,13)              STORE IN DISPLAY UPDATE IN DSA
         LA    14,PR...A(12)
         SR    14,12                    GR14 = OFFSET OF DISPLAY PSEUDO-REGISTER
         ST    14,88(0,13)              STORE IN DISPLAY UPDATE IN DSA
*
*****


*              INITIALIZE ON SLOTS (IF ANY)
         MVI   0(13),X'8F'              IDENTIFY DSA.


*              COPY SKELETON DOPE VECTORS (IF ANY) FROM STATIC INTERNAL
*        CONTROL SECTION TO REAL DOPE VECTORS IN DSA. (THERE IS ALWAYS A
*        SKELETON FOR A REAL DOPE VECTOR), AND RELOCATE THE ADDRESSES WITH THE
*        ADDRESS OF THE DSA FOR THOSE DOPE VECTORS REFERRING TO VARIABLES
*        IN THE DSA.


*              FOR EACH VDA (VARIABLE DATA AREA) REQUIRED BY THE
```

```
*        PROCEDURE THE CODE BETWEEN THE LABELS VDA1 AND VDA2 IS
*        GENERATED



VDA1     SR    7,7                      CLEAR STORAGE ACCUMULATOR AC1
         SR    0,0                      CLEAR SECONDARY DOPE VECTOR STORAGE
*                                       ACCUMULATOR AC2


*              FOR EACH VARIABLE IN THE VDA, THE FOLLOWING CODE IS
*        GENERATED (BETWEEN LABELS VAR1 AND VAR2).


VAR1     EVALUATE EXTENT EXPRESSIONS (DIMENSIONS AND STRING LENGTHS) AND
*        STORE RESULTS IN DOPE VECTOR IN DSA.
*        ALIGN ACCUMULATOR AC1 ON CORRECT BOUNDARY FOR VARIABLE
*        BUMP ACCUMULATOR AC2 BY SIZE OF SECONDARY DOPE VECTOR (IF VARIABLE
*        IS DIMENSIONED AND VARYING).
*        RELOCATE ADDRESS IN VARIABLES DOPE VECTOR RELATIVE TO START OF
*        VDA.
VAR2     BUMP ACCUMULATOR AC1 BY SIZE OF STORAGE REQUIRED FOR VARIABLE
         AR    0,7                      ADD AC1 AND AC2
         L     15,36(0,11)              LOAD GR15 WITH ENTRY POINT IHESADB
         BALR  14,15                    GET VDA
         LA    1,8(0,1)                 BUMP VDA POINTER PAST FLAG AND CHAIN SLOTS
         AR    7,1                      POINT GR7 AT FIRST SECONDARY DOPE VECTOR.
         L     14,DV..VAR(0,13)         FOR EACH VARIABLE IN REGION, RELOCATE
         AR    14,1                     ADDRESS IN DOPE VECTOR.
         ST    14,DV..VAR(0,13)
*        FOR EACH DIMENSIONED VARYING ITEM IN REGION, INITIALIZE
VDA2     SECONDARY DOPE VECTORS.
         LA    10,PROCBASE              SET UP PROCEDURE BASE
                                        CODE (IF ANY) TO SET UP SOME ADDRESSING MECHANISMS
                                        IN E
                                        ADVANCE FOR USE IN PROCEDURE
         BCR   15,8                     RETURN FROM COMMON PROLOGUE.
         CNOP  0,4
AB       BC    15,8(0,15)               BRANCH ROUND BCD OF ENTRY POINT
         DC    AL1(2)
         DC    C'AB'
         STM   14,11,12(13)             SAVE REGISTERS IN CALLER'S SAVE AREA
         L     10,PROBAS(0,15)          SET UP FIRST PROLOGUE BASE
         BAL   8,GETDSA(10)             BRANCH AND LINK TO GET DSA AND TO SET
*                                       UP ADDRESSING VECTOR
         MVI   SWITCH(13),X'X3'         SET UP RETURN (EXP) SWITCH IF THERE IS
*                                       RETURN (EXP) AND DATA ATTRIBUTES OF
*                                       ENTRY LABELS DIFFER
         BC    15,COPRAM2(0,8)          BRANCH TO COPY OVER PARAMETERS
IJK      BC    15,12(0,15)              BRANCH ROUND FOLLOWING CONSTANTS
         DC    AL1(3)                   LENGTH OF BCD
         DC    C'IJK'                   BCD OF ENTRY POINT
PROBAS   DC    A(A)                     FIRST PROLOGUE BASE
         STM   14,11,12(13)             SAVE REGISTERS IN CALLER'S SAVE AREA
         L     10,PROBAS(0,15)          SET UP FIRST PROLOGUE BASE
         BAL   8,GETDSA(0,10)           BRANCH TO GET DSA AND SET UP
*                                       ADDRESSING VECTOR
         MVI   SWITCH(13),X'X4'         SET RETURN (EXP) SWITCH
COPRAM2  L     14,0(0,1)                PICK UP FIRST ARGUMENT ADDRESS
         LA    0,10(0,0)
         SR    14,0
         LH    14,0(0,14)               PICK UP PSEUDO-REGISTER OFFSET OF
         ST    14,Y(0,13)               ARGUMENT AND STORE IN DSA.
         L     14,4(0,1)                PICK UP ADDRESS OF SECOND ARGUMENT
         ST    14,Z(0,13)               AND STORE IN Z
         L     14,8(0,1)                PICK UP ADDRESS OF TARGET FIELD
         ST    14,TARGET(0,13)          AND STORE IN DSA
```

```
         L      10,A...A(0,11)          LOAD GR10 WITH ADDRESS OF FIRST BYTE
*                                       OF PROCEDURE
         BAL    8,COMMON(0,10)          BRANCH AND LINK TO COMMON PROLOGUE
         BC     15,AE...AB(0,10)        BRANCH TO APPARENT ENTRY POINT AB
*        THIS IS THE APPARENT ENTRY POINT OF A.
```

```
*        THE FOLLOWING IS AN ON BLOCK PROLOGUE WHICH IS COMPILED FOR ALL
*        ON BLOCKS EXCEPT IF BLOCK SPECIFIES SYSTEM

         STM    14,11,12(13)            SAVE REGISTERS
         LR     10,15                   SET PROLOGUE BASE

         L      11,STATIC(0,10)         SET UP STATIC INTERNAL DATA POINTER
         L      15,32(0,11)             LOAD GR15 WITH ADDRESS OF IHESADA
         L      0,SIZDSA(0,10)          LOAD GR0 WITH SIZE OF DSA
         BALR   14,15                   CALL IHESADA TO GET A DSA
         LR     14,13
         LA     0,7(0,0)
         SR     15,15
LOOP     A      14,0(0,11)              SET UP ADDRESSING VECTOR IN
         ST     14,ADVEC+4(15,13)       DSA
         LA     15,4(0,15)
         BCT    0,LOOP(0,10)
         BC     15,COMMON(0,10)         BRANCH TO INITIALIZE DSA
         DC     F'SIZE OF DSA'
         DC     A(STATIC INTERNAL CONTROL SECTION)
COMMON   BALR   10,0
*        CODE IS GENERATED HERE FOLLOWING SAME PATTERN AS FOR
*        A BEGIN PROLOGUE (SEE BELOW) COMMON SECTION.
         LA     10,ONSTART
ONSTART
```

```
*        EPILOGUE FOR AN ON BLOCK
         L      15,IHESAFA(0,11)        LOAD GR15 WITH ENTRY POINT TO EPILOGUE
         BALR   14,15                   ROUTINE AND BRANCH AND LINK TO IT
```

```
*        PROLOGUE FOR A BEGIN BLOCK
B        LA     14,BEND                 SET UP RETURN REGISTER
         BALR   15,0                    SET UP ENTRY POINT ADDRESS
         CNOP   0,4
         STM    14,11,12(13)            SAVE REGISTERS IN CONTAINING BLOCK'S DSA
         BALR   9,0                     SET UP PROLOGUE BASE
         L      15,32(0,11)             LOAD GR15 WITH ENTRY POINT TO IHESADA
         L      0,SIZDSA(0,9)
         BALR   14,15                   GET A DSA
         LR     14,13
         LA     0,7(0,0)
         SR     15,15
LOOP     A      14,0(0,11)
         ST     14,ADVEC+4(15,13)       SET UP ADDRESSING VECTOR FOR DSA
         LA     15,4(0,15)
         BCT    0,LOOP(0,9)
         BC     15,COMMON(0,9)
         DC     F'SIZE OF DSA'
         LA     9,ADDAREA(0,13)         SET GRG TO POINT TO ADDRESSING AREA
         ST     9,ADVEC(0,13)           AT END OF DSA AND STORE IN ADDRESSING
*                                       VECTOR
*        THE CODE GENERATED HERE IS THE SAME AS THAT FOR A PROCEDURE PROLOGUE
         EXCEPT THAT A DIFFERENT CODE BYTE IS MOVED TO THE FIRST BYTE OF THE
*        DSA; GR10 IS NOT RESET; AND THE BCR 15,8 IS NOT GENERATED.
```

```
*        EPILOGUE OF A BEGIN BLOCK
         L     15,IHESAFA              LOAD GR15 WITH ENTRY POINT OF
         BALR  14,15                   EPILOGUING ROUTINE AND CALL IT
BEND
```

```
*        RETURN (EXP) STATEMENT EXAMINES THE LOCATION 'SWITCH' IN THE DSA
*        SET BY THE PROLOGUE TO DETERMINE THE CONVERSION REQUIRED ON
*        THE EXPRESSION. IT THEN ASSIGNS THE CONVERTED EXPRESSION TO
*        THE TARGET FIELD FOR WHICH THE LOCATION 'TARGET', IN THE DSA,
*        POINTS TO EITHER ITS DOPE VECTOR (IN THE CASE OF A STRING)
*        OR THE STORAGE. ROUTINE IHESAFA IS THEN INVOKED.
*        END STATEMENT (WHICH IS THE SAME AS A RETURN STATEMENT)
         L     15,IHESAFA
         BALR  14,15
```

DSA OPTIMIZATION

In compilations specifying OPT=1, if a PROCEDURE or BEGIN block has a DSA which requires less than 512 bytes of storage, such storage may, under certain conditions, be obtained from STATIC storage or from library workspace. To obtain a STATIC DSA, the block must satisfy these conditions:

1. The block must not be re-entrant or recursive

2. The block must not be nested (at any depth) within an ON block

3. The block must not have the MAIN or TASK options

A block which is ineligible for a STATIC DSA, and whose DSA will never be active when any new DSA is required, is allocated its DSA from library workspace.

Each block requiring a DSA either in STATIC or in library workspace calls one of two compiled subroutines, instead of IHESAD, to allocate the storage. Either or both subroutines, if required, are compiled onto the end of the program, and are prefixed by the comments 'STATIC PROLOGUE SUBROUTINE' and 'DYNAMIC PROLOGUE subroutine' respectively. Entry may be made to the STATIC prologue subroutine at one of several points.

Any block using one of these prologue subroutines will also use a compiled Epilogue subroutine, which will be called for the END statement of the block, or for a RETURN statement without an expression. (The same Epilogue subroutine serves both STATIC and library workspace DSAs.) If there is any core to be freed, the Epilogue subroutine will call IHESAFA. The Epilogue subroutine is also compiled onto the end of the program, and always immediately precedes the STATIC Prologue subroutine if this is present.

The address of the Dynamic Prologue subroutine and the Epilogue subroutine are placed in the STATIC internal control section, at offsets 40 and 48 from the start respectively. Since the STATIC Prologue subroutine always follows the Epilogue subroutine, which is of fixed length, a third address slot is not required for it.

Listings of the Dynamic and Static Prologue and the Epilogue subroutines

```
*        DYNAMIC PROLOGUE SUBROUTINE

         L     5,PR..IHEQLWF(12)
         LTR   5,5
         BC    8,90(15)
         L     6,PR..IHEQINV(12)
         LTR   6,6
         BC    4,90(15)
         LR    13,5
         SR    2,2
         L     3,PR..IHEQSLA(12)
         ST    13,PR..IHEQSLA(12)
         ST    3,4(13)
         TM    0(3),X'80'
         BC    1,52(15)
         L     3,4(3)
         B     36(15)
         ST    13,8(3)
         L     4,PR..IHEQINV(12)
         LA    4,1(4)
         ST    4,PR..IHEQINV(12)
         ST    4,84(13)
         ST    2,80(13)
         ST    2,8(13)
         MVI   76(13),X'00'
         ST    2,96(13)
         BR    14

         L     15,32(11)
         BR    15

*        END SUBROUTINE
```

```
*       EPILOGUE SUBROUTINE

        TM      1(13),X'80'
        BC      8,60(15)
        L       2,80(13)
        LTR     2,2
        BC      7,60(15)
        C       13,PR..IHEQSLA(12)
        BC      7,60(15)
        L       13,4(13)
        ST      13,PR..IHEQSLA(12)
        TM      0(13),X'80'
        BC      1,50(15)
        L       13,4(13)
        B       34(15)
        ST      2,8(13)
        LM      14,11,12(13)
        BR      14
        L       15,A..IHESAFA
        BR      15

*       END SUBROUTINE
```

```
*       STATIC PROLOGUE SUBROUTINE

        L       4,PR..IHEQINV(12)
        LTR     4,4
        BC      11,86(15)
        L       7,PR..IHEQLWO(12)
        MVC     80(4,3),80(7)
        LA      4,1(4)
        ST      4,PR..IHEQINV(12)
        ST      4,84(3)
        MVI     76(3),X'00'
        ST      3,8(13)
        LR      13,3
        L       3,PR..IHEQSLA(12)
        ST      3,4(13)
        ST      13,PR..IHEQSLA(12)
        SR      2,2
        ST      2,80(13)
        ST      2,8(13)
        ST      2,96(13)
        BR      14

*       END SUBROUTINE
```

Messages produced by the PL/I compiler are explained in the PL/I Programmer's Guide. In the table below, each compiler message number is associated with the phase and module in which the corresponding message is generated.

Message numbers are not listed for PLC and ODC messages. All messages numbered CFBAAxxx (where xxx is a three-digit number) are generated in PLC. All messages numbered CFBABxxx are generated in ODC. These messages are explained in IBM System/360 Time Sharing System, System Messages.

Message Number	Logical Phase	Module
IEM0001I	Read In	CA
IEM0002I	Read In	CA
IEM0003I	Read In	CA,CP
IEM0004I	Read In	CA
IEM0005I	Read In	CA,CL
IEM0006I	Read In	CA
IEM0007I	Read In	CA
IEM0008I	Read In	CA
IEM0009I	Read In	CA
IEM0010I	Read In	CA
IEM0011I	Read In	CA
IEM0012I	Read In	CA
IEM0013I	Read In	CA
IEM0014I	Read In	CA
IEM0015I	Read In	CA
IEM0016I	Read In	CA
IEM0017I	Read In	CA
IEM0018I	Read In	CA
IEM0019I	Read In	CA
IEM0020I	Read In	CA
IEM0021I	Read In	CA
IEM0022I	Read In	CA
IEM0023I	Read In	CA
IEM0024I	Read In	CA
IEM0025I	Read In	CA
IEM0026I	Read In	CA
IEM0027I	Read In	CA
IEM0028I	Read In	CG
IEM0029I	Read In	CA
IEM0031I	Read In	CA,CL,CT
IEM0032I	Read In	CC
IEM0033I	Read In	CC
IEM0035I	Read In	CC
IEM0037I	Read In	CC
IEM0038I	Read In	CC
IEM0039I	Read In	CC
IEM0040I	Read In	CC
IEM0044I	Read In	CC
IEM0045I	Read In	CC
IEM0046I	Read In	CC
IEM0048I	Read In	CG
IEM0050I	Read In	CL,CP
IEM0051I	Read In	CL,CP
IEM0052I	Read In	CO
IEM0053I	Read In	CO
IEM0054I	Read In	CO
IEM0055I	Read In	CP

Message Number	Logical Phase	Module
IEM0057I	Read In	CC
IEM0058I	Read In	CC
IEM0059I	Read In	CP
IEM0060I	Read In	CP
IEM0061I	Read In	CP
IEM0063I	Read In	CO
IEM0064I	Read In	CC
IEM0066I	Read In	CG
IEM0067I	Read In	CL
IEM0069I	Read In	CG
IEM0070I	Read In	CG
IEM0071I	Read In	CG
IEM0072I	Read In	CG
IEM0074I	Read In	CG
IEM0075I	Read In	CG
IEM0076I	Read In	CG
IEM0077I	Read In	CG
IEM0078I	Read In	CG
IEM0080I	Read In	CG
IEM0081I	Read In	CG
IEM0082I	Read In	CG
IEM0083I	Read In	CG
IEM0084I	Read In	CG
IEM0085I	Read In	CI
IEM0086I	Read In	CI
IEM0089I	Read In	CI
IEM0090I	Read In	CI
IEM0094I	Read In	CI
IEM0095I	Read In	CI
IEM0096I	Read In	CG,CI
IEM0097I	Read In	CI
IEM0099I	Read In	CI
IEM0100I	Read In	CI
IEM0101I	Read In	CM
IEM0102I	Read In	CI
IEM0103I	Read In	CI
IEM0104I	Read In	CC
IEM0105I	Read In	CC,CG
IEM0106I	Read In	CI,CV
IEM0107I	Read In	CI
IEM0108I	Read In	CI
IEM0109I	Read In	CG,CI
IEM0110I	Read In	CI
IEM0111I	Read In	CI
IEM0112I	Read In	CI
IEM0113I	Read In	CG,CM
IEM0114I	Read In	CI

Message Number	Logical Phase	Module
IEM0115I	Read In	CL
IEM0116I	Read In	CI
IEM0117I	Read In	CM
IEM0118I	Read In	CL
IEM0119I	Read In	CM
IEM0120I	Read In	CM
IEM0121I	Read In	CO
IEM0122I	Read In	CO
IEM0123I	Read In	CM
IEM0124I	Read In	CO
IEM0125I	Read In	CO
IEM0126I	Read In	CO
IEM0127I	Read In	CO
IEM0128I	Read In	CO
IEM0129I	Read In	CL
IEM0130I	Read In	CL
IEM0131I	Read In	CO
IEM0132I	Read In	CO
IEM0134I	Read In	CP
IEM0135I	Read In	CP
IEM0136I	Read In	CO
IEM0138I	Read In	CP
IEM0139I	Read In	CP
IEM0140I	Read In	CO
IEM0141I	Read In	CP
IEM0144I	Read In	CO
IEM0145I	Read In	CO
IEM0147I	Read In	CO
IEM0149I	Read In	CL,CM
IEM0150I	Read In	CL
IEM0151I	Read In	CO
IEM0152I	Read In	CO
IEM0153I	Read In	CO
IEM0154I	Read In	CA
IEM0158I	Read In	CO
IEM0159I	Read In	CO
IEM0163I	Read In	CT
IEM0166I	Read In	CL
IEM0172I	Read In	CL
IEM0180I	Read In	CT
IEM0181I	Read In	CL
IEM0182I	Read In	CL,CS, CT,CV
IEM0185I	Read In	CT
IEM0187I	Read In	CT
IEM0191I	Read In	CT
IEM0193I	Read In	CT
IEM0194I	Read In	CT
IEM0195I	Read In	CT
IEM0198I	Read In	CT
IEM0202I	Read In	CL
IEM0207I	Read In	CG
IEM0208I	Read In	CG
IEM0209I	Read In	CC
IEM0211I	Read In	CL
IEM0212I	Read In	CP
IEM0213I	Read In	CP
IEM0214I	Read In	CP
IEM0216I	Read In	CP
IEM0217I	Read In	CP
IEM0220I	Read In	CT

Message Number	Logical Phase	Module
IEM0221I	Read In	CT
IEM0222I	Read In	CT
IEM0223I	Read In	CT
IEM0224I	Read In	CT
IEM0225I	Read In	CT
IEM0226I	Read In	CT
IEM0227I	Read In	CT
IEM0228I	Read In	CT
IEM0229I	Read In	CT
IEM0230I	Read In	CS,CT
IEM0231I	Read In	CT
IEM0232I	Read In	CT
IEM0233I	Read In	CV
IEM0235I	Read In	CS
IEM0236I	Read In	CS
IEM0237I	Read In	CS
IEM0238I	Read In	CV
IEM0239I	Read In	CS
IEM0240I	Read In	CV
IEM0241I	Read In	CV
IEM0242I	Read In	CV
IEM0243I	Read In	CV
IEM0244I	Read In	CV
IEM0245I	Read In	CV
IEM0247I	Read In	CW
IEM0254I	Read In	CC
IEM0255I	Read In	CG
IEM0510I	Dictionary	EH
IEM0511I	Dictionary	EH
IEM0512I	Dictionary	EH
IEM0513I	Dictionary	EG
IEM0514I	Dictionary	EG
IEM0515I	Dictionary	EG
IEM0516I	Dictionary	EG
IEM0517I	Dictionary	EG
IEM0518I	Dictionary	EG
IEM0519I	Dictionary	EG
IEM0520I	Dictionary	EG
IEM0521I	Dictionary	EG
IEM0522I	Dictionary	EG
IEM0523I	Dictionary	EG
IEM0524I	Dictionary	EH
IEM0525I	Dictionary	EI
IEM0527I	Dictionary	EJ
IEM0528I	Dictionary	EH
IEM0529I	Dictionary	EI
IEM0530I	Dictionary	EI
IEM0531I	Dictionary	EI
IEM0532I	Dictionary	EI
IEM0533I	Dictionary	EI
IEM0534I	Dictionary	EI
IEM0536I	Dictionary	EI
IEM0537I	Dictionary	EI
IEM0538I	Dictionary	EJ
IEM0539I	Dictionary	EJ
IEM0540I	Dictionary	EJ
IEM0541I	Dictionary	EJ
IEM0542I	Dictionary	EJ
IEM0543I	Dictionary	EL,EK,EM
IEM0544I	Dictionary	EL,EK,EM
IEM0545I	Dictionary	EL,EK,EM

Message Number	Logical Phase	Module
IEM0546I	Dictionary	EL,EK,EM
IEM0547I	Dictionary	EL,EK,EM
IEM0548I	Dictionary	EL,EK,EM
IEM0549I	Dictionary	EL,EK,EM
IEM0550I	Dictionary	EL,EK,EM
IEM0551I	Dictionary	EK,EL,EM
IEM0552I	Dictionary	EL,EK,EM
IEM0553I	Dictionary	EL,EK,EM
IEM0554I	Dictionary	EL,EK,EM
IEM0555I	Dictionary	EL,EK,EM
IEM0556I	Dictionary	EL,EK,EM
IEM0557I	Dictionary	EL,EK,EM
IEM0558I	Dictionary	EL,EK,EM
IEM0559I	Dictionary	EL,EK,EM
IEM0560I	Dictionary	EL,EK,EM
IEM0561I	Dictionary	EL,EK,EM
IEM0562I	Dictionary	EK,EL,EM
IEM0563I	Dictionary	EK,EL,EM
IEM0564I	Dictionary	EK,EL,EM
IEM0565I	Dictionary	EK,EL,EM
IEM0566I	Dictionary	EK,EL,EM
IEM0567I	Dictionary	EP
IEM0568I	Dictionary	EP
IEM0569I	Dictionary	EP
IEM0570I	Dictionary	EP
IEM0571I	Dictionary	EK
IEM0572I	Dictionary	EL
IEM0573I	Dictionary	EL
IEM0576I	Dictionary	EL
IEM0577I	Dictionary	EL
IEM0578I	Dictionary	EL
IEM0579I	Dictionary	EL
IEM0580I	Dictionary	EL
IEM0589I	Dictionary	EW
IEM0590I	Dictionary	EW
IEM0591I	Dictionary	EW
IEM0592I	Dictionary	EW
IEM0593I	Dictionary	EW
IEM0594I	Dictionary	EW
IEM0595I	Dictionary	EW
IEM0596I	Dictionary	EW
IEM0597I	Dictionary	EW
IEM0598I	Dictionary	EW
IEM0599I	Dictionary	EW
IEM0602I	Dictionary	FV,FW
IEM0603I	Dictionary	FV,FW
IEM0604I	Dictionary	FV,FW
IEM06052	Dictionary	FV,FW
IEM0606I	Dictionary	FV,FW
IEM0607I	Dictionary	FV,FW
IEM0608I	Dictionary	FV,FW
IEM0609I	Dictionary	FV,FW
IEM0610I	Dictionary	FV,FW
IEM0611I	Dictionary	FV,FW
IEM0612I	Dictionary	FV
IEM0613I	Dictionary	FW
IEM0614I	Dictionary	FW
IEM0623I	Dictionary	FV,FW
IEM0624I	Dictionary	FV,FW
IEM0625I	Dictionary	FV,FW
IEM0626I	Dictionary	FV,FW
IEM0628I	Dictionary	FV,FW

Message Number	Logical Phase	Module
IEM0629I	Dictionary	FV,FW
IEM0630I	Dictionary	FV,FW
IEM0631I	Dictionary	FV,FW
IEM0632I	Dictionary	FV,FW
IEM0633I	Dictionary	EY
IEM0634I	Dictionary	EY
IEM0636I	Dictionary	EY
IEM0637I	Dictionary	EY
IEM0638I	Dictionary	EY
IEM0640I	Dictionary	EY
IEM0641I	Dictionary	EY
IEM0642I	Dictionary	EY
IEM0643I	Dictionary	EY
IEM0644I	Dictionary	EY
IEM0645I	Dictionary	EY
IEM0646I	Dictionary	EY
IEM0647I	Dictionary	EY
IEM0653I	Dictionary	FE
IEM0655I	Dictionary	FE
IEM0656I	Dictionary	FE
IEM0657I	Dictionary	FE
IEM0658I	Dictionary	FE
IEM0660I	Dictionary	FE
IEM0661I	Dictionary	FE
IEM0662I	Dictionary	FE
IEM0673I	Dictionary	FE
IEM0674I	Dictionary	FF
IEM0675I	Dictionary	FF
IEM0676I	Dictionary	FF
IEM0677I	Dictionary	FE
IEM0682I	Dictionary	FI
IEM0683I	Dictionary	FI
IEM0684I	Dictionary	FI
IEM0685I	Dictionary	FI
IEM0686I	Dictionary	FI
IEM0687I	Dictionary	FI
IEM0688I	Dictionary	FI
IEM0689I	Dictionary	FI
IEM0690I	Dictionary	FI
IEM0691I	Dictionary	FI
IEM0692I	Dictionary	FI
IEM0693I	Dictionary	FI
IEM0694I	Dictionary	FI
IEM0695I	Dictionary	FI
IEM0696I	Dictionary	FI
IEM0697I	Dictionary	FI
IEM0698I	Dictionary	FI
IEM0699I	Dictionary	FI
IEM0700I	Dictionary	FI
IEM0701I	Dictionary	FI
IEM0702I	Dictionary	FI
IEM0703I	Dictionary	FI
IEM0704I	Dictionary	FI
IEM07051	Dictionary	FI
IEM0706I	Dictionary	FI
UEM0707I	Dictionary	FI
IEM0715I	Dictionary	EJ
IEM0718I	Dictionary	FO
IEM0719I	Dictionary	FO
IEM0720I	Dictionary	FO
IEM0721I	Dictionary	FO
IEM0722I	Dictionary	FO

Message Number	Logical Phase	Module
IEM0723I	Dictionary	FO
IEM0724I	Dictionary	FO
IEM0725I	Dictionary	FO
IEM0726I	Dictionary	FO
IEM0727I	Dictionary	FO
IEM0728I	Dictionary	FO
IEM0729I	Dictionary	FO
IEM0730I	Dictionary	FQ
IEM0731I	Dictionary	FQ
IEM0732I	Dictionary	FQ
IEM0733I	Dictionary	FQ
IEM0734I	Dictionary	FQ
IEM0735I	Dictionary	FQ
IEM0736I	Dictionary	FQ
IEM0737I	Dictionary	FQ
IEM0739I	Dictionary	FQ
IEM0740I	Dictionary	FQ
IEM0741I	Dictionary	FQ
IEM0742I	Dictionary	FQ
IEM0745I	Dictionary	FQ
IEM0746I	Dictionary	FQ
IEM0747I	Dictionary	FQ
IEM0748I	Dictionary	FQ
IEM0749I	Dictionary	FQ
IEM0750I	Dictionary	FQ
IEM0751I	Dictionary	FQ
IEM0752I	Dictionary	FQ
IEM0754I	Dictionary	FQ
IEM0755I	Dictionary	FQ
IEM0756I	Dictionary	FQ
IEM0758I	Dictionary	FQ
IEM0759I	Dictionary	FQ
IEM0760I	Dictionary	FQ
IEM0761I	Dictionary	FQ
IEM0762I	Dictionary	FQ
IEM0769I	Pretranslator	GB
IEM0770I	Pretranslator	GB
IEM0771I	Pretranslator	GB
IEM0778I	Pretranslator	GB
IEM0779I	Pretranslator	GB
IEM0780I	Pretranslator	GB
IEM0781I	Pretranslator	GB
IEM0782I	Pretranslator	GB
IEM0786I	Pretranslator	GK
IEM0787I	Pretranslator	GK
IEM0791I	Pretranslator	GK
IEM0792I	Pretranslator	GP,GQ,GR
IEM0793I	Pretranslator	GP,GQ,GR
IEM0794I	Pretranslator	GP,GQ,GR
IEM0795I	Pretranslator	GP,GQ,GR
IEM0796I	Pretranslator	GP,GQ,GR
IEM0797I	Pretranslator	GP,GQ,GR
IEM0798I	Pretranslator	GP,GQ,GR
IEM0799I	Pretranslator	GP,GQ,GR
IEM0800I	Pretranslator	GP,GQ,GR
IEM0801I	Pretranslator	GP,GQ,GR
IEM0802I	Pretranslator	GP,GQ,GR
IEM0803I	Pretranslator	GP,GQ,GR
IEM0804I	Pretranslator	GP,GQ,GR
IEM0805I	Pretranslator	GP,GQ,GR
IEM0806I	Pretranslator	GP,GQ,GR
IEM0807I	Pretranslator	GP,GQ,GR
IEM0816I	Pretranslator	GU,GV

Message Number	Logical Phase	Module
IEM0817I	Pretranslator	GU,GV
IEM0818I	Pretranslator	GU,GV
IEM0819I	Pretranslator	GU,GV
IEM0820I	Pretranslator	GU,GV
IEM0821I	Pretranslator	GU,GV
IEM0823I	Pretranslator	GU,GV
IEM0824I	Pretranslator	GU
IEM0825I	Pretranslator	GU,GV
IEM0826I	Pretranslator	GU,GV
IEM0832I	Pretranslator	HF,HG
IEM0833I	Pretranslator	HF,HG
IEM0834I	Pretranslator	HF,HG
IEM0835I	Pretranslator	HF,HG
IEM0836I	Pretranslator	HF,HG
IEM0837I	Pretranslator	HF,HG
IEM0838I	Pretranslator	HF
IEM0848I	Pretranslator	HF,HG
IEM0849I	Pretranslator	HF,HG
IEM0850I	Pretranslator	HF,HG
IEM0851I	Pretranslator	HF,HG
IEM0852I	Pretranslator	HF,HG
IEM0853I	Pretranslator	HF,HG
IEM0864I	Pretranslator	HK,HL
IEM0865I	Pretranslator	HK,HL
IEM0866I	Pretranslator	HK,HL
IEM0867I	Pretranslator	HK,HL
IEM0868I	Pretranslator	HK,HL
IEM0869I	Pretranslator	HK,HL
IEM0870I	Pretranslator	HK,HL
IEM0871I	Pretranslator	HK,HL
IEM0872I	Pretranslator	HK,HL
IEM0873I	Pretranslator	HK,HL
IEM0874I	Pretranslator	HK,HL
IEM0875I	Pretranslator	HK,HL
IEM0876I	Pretranslator	HK,HL
IEM0877I	Pretranslator	HK,HL
IEM0878I	Pretranslator	HK,HL
IEM0879I	Pretranslator	HK,HL
IEM0880I	Pretranslator	HK,HL
IEM0881I	Pretranslator	HK,HL
IEM0882I	Pretranslator	HK
IEM0896I	Pretranslator	HP
IEM0897I	Pretranslator	HP
IEM0898I	Pretranslator	HP
IEM0899I	Pretranslator	HP
IEM0900I	Pretranslator	HP
IEM0901I	Pretranslator	HP
IEM0902I	Pretranslator	HP
IEM0903I	Pretranslator	HP
IEM0906I	Pretranslator	HP
IEM0907I	Pretranslator	HP
IEM1024I	Translator	IA
IEM1025I	Translator	IA
IEM1026I	Translator	IA
IEM1027I	Translator	IA
IEM1028I	Translator	IA
IEM1029I	Translator	IA
IEM1030I	Translator	IA
IEM1040I	Translator	IM
IEM1051I	Translator	IM
IEM1056I	Translator	IM
IEM1057I	Translator	IM
IEM1058I	Translator	IM

Message Number	Logical Phase	Module
IEM1059I	Translator	IM
IEM1060I	Translator	IM
IEM1061I	Translator	IM
IEM1062I	Translator	IM
IEM1063I	Translator	IM
IEM1064I	Translator	IM
IEM1065I	Translator	IM
IEM1066I	Translator	IM
IEM1067I	Translator	IM
IEM1068I	Translator	IM
IEM1070I	Translator	IM
IEM1071I	Translator	IM
IEM1072I	Translator	IM
IEM1073I	Translator	IM
IEM1074I	Translator	IM
IEM1076I	Translator	JD
IEM1082I	Translator	IX
IEM1088I	Aggregates	JK
IEM1089I	Aggregates	JK
IEM1090I	Aggregates	JK
IEM1091I	Aggregate Preprocessor	JI
IEM1092I	Aggregates	JK
IEM1104I	Aggregates	JP
IEM1105I	Aggregates	JP
IEM1106I	Aggregates	JP
IEM1107I	Aggregates	JP
IEM1108I	Aggregates	JP
IEM1110I	Aggregates	JP
IEM1111I	Aggregates	JP
IEM1112I	Aggregates	JP
IEM1113I	Aggregates	JP
IEM1114I	Aggregates	JP
IEM1115I	Aggregates	JP
IEM1120I	Aggregates	JP
IEM1121I	Aggregates	JP
IEM1122I	Aggregates	JP
IEM1123I	Pseudo-code	LD
IEM1125I	Pseudo-code	LD
IEM1200I	Pseudo-code	KT
IEM1210I	Do loop optimization	KC
IEM1211I	Pseudo-code	KE
IEM1220I	Do loop optimization	KU
IEM1223I	Do loop optimization	KO
IEM1224I	Do loop optimization	KA
IEM1569I	Pseudo-code	LG-ON
IEM1570I	Pseudo-code	LG
IEM1571I	Pseudo-code	LG
IEM1572I	Pseudo-code	LG
IEM1574I	Pseudo-code	LG
IEM1575I	Pseudo-code	LG
IEM1588I	Pseudo-code	MD
IEM1600I	Pseudo-code	LS,LT,LU
IEM1601I	Pseudo-code	LS
IEM1602I	Pseudo-code	LS,LT,LU
IEM1603I	Pseudo-code	LS,LT,LU
IEM1604I	Pseudo-code	LS,LT,LU
IEM1605I	Pseudo-code	LS,LT,LU
IEM1606I	Pseudo-code	LS,LT,LU
IEM1607I	Pseudo-code	LS,LT,LU
IEM1608I	Pseudo-code	LS,LT,LU
IEM1609I	Pseudo-code	LS,LT,LU
IEM1610I	Pseudo-code	LW
IEM1611I	Pseudo-code	LW

Message Number	Logical Phase	Module
IEM1612I	Pseudo-code	LW
IEM1613I	Pseudo-code	LS,LT,LU
IEM1614I	Pseudo-code	LW
IEM1615I	Pseudo-code	ME
IEM1616I	Pseudo-code	ME
IEM1617I	Pseudo-code	MB
IEM1618I	Pseudo-code	MB
IEM1619I	Pseudo-code	MB
IEM1620I	Pseudo-code	MB
IEM1622I	Pseudo-code	MB,ME
IEM1623I	Pseudo-code	MB
IEM1624I	Pseudo-code	MB
IEM1625I	Pseudo-code	MB
IEM1626I	Pseudo-code	ME
IEM1627I	Pseudo-code	ME
IEM1628I	Pseudo-code	ME
IEM1629I	Pseudo-code	ME
IEM1630I	Pseudo-code	MG,MH
IEM1631I	Pseudo-code	MI,MJ
IEM1632I	Pseudo-code	MI,MJ
IEM1633I	Pseudo-code	ME
IEM1634I	Pseudo-code	ME
IEM1635I	Pseudo-code	ME
IEM1636I	Pseudo-code	ME
IEM1637I	Pseudo-code	ME
IEM1638I	Pseudo-code	ME
IEM1639I	Pseudo-code	MF
IEM1640I	Pseudo-code	MM,MN
IEM1641I	Pseudo-code	MM,MN
IEM1642I	Pseudo-code	MM,MN
IEM1643I	Pseudo-code	MM,MN
IEM1644I	Pseudo-code	MM,MN
IEM1645I	Pseudo-code	MM,MN
IEM1648I	Pseudo-code	MM,MN
IEM1649I	Pseudo-code	MM,MN
IEM1650I	Pseudo-code	MM,MN
IEM1651I	Pseudo-code	MM,MN
IEM1652I	Pseudo-code	MM,MN
IEM1653I	Pseudo-code	MM,MN
IEM1654I	Pseudo-code	MM,MN
IEM1655I	Pseudo-code	MN
IEM1656I	Pseudo-code	ME
IEM1657I	Pseudo-code	MM
IEM1658I	Pseudo-code	MN
IEM1670I	Pseudo-code	MP
IEM1671I	Pseudo-code	MP
IEM1680I	Pseudo-code	MS
IEM1687I	Pseudo-code	MS
IEM1688I	Pseudo-code	MS
IEM1689I	Pseudo-code	MS
IEM1692I	Pseudo-code	MS
IEM1693I	Pseudo-code	MS
IEM1695I	Pseudo-code	MA
IEM1696I	Pseudo-code	MA
IEM1750I	Pseudo-code	MS
IEM1751I	Pseudo-code	MS
IEM1752I	Pseudo-code	NA
IEM1753I	Pseudo-code	NA
IEM1754I	Pseudo-code	NA
IEM1790I	Pseudo-code	OG,OM
IEM1793I	Pseudo-code	OE
IEM1794I	Pseudo-code	OE
IEM1795I	Pseudo-code	OE

Message Number	Logical Phase	Module
IEM1796I	Pseudo-code	OE
IEM1797I	Pseudo-code	OE
IEM1800I	Pseudo-code	OS
IEM1801I	Pseudo-code	OS
IEM1802I	Pseudo-code	OS
IEM1803I	Pseudo-code	OS
IEM1804I	Pseudo-code	OS
IEM1805I	Pseudo-code	OS
IEM1806I	Pseudo-code	OS
IEM1807I	Pseudo-code	OS
IEM1808I	Pseudo-code	OS
IEM1809I	Pseudo-code	OS
IEM1810I	Pseudo-code	OS
IEM1811I	Pseudo-code	OS
IEM1812I	Pseudo-code	OS
IEM1813I	Pseudo-code	OS
IEM1814I	Pseudo-code	OS
IEM1815I	Pseudo-code	OS
IEM1816I	Pseudo-code	NJ
IEM1817I	Pseudo-code	NJ
IEM1818I	Pseudo-code	NJ
IEM1819I	Pseudo-code	NJ
IEM1820I	Pseudo-code	NJ
IEM1821I	Pseudo-code	NJ
IEM1822I	Pseudo-code	NJ
IEM1823I	Pseudo-code	NJ
IEM1824I	Pseudo-code	NM
IEM1825I	Pseudo-code	NG
IEM1826I	Pseudo-code	NG
IEM1827I	Pseudo-code	NG
IEM1828I	Pseudo-code	NG
IEM1829I	Pseudo-code	NG
IEM1830I	Pseudo-code	NG
IEM1831I	Pseudo-code	NJ
IEM1832I	Pseudo-code	NM
IEM1833I	Pseudo-code	NM
IEM1834I	Pseudo-code	NM
IEM1835I	Pseudo-code	NM
IEM1836I	Pseudo-code	NM
IEM1837I	Pseudo-code	NM
IEM1838I	Pseudo-code	NM
IEM1839I	Pseudo-code	NM
IEM1840I	Pseudo-code	NM
IEM1841I	Pseudo-code	NM
IEM1843I	Pseudo-code	NM
IEM1844I	Pseudo-code	NM
IEM1845I	Pseudo-code	NM
IEM1846I	Pseudo-code	NM
IEM1847I	Pseudo-code	NM
IEM1848I	Pseudo-code	NM
IEM1849I	Constant Conversions	OS
IEM1850I	Constant Conversions	OS
IEM1860I	Pseudo-code	NU
IEM1861I	Pseudo-code	NU
IEM1862I	Pseudo-code	NU
IEM1870I	Pseudo-code	NU
IEM1871I	Pseudo-code	NU
IEM1872I	Pseudo-code	NU
IEM1873I	Pseudo-code	NU
IEM1874I	Pseudo-code	NU
IEM1875I	Pseudo-code	NV
IEM2304I	Storage Allocation	PD
IEM2305I	Storage Allocation	PD

Message Number	Logical Phase	Module
IEM2352I	Storage Allocation	PD
IEM2650I	Register Allocation	RA
IEM2660I	Register Allocation	RD
IEM2661I	Register Allocation	RD
IEM2700I	Register Allocation	RF,RG,RH
IEM2701I	Register Allocation	RF,RG,RH
IEM2702I	Register Allocation	RF,RG,RH
IEM2703I	Register Allocation	RF,RG,RH
IEM2704I	Register Allocation	RF,RG,RH
IEM2705I	Register Allocation	RF,RG,RH
IEM2706I	Register Allocation	RF,RG,RH
IEM2707I	Register Allocation	RF,RG,RH
IEM2708I	Register Allocation	RF,RG,RH
IEM2709I	Register Allocation	RF,RG,RH
IEM2710I	Register Allocation	RF,RG,RH
IEM2711I	Register Allocation	RF,RG,RH
IEM2712I	Register Allocation	RF,RG,RH
IEM2817I	DCB Generation	GA
IEM2818I	DCB Generation	GA
IEM2819I	DCB Generation	GA
IEM2820I	DCB Generation	GA
IEM2821I	DCB Generation	GA
IEM2822I	DCB Generation	GA
IEM2823I	DCB Generation	GA
IEM2824I	DCB Generation	GA
IEM2825I	DCB Generation	GA
IEM2826I	DCB Generation	GA
IEM2827I	DCB Generation	GA
IEM2828I	DCB Generation	GA
IEM2829I	DCB Generation	GA
IEM2830I	DCB Generation	GA
IEM2831I	DCB Generation	GA
IEM2832I	DCB Generation	GA
IEM2833I	Final Assembly	TF
IEM2834I	Final Assembly	TF
IEM2835I	Final Assembly	TF
IEM2836I	Final Assembly	TF
IEM2837I	Final Assembly	TF
IEM2852I	Final Assembly	TJ
IEM2853I	Final Assembly	TJ
IEM2854I	Final Assembly	TJ
IEM2855I	Final Assembly	TJ
IEM2865I	Final Assembly	TO
IEM2866I	Final Assembly	TO
IEM2867I	Final Assembly	TO
IEM2868I	Final Assembly	TO
IEM2881I	Final Assembly	TT
IEM2882I	Final Assembly	TT
IEM2883I	Final Assembly	TT
IEM2884I	Final Assembly	TT
IEM2885I	Final Assembly	TT
IEM2886I	Final Assembly	TT
IEM2887I	Final Assembly	TT
IEM2888I	Final Assembly	TT
IEM2897I	Final Assembly	UA
IEM2898I	Final Assembly	UA
IEM2899I	Final Assembly	UC
IEM2900I	Final Assembly	UC
IEM2913I	Final Assembly	UF
IEM3088I	Dictionary, Declare Pass 2	EL
IEM3136I-3149I	Dictionary, Declare Pass 2	EL

Message Number	Logical Phase	Module
IEM3151I-	Dictionary, Declare Pass 2	EL
IEM3153I-	Dictionary, Declare Pass 2	EL
IEM3154I	Dictionary, Declare Pass 2	EL
IEM3156I	Dictionary, Declare Pass 2	EL
IEM3162I	Dictionary, Declare Pass 2	EL
IEM3167I- 3173I	Dictionary, Declare Pass 2	EL
IEM3176I- 3190I	Dictionary, Declare Pass 2	EL
IEM3199I- 3213I	Dictionary, Declare Pass 2	EL
IEM3216I	-	AB,AM
IEM3217I	Dictionary	F1
IEM3218I	Dictionary	F1
IEM3219I	Dictionary	F1
IEM3220I	Dictionary	F1
IEM3221I	Dictionary	F1
IEM3222I	Compiler Control	AB
IEM3584I	48 Character Preprocessor	BX
IEM3840I	Compiler Control	AA
IEM3841I	Compiler Control	AA
IEM3842I	Compiler Control	AA
IEM3843I	Compiler Control	AA
IEM3844I	Compiler Control	AA
IEM3845I	Compiler Control	AA
IEM3846I	Compiler Control	AA
	Optimization	KA
IEM3847I	Compiler Control	AA
IEM3848I	Compiler Control	AA
IEM3849I	Compiler Control	AA
IEM3850I	Compiler Control	AA
IEM3851I	Compiler Control	AA
IEM3852I	Compiler Control	AA
IEM3853I	Compiler Control	AA
IEM3855I	Compiler Control	AA
IEM3856I	Compiler Control	AA
IEM3857I	Compiler Control	AA
IEM3858I	Compiler Control	AA
IEM3859I	Compiler Control	AA
IEM3860I	Compiler Control	AA
IEM3861I	Compiler Control	AA
IEM3862I	Compiler Control	AA
IEM3864I	Compiler Control	AA
IEM3865I	Compiler Control	AA
IEM3866I	Compiler Control	AA
IEM3872I	Compiler Control	AA
IEM3873I	Compiler Control	AA
IEM3874I	Compiler Control	AA
IEM3876I	Compiler Control	AA
IEM3878I	Compiler Control	AA
IEM3887I	Compiler Control	AA
IEM3888I	Compiler Control	AB
IEM3889I	Compiler Control	AB
IEM3890I	Compiler Control	AA
IEM3891I	Compiler Control	AA
IEM3892I	Compiler Control	AA
IEM3893I	Compiler Control	AA

Message Number	Logical Phase	Module
IEM3894I	Compiler Control	AA
IEM3895I	Compiler Control	AA
IEM3896I	Compiler Control	AA
IEM3897I	Compiler Control	AA
IEM3898I	Compiler Control	AA
IEM3899I	Compiler Control	AL
IEM3900I	Compiler Control	AB
IEM3901I	Compiler Control	AB
IEM3902I	Compiler Control	AB
IEM3902I	Compiler Control	AB
IEM3903I	Compiler Control	AB
IEM3904I	Compiler Control	AA
IEM3905I	Compiler Control	AA
IEM3906I	Compiler Control	AA
IEM3907I	Compiler Control	AA
IEM3908I	Compiler Control	AA
IEM3909I	Compiler Control	AL
IEM3910I	Compiler Control	AB
IEM3912I	Compiler Control	AB
IEM3914I	Compile-time Processor	AB
IEM4106I	Compile-time Processor	AS
IEM4109I	Compile-time Processor	AS
IEM4112I	Compile-time Processor	AS
IEM4115I	Compile-time Processor	AS
IEM4118I	Compile-time Processor	AS
IEM4121I	Compile-time Processor	AS,BC,BG
IEM4124I	Compile-time Processor	BC,BG
IEM4130I	Compile-time Processor	BG
IEM4133I	Compile-time Processor	BC
IEM4134I	Compile-time Processor	BC
IEM4136I	Compile-time Processor	BC
IEM4139I	Compile-time Processor	BC
IEM4142I	Compile-time Processor	BC
IEM4143I	Compile-time Processor	BC
IEM4148I	Compile-time Processor	BC
IEM4150I	Compile-time Processor	BC
IEM4151I	Compile-time Processor	BC
IEM4152I	Compile-time Processor	BC
IEM4153I	Compile-time Processor	BC
IEM4154I	Compile-time Processor	BC
IEM4157I	Compile-time Processor	BC
IEM4160I	Compile-time Processor	BC
IEM4163I	Compile-time Processor	BC
IEM4166I	Compile-time Processor	BC
IEM4169I	Compile-time Processor	BC
IEM4172I	Compile-time Processor	BC
IEM4175I	Compile-time Processor	BC
IEM4176I	Compile-time Processor	BC
IEM4178I	Compile-time Processor	BC
IEM4184I	Compile-time Processor	BC
IEM4187I	Compile-time Processor	BC
IEM4188I	Compile-time Processor	BC
IEM4190I	Compile-time Processor	BC
IEM4193I	Compile-time Processor	BC
IEM4196I	Compile-time Processor	BC
IEM4199I	Compile-time Processor	BC
IEM4202I	Compile-time Processor	BC
IEM4205I	Compile-time Processor	BC
IEM4208I	Compile-time Processor	BC
IEM4211I	Compile-time Processor	BC
IEM4212I	Compile-time Processor	BC
IEM4214I	Compile-time Processor	BC

Message Number	Logical Phase	Module
IEM4215I	Compile-time Processor	BC
IEM4217I	Compile-time Processor	BC
IEM4220I	Compile-time Processor	BC
IEM4223I	Compile-time Processor	BC
IEM4226I	Compile-time Processor	BC
IEM4229I	Compile-time Processor	BC
IEM4232I	Compile-time Processor	BC
IEM4235I	Compile-time Processor	BC
IEM4238I	Compile-time Processor	BC
IEM4241I	Compile-time Processor	BC
IEM4244I	Compile-time Processor	BC
IEM4247I	Compile-time Processor	BC
IEM4248I	Compile-time Processor	BC
IEM4250I	Compile-time Processor	BC
IEM4253I	Compile-time Processor	BC
IEM4254I	Compile-time Processor	BC
IEM4256I	Compile-time Processor	BC
IEM4259I	Compile-time Processor	BC
IEM4262I	Compile-time Processor	BC
IEM4265I	Compile-time Processor	BC
IEM4271I	Compile-time Processor	BC
IEM4277I	Compile-time Processor	BC
IEM4280I	Compile-time Processor	BC
IEM4283I	Compile-time Processor	BC
IEM4286I	Compile-time Processor	BC
IEM4289I	Compile-time Processor	BC
IEM4292I	Compile-time Processor	BC
IEM4295I	Compile-time Processor	BC
IEM4296I	Compile-time Processor	BC
IEM4298I	Compile-time Processor	BC
IEM4299I	Compile-time Processor	BC
IEM4301I	Compile-time Processor	BC
IEM4304I	Compile-time Processor	BC
IEM4307I	Compile-time Processor	BC
IEM4310I	Compile-time Processor	BC
IEM4313I	Compile-time Processor	BC
IEM4319I	Compile-time Processor	BC
IEM4322I	Compile-time Processor	BC
IEM4325I	Compile-time Processor	BC
IEM4326I	Compile-time Processor	AV
IEM4328I	Compile-time Processor	BC
IEM4331I	Compile-time Processor	BC
IEM43321	Compile-time Processor	BC
IEM4334I	Compile-time Processor	BC
IEM4337I	Compile-time Processor	BC
IEM4340I	Compile-time Processor	BC
IEM4343I	Compile-time Processor	BC
IEM4346I	Compile-time Processor	BC
IEM4349I	Compile-time Processor	BC
IEM4352I	Compile-time Processor	BC
IEM4355I	Compile-time Processor	BC
IEM4358I	Compile-time Processor	BC
IEM4361I	Compile-time Processor	BC
IEM4364I	Compile-time Processor	BC
IEM4367I	Compile-time Processor	BC
IEM4370I	Compile-time Processor	BC
IEM4373I	Compile-time Processor	BC
IEM4376I	Compile-time Processor	BC

Message Number	Logical Phase	Module
IEM4379I	Compile-time Processor	BC
IEM4382I	Compile-time Processor	BC
IEM4283I	Compile-time Processor	BC
IEM4391I	Compile-time Processor	BC
IEM4394I	Compile-time Processor	BC
IEM4397I	Compile-time Processor	BC
IEM4400I	Compile-time Processor	BC
IEM4403I	Compile-time Processor	BC
IEM4406I	Compile-time Processor	BC
IEM4407I	Compile-time Processor	BC
IEM4409I	Compile-time Processor	BC
IEM4412I	Compile-time Processor	BC
IEM4415I	Compile-time Processor	BC
IEM4421I	Compile-time Processor	BC
IEM4433I	Compile-time Processor	BG
IEM4436I	Compile-time Processor	BG
IEM4439I	Compile-time Processor	BG
IEM4448I	Compile-time Processor	BG
IEM4451I	Compile-time Processor	BG
IEM4452I	Compile-time Processor	BG
IEM4454I	Compile-time Processor	BG
IEM4457I	Compile-time Processor	BG
IEM4460I	Compile-time Processor	BG
IEM4463I	Compile-time Processor	BG
IEM4469I	Compile-time Processor	BG
IEM4472I	Compile-time Processor	BG
IEM4473I	Compile-time Processor	BG
IEM4475I	Compile-time Processor	BG
IEM4478I	Compile-time Processor	BG
IEM4481I	Compile-time Processor	BG
IEM4484I	Compile-time Processor	BG
IEM4499I	Compile-time Processor	BG
IEM4502I	Compile-time Processor	BG
IEM4504I	Compile-time Processor	BG
IEM4505I	Compile-time Processor	BG
IEM4506I	Compile-time Processor	BG
IEM4508I	Compile-time Processor	BG
IEM4510I	Compile-time Processor	BG
IEM4511I	Compile-time Processor	BC
IEM4514I	Compile-time Processor	BG
IEM4517I	Compile-time Processor	BG
IEM4520I	Compile-time Processor	BG
IEM4523I	Compile-time Processor	BG
IEM4526I	Compile-time Processor	AS
IEM4529I	Compile-time Processor	BC,BG
IEM4532I	Compile-time Processor	AS
IEM4535I	Compile-time Processor	AS
IEM4547I	Compile-time Processor	AV
IEM4550I	Compile-time Processor	BG
IEM4553I	Compile-time Processor	BG
IEM4559I	Compile-time Processor	BG
IEM4562I	Compile-time Processor	BG
IEM4570I	Compile-time Processor	BG
IEM4572I	Compile-time Processor	BG
IEM4574I	Compile-time Processor	BG
IEM4576I	Compile-time Processor	BG
IEM4578I	Compile-time Processor	BG
IEM4580I	Compile-time Processor	BG

This appendix describes, for the Compile-time Processor Logical Phase, the internal formats of text and tables, communication region use, system interfaces and compiler control interfaces.

1. INTERNAL FORMATS OF TEXT

The internal format of text used by the compile-time processor is EBCDIC. As source input is read into storage, non-macro text is moved directly into text blocks after translation to internal format. Encoded compile-time statements and line numbers are also placed in text blocks.

```
      Byte 1   Byte 2     Byte 3      Byte 4
Word ┌───────┬──────────┬───────────────────┐
 1   │LENGTH │PROC NO.  │HASH-CHAIN-POINTER │
     ├───────┼──────────┴───────────────────┤
 2   │ TYPE  │     VALUE/VALUE-POINTER      │
     ├───────┴──────────┬───────────────────┤
 3   │      FLAGS       │       COUNT       │
     ├──────────────────┴───────────────────┤
 4   │        POINTERS/PARM-TYPES           │
     ├──────────────────────────────────────┤
 5   │EBCDIC NAME ------>                   │
     └──────────────────────────────────────┘
     ~                                      ~
     ├──────────────────────────────────────┤
 n   │------> EBCDIC NAME                   │
     ├──────────────────────────────────────┤
```

Format of a Dictionary Entry

The compile-time processor uses a set of chained dictionary entries. Hashing techniques are used to add an item to the dictionary or to search for an entry. A compile-time processor dictionary item is a variable-length item with the following skeletal format:

The fields defined in this skeleton have the following meaning and usage:

LENGTH:
The length of the EBCDIC name. If the item has no name (e.g., a constant) this field is zero.

PROC NO.:
The number assigned to the procedure in which the identifier was declared. Each procedure is assigned a unique number. The identifiers in the non-procedural text are given the procedure number 1. The built-in function SUBSTR is given the procedure number 0.

HASH-CHAIN-POINTER:
The dictionary address of the next item on this hash chain. This address is zero if no item follows.

TYPE:
A byte which gives the attributes of the entry. The bits (if on) have been assigned the following meanings:

Bit	Meaning
0	fixed
1	character
2	bit
3	entry
4	label
5	INCLUDE identifier
6	iterative DO
7	constant

VALUE/VALUE-POINTER:
If the item is fixed, this contains the value proper stored as a five-digit packed decimal number. Otherwise it contains a pointer to the value stored in IVBs. The definition of value for the various kinds of entries is given below. For a fixed macro variable, this contains the value. For a character variable, it contains a pointer to IVBs containing the value. For a procedure, it points to the text-block location of the code. For a label, it points to the text-block location of the label. If references to the label are found before the label is discovered, the value pointer temporarily points to a chain of IVBs with a description of every GOTO transferring to this label. This information is processed and discarded when the label is found. For an INCLUDE identifier, it points to the beginning of the included text.

FLAGS:
This set of bits provides additional information about the use of the item. They are used as follows:

Bit	Meaning
0	special entry bit
1	DECLARE encountered (Phase BC)
2	procedure body encountered (Phase BC)
3	parameter
4	used to indicate a procedure called by Phase II scan.
5	DECLARE encountered (Phase BG)
6	unused
7	ACTIVATE bit
8	"in-use" bit
9	"indirect reference" bit
10	"undefined" bit for multiple declarations
11	left-hand side (LHS bit)

This field occupies a half-word.

COUNT:
For a procedure entry, this field contains a count of the number of parameters for the procedure. For INCLUDE identifier it is zero initially, and subsequently contains the initial line number assigned to the included text.

POINTERS/PARAM-TYPES:
For a procedure, the field contains an encoding of the type information for each formal parameter. Two bits are reserved for each parameter. One indicates fixed; the other indicates character. If neither bit is set, this indicates that the entry declaration did not specify attributes for the parameters.

For a label, word 4 contains two pointers to dictionary items. One points to the dictionary entry for the immediately embracing iterative DO. The second half-word contains a pointer to the dictionary entry for the immediately embracing INCLUDE. This provides a method of checking the legality of GOTOs. For an INCLUDE identifier, only the pointer to the immediately embracing INCLUDE is kept.

During Phase I, word 4 is used for labels and simple variables to hold two pointers. These form a bi-directional chain of all labels and variables having the same procedure number which have been used but not defined. This information is used only in Phase I and can therefore be overlaid.

EBCDIC NAME:
A variable length field, containing the EBCDIC name of the item. If the item has no name, this field is not included.

Format of an Identifier Value Block (IVB)

To hold character and bit string values, some text blocks are organised into sub-blocks of 32 bytes each. Of these 32 bytes, 27 are used to hold values or parts of values. The first byte is used to hold a copy of the last character in the preceding IVB. This copy is made to facilitate backup. The last four bytes consist of a condition code of one byte followed by a 3-byte chain pointer. A set of these sub-blocks, chained together, is used to hold a value. The condition byte is 27 for all except the last sub-block in a value. In this last condition code byte the first bit is set to 1 to indicate "end-of-value." The remaining bits are a count of the significant bytes in the sub-block. There is a maximum of 27 significant bytes in an IVB.

The chain address is used to point to the next sub-block in a value. The meaning of the chain address in the last sub-block in a chain depends on how the chain is being used.

These small chained sub-blocks are referred to as "identifier value blocks," or IVBs.

Text blocks are allocated to hold IVBs as the need arises. Those IVBs not currently in use are chained together into an availability chain and are re-used when needed.

An example of a character string value held in IVBs is shown. The character string, which starts with AB and ends with CD, is 28 characters long. Two IVBs are thus required to hold the value. The string AB...C is put into the first IVB, while the last character, D, is put into another IVB. The condition code byte of the first IVB is 27. The second condition code byte is 10000001. The first "1" indicates end-of-value, while 0000001 is a count of the significant characters in the IVB.

```
┌──────────┬─┬─┐    ┌──────────┬────────┬───┐
│?AB ... C │0│----->│CD/////   │10000001│ ? │
└──────────┴─┴─┘    └──────────┴────────┴───┘
 <---28---> 1 3      <--28-->   <--1-->  <-3->
```

Besides holding character-string values, IVBs are used in many places by the compile-time processor to hold information which must be chained from a dictionary entry and which is of indefinite length. These uses are noted elsewhere.

Instruction Codes for the Compile-time processor

Compile-time statements are handled in two parts. During Phase BC, each statement is recognized and syntax checked. An encoded form of the statement is then placed into the current text block. During Phase BG these encoded statements are executed by an interpreter.

All expressions are encoded in postfix Polish. A stack is used during Phase II to hold all operands. Conversions are done in Phase BG.

Thus the expression (A+B)||C, for example, is turned into

A B + C||

To be more explicit, it is turned into the instructions

PUSH A;

PUSH B

ADD;

PUSH C;

CONCAT

The PUSH operator pushes its operand onto the phase II stack. This stack consists of 150 full words in scratch storage. The first byte of each call is a status byte; the last three bytes hold the value if the item is FIXED, a pointer if the item is CHARACTER or BIT, or an indirect reference to a dictionary entry if the indirect bit is on.

The bits of the status byte have the following meaning if set to one:

Bit	Meaning
0	FIXED
1	CHARACTER
2	BIT
3	Indirect reference (i.e., points to a dictionary entry)
4	Character string value does not "belong" to the stack and should not be erased when stack is popped. (Shared with Phase BG scan.)

Bits 6-8 are unused by the interpreter. They are reserved for Phase BG scan.

All instructions generated by the Phase BC code generators begin with an operation byte. Depending on the operation, it may be followed by zero or more bytes of information which are intrinsically part of the instruction. Each instruction may have either or both of the characteristics STACK and FIXED. The definition of these characteristics follows:

1. STACK. These instructions consist only of the one-byte operator. They take their operands, if any, from the Phase II stack. These operators correspond in general to the PL/I arithmetic and string operators. Depending on whether they are unary or binary, they use the top one or two items on the stack. Before these operands are used, they are converted, if necessary, <u>in place</u> to the required type. After the items are used they are popped from the stack. The result of the operation is pushed onto the stack.

   The conversion, the popping, and the pushing are all implied for a stack operator.

2. FIXED LENGTH. These operations are followed by a fixed number of bytes -- usually two. These bytes, which usually refer to a dictionary entry, serve as the operand(s) of the instruction.

The table below shows the operations that are to encode macro instructions. The operand description indicates only the general operand type for a variable-length item. The count byte is omitted.

<table>
<tr><th>Mnemonic</th><th>Type</th><th>Operand Description</th><th>Function</th></tr>
<tr><td>ADD</td><td>STACK</td><td>BINARY; OPERANDS, RESULT FIXED</td><td>A+B</td></tr>
<tr><td>SUB</td><td>STACK</td><td>BINARY; OPERANDS, RESULT FIXED</td><td>A-B</td></tr>
<tr><td>MUL</td><td>STACK</td><td>BINARY; OPERANDS, RESULT FIXED</td><td>A*B</td></tr>
<tr><td>DIV</td><td>STACK</td><td>BINARY; OPERANDS, RESULT FIXED</td><td>A/B</td></tr>
<tr><td>UNMIN</td><td>STACK</td><td>UNARY; OPERAND, RESULT FIXED</td><td>-B</td></tr>
<tr><td>UNPLS</td><td>STACK</td><td>UNARY; OPERAND, RESULT FIXED</td><td>+B</td></tr>
<tr><td>ASSIGN[1]</td><td>STACK
FIXED</td><td>UNARY; B CONVERTED TO TYPE OF A</td><td>A=B (assignment)</td></tr>
<tr><td>NOT</td><td>STACK</td><td>UNARY; OPERAND, RESULT BIT</td><td>¬B</td></tr>
<tr><td>AND</td><td>STACK</td><td>BINARY; OPERANDS, RESULT BIT</td><td>A&B</td></tr>
<tr><td>OR</td><td>STACK</td><td>BINARY; OPERANDS, RESULT BIT</td><td>A|B</td></tr>
<tr><td>CONCAT</td><td>STACK</td><td>BINARY; OPERANDS, RESULT CHAR</td><td>A||B</td></tr>
<tr><td>EQU[2]</td><td>STACK</td><td>BINARY; OPERANDS, RESULT VARY</td><td>A=B (equality)</td></tr>
<tr><td>GT[2]</td><td>STACK</td><td>BINARY; OPERANDS, RESULT VARY</td><td>A>B</td></tr>
<tr><td>LT[2]</td><td>STACK</td><td>BINARY; OPERANDS, RESULT VARY</td><td>A<B</td></tr>
<tr><td>INC</td><td>FIXED</td><td>Two-byte dictionary reference</td><td>INCLUDE A</td></tr>
<tr><td>ABORT</td><td>FIXED</td><td>One-byte code</td><td>ABORT processing</td></tr>
<tr><td>TRA</td><td>FIXED</td><td>Two-byte dictionary reference</td><td>Transfer to label</td></tr>
<tr><td>TRAC</td><td>FIXED</td><td>Two-byte dictionary reference</td><td>Transfer to label</td></tr>
<tr><td>TRAF[3]</td><td>STACK
FIXED</td><td>Two-byte dictionary reference</td><td>Transfer to label
if top of stack false.</td></tr>
<tr><td>INV[5]</td><td>STACK
FIXED</td><td>Two-byte dictionary reference and
a one-byte argument count</td><td>Invokes the procedure</td></tr>
<tr><td>TRAI[4]</td><td>FIXED</td><td>two two-byte dictionary references</td><td>Transfer out of INCLUDE</td></tr>
<tr><td>PUSH</td><td>FIXED</td><td>Two-byte dictionary reference</td><td>Push A onto stack</td></tr>
<tr><td>PUSHI</td><td>FIXED</td><td>Two-byte dictionary reference</td><td>Push address of A
onto stack</td></tr>
<tr><td>UPDT</td><td>FIXED</td><td>Three-byte line count</td><td>Put line count into LINCNT</td></tr>
<tr><td>ENTM</td><td>FIXED</td><td>no operand</td><td>Enter interpreter</td></tr>
<tr><td>RTNS</td><td>FIXED</td><td>no operand</td><td>Return to Phase II scan</td></tr>
<tr><td>ENB</td><td>FIXED</td><td>Two-byte dictionary references</td><td>ACTIVATE A</td></tr>
<tr><td>DSB</td><td>FIXED</td><td>Two-byte dictionary references</td><td>DEACTIVATE A</td></tr>
<tr><td>DCL</td><td>FIXED</td><td>Dictionary reference</td><td>DECLARE A</td></tr>
<tr><td>NOPD</td><td>FIXED</td><td>Dictionary reference</td><td>No-ops the DECLARE, once
executed</td></tr>
</table>

Mnemonic	Type	Operand Description	Function
CVT[6]	FIXED	Dictionary reference	Convert to RETURNS attribute
RETN[7]	FIXED	Dictionary reference	Return from procedure A

[1]The ASSIGN operator does not push a result. The expression result is found on the PDS and is popped; the dictionary reference for the left hand side is the single argument.

[2]Operand conversion for EQU, GT, and LT is as specified in IBM System/360 Operating System: PL/I Language Specifications.

[3]The TRAF uses and pops the top operand on the stack. It is treated as a bit string for conditional transfers.

[4]This handles GOTOs out of included text. At this point CLNUP is performed. The arguments are (a) the dictionary entry for the label to which control is to pass; and (b) the dictionary entry for the current INCLUDE.

[5]The arguments for the invocation are contained on the stack. The dictionary reference is to the procedure entry.

[6]This converts the top of the stack to the attributes specified in the RETURNS attribute for the procedure A.

[7]This terminates the invocation of procedure A and converts the value on the top of the stack to the attribute specified on the PROCEDURE statement.

2. COMMUNICATIONS REGION USE

The region from offset 0 to offset 304 (ZCOMM) is used as a general communications region throughout the compiler, including the compile-time processor. The region from ZCOMM to ZCOMM+463 is also used by the compiler; however, during the compile-time processor phase, this region is used exclusively by the compile-time processing. The details of this usage are shown below.

Name	Dec. Offset	Length	Contents
STATUS	ZCOMM	1	Byte 1: Bit 0 not used 1 PROCSW -- processing macro procedure 2 FINDBIT -- SRHDIC has found dictionary item 3 ERSW -- diagnostic produced in Phase II 4 EFSW -- end of file encountered (input) 5 LEVBIT -- processing IVB 6 INCSW -- processing included text 7 PH2SW -- in Phase II Note: Condition Settings "1" = set "0" = off
STA2	ZCOMM+1	1	Byte 2: Bit 0 OLDINC -- processing already listed INCLUDE 1 SKPSW -- indicates entry to END from PRCSCN 2 NOPERCENTSW -- look ahead for % completed 3 SYSOPN -- SYSLIB DCB is open 4 MACRO -- indicates current macro action 5 PR2SW -- indicates in macro procedure 7 ARG -- indicates that Phase II is looking for arguments of activated procedure
SUBSTRDR	ZCOMM+2	2	Holds dictionary reference of 0 level SUBSTR entry
TOKPTR	ZCOMM+4	4	Address of character being scanned, text reference or absolute, right justified

Name	Dec. Offset	Length	Contents
INCPTR	ZCOMM+8	4	Save area for TOKPTR
INBUF	ZCOMM+12	4	Absolute address of 132-character input buffer, right justified
OUTBUF	ZCOMM+16	4	Absolute address of output buffer, right justified
PDSPTR	ZCOMM+20	4	Absolute address to top of pushdown stack, right justified
ENDBUF	ZCOMM+24	4	Absolute address to last significant character in input buffer, right justified
WHERE	ZCOMM+28	4	Address of next available byte in output buffer, text reference or absolute, right justified
IVBPTR	ZCOMM+32	4	Text reference to next free IVB, right justified
LINCNT	ZCOMM+36	4	Holds current line number, right justified
TEMPTR	ZCOMM+40	2	Dictionary reference to top of "in-use" temporary stack
DCENTY	ZCOMM+42	2	Dictionary reference for chaining dictionary items
CURINC	ZCOMM+44	2	Dictionary reference to INCLUDE entry being processed
CURDO	ZCOMM+46	2	Dictionary reference to DO entry being processed
PROCNO	ZCOMM+48	1	Current procedure number, right justified
NXTPC	ZCOMM+49	1	Next available procedure number, right justified
DPHCNT	ZCOMM+50	2	Current depth count
CODE	ZCOMM+52	1	Code for token type
LENGTH	ZCOMM+54	2	Number of significant characters in TOKBUF, right justified
MXDPTH	ZCOMM+56	2	Integer value of depth of replacement, right justified
INDEX	ZCOMM+58	2	Hash table index for dictionary routines
ATTR	ZCOMM+60	2	"Type" byte for dictionary routines
GRSAVE	ZCOMM+64	4	Save area for GRG
NEWIVB	ZCOMM+68	4	Pointer to IVB chain to be freed or obtained
VALUE	ZCOMM+72	4	Type and value/value pointer for dictionary entries
PREINB	ZCOMM+76	4	Pointer to header information for INBUF
BUFSRT	ZCOMM+80	4	Pointer to left margin in INBUF
INIVB	ZCOMM+84	1	Current busy block number
OUTIVB	ZCOMM+85	1	Current busy block number
TXTBLK	ZCOMM+86	1	Current busy block number
INVBAB	ZCOMM+88	4	Current block used in absolute address calculation
OUTIVBAB	ZCOMM+92	4	Current block used in absolute address calculation

Name	Dec. Offset	Length	Contents
TXTBLKAB	ZCOMM+96	4	Current block used in absolute address calculation
MTABC	ZCOMM+100	4	Address of translate table for TOKSCN and FINDPC
TXTEST	ZCOMM+104	4	Length of text block adjusted for chain address
BUF1	ZCOMM+108	4	Pointer to first INCLUDE buffer (Not used)
BUF2	ZCOMM+112	4	Pointer to second INCLUDE buffer (Not used)
LIBDCB	ZCOMM+116	4	Pointer to DCB for SYSLIB data set
USRDCB	ZCOMM+120	4	Pointer to DCB for user data sets
MAXLCT	ZCOMM+124	4	Maximum line count used so far
PRCWHR	ZCOMM+128	4	Pointer to next byte in which to put procedure text
DCENTYAB	ZCOMM+132	4	Absolute address of dictionary entry
SCHK	ZCOMM+136	4	Pointer to level 1 SUBSTR entry
PROCCL	ZCOMM+140	2	Dictionary reference of procedure check list
OUTERCL	ZCOMM+142	2	Dictionary reference of outer check list
PROCCLDR	ZCOMM+144	2	Dictionary reference for PROCCL cell
OUTRCLDR	ZCOMM+146	2	Dictionary reference for OUTERCL cell
DECIDR	ZCOMM+148	4	Dictionary reference of dictionary entry for DECIMAL 1
CURPRC	ZCOMM+152	4	Pointer to current procedure entry on PDS
TOKBUF	ZCOMM+164	32	32-byte buffer, characters inserted left justified
HASTB	ZCOMM+300	128	64 two-byte dictionary references to hash chains for named items
CONSCH	ZCOMM+428	2	Dictionary reference to constant chain
SPECCH	ZCOMM+430	2	Dictionary reference to special chain -- debugging only

3. COMPILE-TIME PROCESSOR, TIME SHARING SYSTEM, AND COMPILER CONTROL INTERFACES

Although the compile-time processor makes considerable use of the time sharing system facilities, it usually does so indirectly through the compiler control. However, those time sharing system services required to support the INCLUDE facility are invoked directly. Since included text is required to be a member of a partitioned data set, it is those data management facilities which support VPAM which are used. Specifically the macros OPEN, FIND, CLOSE, and GET are used by various parts of the INCLUDE handler. Details of these macros can be found in IBM System/360 Time Sharing System: Assembler User Macro Instructions.

The root phase is invoked by the compiler control if the MACRO option is specified. All subsequent communication between the compile-time phases and the compiler control is done by way of cells in the communications region. This includes the parameters passed to the compiler service routines, the decoded options which are tested, and the cells set to indicate the status of source margins and mode (EBCDIC) of the output.

Specifically, the following cells in the communications region are either used or set:

PAR1

PAR2

ZTV

ZMYNAM

MCSIZE

CCCODE

TXTSZ

ZSOR -- column number in which to begin scan of input text

ZMAG -- column number in which to end scan of input text

ZTRAN1

The following compiler control routines are referenced:

ZUPL	RELESE
ZURD	RLSCTL
ZUGC	ZTXTRF
ZUTXTC	ZTXTAB
ZURC	ZCHAIN
ZABORT	ZALTER
ZLOADW	ZDABRF
ZDICRF	ZEND
ZUERR	ZUBW
ZDRFAB	

The purpose of these routines is to permit the user to build, scan and otherwise manipulate tables in text blocks without any concern for physical block boundaries, status of text blocks or maintaining pointers to first, last and current table entries. The routines also handle text, which is assumed to be a special type of table.

The user may:

1. Define a table by using the IEMKTCA macro to set up a TCA (task communication area) control block. The address of the TCA is always passed to a table handling routine and identifies the table concerned. Most TCAs will be held in the local communications region in phase KA.

2. Add new entries to the end of a table. Table entries may be of fixed or varying lengths. For fixed length entries, the length is held in the TCA for the table. For varying length entries, the TCA contains information enabling the routine to determine the length of the entry. Fixed length entries may be built in storage and moved into the table by the routines, or space allocated for an entry by the routines and the entry built directly into the table. For varying length entries, the entry must be built in storage and moved into the table by the routines.

3. Scan a table either forwards or backwards. The user requests the address and text reference of the 'next' table entry. The user may position a scan to the start or end of a table, or to some intermediate point.

4. Reference individual table entries at random. This may be done while a sequential scan of the table is being performed, and will not affect the scan.

5. Specify that a table is to be deleted.

6. Specify that entries are to be 'locked in'. This means that the absolute address of a table entry will remain valid until the entry is explicitly or implicitly unlocked, or the table is deactivated. (See 7). If an entry is not 'locked in', any subsequent call to the table handling routines may render the absolute address returned for the entry invalid. The current entry of a sequential scan, either creating or reading, is automatically 'locked in'.

   A randomly referenced entry is only locked by an explicit lock request.

   A current scan entry may be unlocked explicitly by deactivating the table, or implicitly by making or requesting the next entry, or repositioning the scan.

   A random entry may be explicitly unlocked, or implicitly unlocked by another random reference to the same table specifying lock.

   All locks are released when a table is deactivated. The total number of locked entries for all tables must not exceed four at any one time.

7. Activate or deactivate a table. All tables are initially deactivated. Tables are always activated implicitly, initially by a request to add an entry to the table, and subsequently by any valid request for an operation to be performed on that table.

   A table may be deactivated by an explicit request. Deactivation causes all locked entries to be unlocked, and renders all absolute addresses of entries in that table invalid. Sequential scans are not otherwise affected.

   A table is implicitly deactivated by a request to free the table (as in 5), or if the table contains no locked entries and any call is made to the table handling routines.

DESCRIPTION AND FORMAT OF MACRO INSTRUCTIONS

The IEMKTCA Macro

This macro is used to set up a TCA (task communication area) control block describing a table to be processed by the table handling routines. The macro has two functions:

1. Sets up global variables describing tables which are used by the IEMKTAB macro to generate appropriate linkages to the table handling routines.

2. Used with the R operand to set up a TCA control block the address of which is passed to the table handling routines to identify the table. The 'table identifier' is the same as the 'table2 identifier' used in IEMKTAB macro instructions. It must also be the label of a fullword containing, at execution time, the address of the TCA set up by a IEMKTCA macro instruction with the R operand.

Format:

```
                                        (F)
   table identifier IEMKTCA [,R,]ET=(V)
                                        (T)
     [,L=entry length][,OPS=[S][R]]

       [,NPTRS=no of scan pointers.]
       [,DLF=displacement to length field]
```

Description of Parameters

ET=entry type
    This parameter indicates the type of entry contained in the table, as follows:
    F - fixed length entries
    V - variable length entries
    T - text

L=entry length
    This parameter is required if ET=F is coded, and indicates the length of an entry.

OPS=operations
    This parameter indicates the type of operations to be performed on the table, as follows:
    S - sequential scans will be performed
    R - random references will be made

NPTRS=n
    This parameter permits more than one sequential scan of one table to be made at one time. 'n' indicates the maximum number of sequential scans which will be in progress at any one time. An individual scan is identified by coding PTR=n in the SET, SETZ, or SCAN operation (see IEMKTAB macro). The default value is n=1 if OPS=S or RS is coded, or OPS is omitted, otherwise n=0. If ET=T is coded, this parameter must be omitted and only one scan is permitted.

DLF=dist. to length field
    This indicates the displacement from the first byte of the entry to the two byte field containing the length of the entry. It must be coded if ET=V is coded.

R
    This parameter is supplied only if actual code is to be generated from the macro instruction. It provides a label for the TCA which may be used in an A type address constant having the label identifier as its label.

Example of use:

```
TABLE1 DC A(ATAB1)

TABLE1 DTCA ATAB1,ET=F, L=8, OPS=S
```

Note: The label on the IEMKTCA macro instruction statement is not made the label of any generated statement, so no multiple definition will result. Its only use is to provide a link between the two macros IEMKTCA and IEMKTAB.

The IEMKTAB Macro

This macro specifies operations to be performed on a table or tables. The table to be operated on is always specified by supplying the address of its TCA.

Format:

```
[label] IEMKTAB code, parameters
```

Note: 'code' specifies the type of operation, and the parameters depend on this as shown below:

Code	Parameters
BLDC	Table Identifier, address of entry skeleton [,AATO=] [,SATO=]
BLDT	Table Identifier [,AATO=] [,SATO=]
DR	Table Identifier, SA=[,AATO=](N) [,OPT=(L)]
ULDR	(Table Identifier,...) or Table Identifier
SET	Table Identifier, SA=[,PTR=n]
SETZ	Table Identifier [,OPT=(B)] (F) [,PTR=n]
SCAN	Table Identifier [,AATO=] [,SATO=] [,OPT=(V)] (F) [,PTR=n][,ETA=] [,TRTAB=,FBTO=] [,PSATO=]
FREE	(Table Identifier,...) or Table Identifier
DEACT	(Table Identifier,...) or Table Identifier
TEST	Table Identifier, NTA=

Notes: The 'Table Identifier' operand must appear in the label field of a IEMKTCA macro instruction physically preceding the IEMKTAB macro instruction. The last three operands of the SCAN operation (TRTAB, FETO, PSATO) only apply to text tables ET = T in IEMKTCA).

Description of Keyword Parameters

AATO
: A register designation or address of a fullword in which the returned absolute address is to be placed.

SATO
: Address of a three byte field in which the returned symbolic address is to be placed.

SA
: Address of a three byte area containing a symbolic address.

OPT
: Options applying to this operation. The option letters may appear in any order.

PTR
: Specifies the pointer which identifies the current record of the scan. 'n' must not exceed the number specified in the NPTRS parameter of the IEMKTCA macro instruction. C indicates the end of table pointer for creating new entries.

ETA
: Specifies the address of a routine to be given control when the end of a table is detected during a sequential scan.

TRTAB
: Address of translate table for selective scan.

FBTO
: Location or register in which non-zero function byte from translate operation is to be placed.

PSATO
: Address of a three byte area into which the symbolic address of the previous entry is to be placed.

NTA
: Address of routine to receive control if table is null.

Note: A register designation (absolute expression in parentheses) identifying general registers 2-9 may be used in the AATO, SATO, SA, ETA, FBTO, PSATO and NTA operands.

Description of Table Handling Operations

BLDC
: Adds an entry to the end of a table. The entry is built in storage by the user and moved to the table by the table handling routines. The routines return the symbolic and absolute addresses of the new entry.

BLDT
: The table handling routines allocate space for a new entry at the end of the specified table, and return the absolute and symbolic addresses of the space. The user builds his entry in the space allocated. This operation can only be specified if ET=F was coded in the IEMKTCA macro instruction.

DR
: Direct (random) reference. The entry is identified by the SA parameter. The absolute address of the entry is returned. If OPT=L is specified, the entry is locked in.

ULDR
: The last directly referenced entry for this table that specified OPT=L, is unlocked.

SET
: The sequential scan is positioned at the entry identified by the SA parameter. The next SCAN operation causes the absolute address of this entry to be returned. PTR=n (see SETZ).

SETZ
: The sequential scan is positioned to the beginning of the table if the OPT parameter is omitted or OPT=F is coded, or to the end of the table if OPT=B is coded. The next SCAN operation returns the absolute address of the first or last table entry to be returned. The PTR=n parameter indicates which sequential scan is meant if more than one is in progress at one time. 'n' may not exceed the number specified in the NPTRS parameter of the IEMKTCA macro instruction. PTR=1 is assumed if the parameter is omitted. OPT=B may not be coded if ET=V or T is coded in the IEMKTCA macro instruction for the table.

SCAN
: The symbolic and absolute addresses of the next entry in the table are returned. If a SET or SETZ was the last operation, the next entry is that pointed to by the SET or SETZ operations. The options F or B indicate:

F a forward scan is required. This is the default.

B a backward scan is required. This may only be specified if ET=F is coded in the IEMKTCA macro instruction.

Selective Scanning Facilitiy

This facility is available for text tables only (ET=T in IEMKTCA macro instruction). The TRTAB operand identifies a 256 byte translate table which is used to translate the code byte of the entry. If the result is zero, the scan continues until the routine exits to the ETA address. If the result is non-zero, it is placed in the register or location identified by the FBTO operand, and control returns to the user, the AATO and SATO operands identifying the selected entry.

FREE
: One or more tables are completely freed and deactivated. The next operation on the table must be a BLDC or BLDT.

DEACT
: The table or tables are deactivated. All locked entries are unblocked and all absolute addresses of entries rendered invalid.

TEST
: Tests for a null table. If table is null, control is passed to routine identified by NTA pointer.

TRANSFER VECTOR TABLE

Entry to the various compiler control routines is via a transfer vector. Details of the transfer vector appear below.

Hex. Offset	Name	Description
8	ZUPL	Print a line
C	ZURD	Read a card
10	ZUGC	Get scratch storage
14	ZUTXTC	Get text block
18	ZURC	Release scratch storage
1C	ADDLST	Initialization List
20	ZABORT	Dump and go to error message routines
24	ZLOADW	Load and return to caller
28	ZDICAB	Make dictionary entry. Absolute address returned
2C	ZDICRF	Make dictionary entry. Dictionary reference returned
30	ZUERR	Make error message entry
34	ZDRFAB	Convert dictionary reference to absolute address
38	ZLOADX	Load with overlay and return to caller
40	REQUEST	Give a list of phase names required or not wanted for this compilation
44	RELESE	Release all named phases
48	RLSCTL	Release all named phases and pass to next phase
4C	ZDUMP	Dump specified main storage and continue
50	ZTXTRF	Converts absolute address to text reference
54	ZTXTAB	Convert text reference to absolute address
58	ZCHAIN	Find next block in chain
5C	ZALTER	Change text block status
60	ZDABRF	Convert absolute address to dictionary reference
64	ZNALRF	Make unaligned dictionary entry. Reference returned
68	ZNALDB	Make unaligned dictionary entry. Absolute address returned
6C	ZEND	Terminate job
70	ZULF	Write on load file
74		
78	IEMAC	Intermediate file routine
80	RLSCTLX	Release all named phases and hand control to the next phase, after having loaded it with overlay
84	RECONS	Reconstitute instructions in IEMAL
88	DYNAMIC	Pass control to the dynamic dump routines, if required
8C	IEMAL	Address of second control phase
90	FOOTPRNT	Indicates which output modules are loaded and that compiler is invoked
94	IEMAF	DEFAULT options
98	RESURCT	Routine to clean-up in case of reinvocation after interrupting out

Name	Hex. Offset	Details
ADDLST	1C	Description: Initializes a list of address in second table of AA Parameters: None
RELESE	44	Description: Deletes list of phases Parameters Passed PAR1 byte 0 unused 1-3 address of list of phases to be deleted. The list of phase names, each of two characters, is terminated by the name ZZ Parameters Returned: None
REQEST	40	Description: Marks phases as 'wanted' or 'not wanted' Parameters Passed PAR1 byte 0 unused 1-3 address of list of phases to be marked as 'wanted' PAR2 byte 0 unused 1-3 address of list of phases to be marked as 'not wanted' Parameters Returned: None Note: A phase list containing only 'ZZ' is effectively a null list. During module AM phases of the compiler are all marked as either normally loaded or not loaded. A phase which is normally not loaded is only loaded if it has previously been marked as 'wanted'. A phase which is normally loaded will always be loaded unless it has previously been marked as 'not wanted'
RESURCT	98	Description Performs clean-up on compiler reinvocation. Examines footprint and unloads all loaded output modules, except control output module; closes and releases load and macro files, if they are open; reinitializes code in control output module. Parameters Passed Examines FOOTPRNT at hex offset 90 in transfer vector table.
RLSCTL	48	Description: Deletes a list of loaded phases and passes control to next phase Parameters Passed PAR1 byte 0 unused 1-3 address of list of phases to be deleted before next phase is loaded PAR2 byte 0 unused 1-3 (a) zero, if next phase is to be taken from compiler control phase directory (b) address of phase name to be loaded next Parameters Returned (to next phase) PAR1 byte 0 unused 1-3 address of new phase load point Note: List of phases given by the address in PAR1 is deleted. Then the next phase is selected and loaded and control is passed to a point two bytes from the load point of the new phase

Name	Hex. Offset	Details
RLSCTLX	80	<u>Description</u>: Releases all named phases and passes control to next phase. The next phase may be loaded more than once <u>Parameters Passed</u> PAR1 byte 0 unused 1-3 address of list of phases to be deleted before next phase is loaded PAR2 byte 0 unused 1-3 (a) zero of next phase is to be taken from compiler control phase directory (b) address of phase to be loaded next <u>Parameters Returned</u> (to next phase) PAR1 byte 0 unused 1-3 address of load point of new phase <u>Notes</u>: 1. List of phases given by the address in PAR1 is deleted. The next phase is selected and loaded and control is passed to a point two bytes from the load point of the new phase 2. The entry point RLSCTLX does not cause the compiler control routines to advance the pointer in the table of phases still to be loaded, and does not cause the phase to be marked as already loaded once
ZABORT	20	<u>Description</u>: Deletes currently loaded phases (after dumping if DP SPECIFIED IN PARAMETER) AND PASSES CONTROL TO Error Editor <u>Parameters Passed</u>: None <u>Parameters Returned</u>: None
ZALTER	5C	<u>Description</u>: Changes text clock status <u>Parameters Passed</u> PAR1 byte 0 unused 1 text block number 2-3 unused PAR2 bytes 0-2 unused 3 status required bits 0-3 unused 4 X'4' 'busy' 5 X'3' 'wanted' 6 X'2' 'not wanted' 7 X'1' 'free' <u>Parameters Returned</u>: PAR2 is unaltered and may be used in successive calls without reloading <u>Note</u>: 1. Terminology: 'busy' - lock into storage i.e., address preserved 'Wanted' - information required, do not spill unless necessary 'Not wanted' - information required, block may be spilt 'Free' - information no longer required

Name	Hex. Offset	Details
ZCHAIN	58	Description: Finds next text block in chain Parameters Passed PAR1 byte 0 unused 1 current block number 2-3 unused PAR2 bytes 0-2 unused 3 status required for old block Parameters Returned PAR1 byte 0 unused 1-3 text reference of start of new block PAR2 byte 0 unused 1-3 absolute address of start of new block Note: The new text block is marked as busy
ZDABRF	60	Description: Converts absolute address to dictionary reference Parameters Passed PAR1 bytes 0-1 unused 2-3 any reference in the same dictionary block PAR2 byte 0 unused 1-3 absolute address to be converted Parameters Returned PAR1 bytes 0-1 unused 2-3 dictionary reference corresponding to absolute address Notes: 1. No check is made that this address is the start of a dictionary entry, or that it is any other specific point 2. No check is made that the address is in the same block as the dictionary reference passed. If this is the case, that is, the parameters passed are in error, the dictionary reference returned may not correspond to the address
ZDICAB	28	Description: Makes aligned dictionary entry and returns absolute address Parameters Passed PAR1 byte 0 unused 1-3 address of entry (as built by calling phase) PAR2 bytes 0-1 unused 2-3 length of entry (binary) Parameters Returned PAR1 byte 0 unused 1-3 dictionary entry address PAR4 bytes 0-1 unused 2-3 reference to some point in the same dictionary block Notes: 1. The entry built is constructed complete with code byte and length fields. The length passed in PAR2 is the length of the complete entry 2. ZDICRF performs the same function and returns more information, with no loss in efficiency

Name	Hex. Offset	Details
ZDICRF	2C	<u>Description</u>: Makes an aligned dictionary entry and returns its dictionary reference and absolute address <u>Parameters Passed</u> PAR1 byte 0 unused 1-3 address of entry PAR2 bytes 0-1 unused 2-3 length of entry (binary) <u>Parameters Returned</u> PAR1 bytes 0-1 unused 2-3 dictionary reference of entry PAR4 byte 0 unused 1-3 dictionary entry address <u>Note</u>: See ZDICAB
ZDRFAB	34	<u>Description</u>: Converts dictionary reference to absolute address <u>Parameters Passed</u> PAR1 bytes 0-1 unused 2-3 dictionary reference <u>Parameters Returned</u> PAR1 byte 0 unused 1-3 absolute address
ZDUMP	4C	<u>Description</u>: Dumps specified storage and continues <u>Parameters Passed</u> PAR1 byte 0 X'00' 1-3 unused PAR3 bytes 0-3 either (a) zero or (b) address of a page heading to be printed <u>Parameters Returned</u>: None <u>Notes</u>: 1. The areas to be dumped, and the editing to be done on them, is given in the DUMP parameter to the compilation 2. The dump is only produced if the two character name in ZMYNAM matches one of the phase names specified in the DUMP parameter 3. The message "PHASE zz COMPLETED" is printed if 'P' is included in the DUMP parameter even though control is returned to the point of invocation in the phase 4. The registers printed by the dump routine are not necessarily as when control was passed from the calling phase
ZEND	6C	<u>Description</u>: Terminates compilation immediately <u>Parameters Passed</u>: None <u>Parameters Returned</u>: None

Name	Hex. Offset	Details
ZLOADW	24	Description: Loads phase and returns control to calling phase Parameters Passed PAR1 byte 0 unused 1-3 address of name of phase to be loaded Parameters Returned PAR1 byte 0 unused 1-3 address of phase load point Notes: 1. Control is returned to calling phase, not to phase just loaded 2. The entry point ZLOADW causes the phase loaded to be marked as such, and therefore cannot be loaded again (see ZLOADX)
ZLOADX	38	Description: Loads phase and control is returned to calling phase Parameters Passed PAR1 byte 0 unused 1-3 address of name of phase to be loaded Parameters Returned PAR1 byte 0 unused 1-3 address of phase load point Notes: 1. Control is returned to calling phase, not to phase loaded 2. The phase loaded may be loaded again, since it is not marked as loaded by this entry point (see ZLOADW)
ZNALAB	68	Description: Makes unaligned dictionary entry and returns absolute address Makes unaligned dictionary entry and returns absolute address Parameters Passed PAR1 byte 0 unused 1-3 address of entry PAR2 bytes 0-1 unused 2-3 length of entry (binary) Parameters Returned PAR1 byte 0 unused 1-3 dictionary entry address PAR4 bytes 0-1 unused 2-3 reference to some point in the same dictionary block Notes: 1. The entry is constructed exactly as it will appear in the dictionary, complete with code byte and length field 2. ZNALRF performs the same function and returns more information, with no loss in efficiency

Name	Hex. Offset	Details
ZNALRF	64	<u>Description</u>: Makes unaligned dictionary entry and returns its dictionary reference and absolute address <u>Parameters Passed</u> PAR1 byte 0 unused 1-3 address of entry PAR2 bytes 0-1 unused 2-3 length of entry (binary) <u>Parameters Returned</u> PAR1 bytes 0-1 unused 2-3 dictionary reference of entry PAR4 bytes 0 unused 1-3 dictionary entry address <u>Note</u>: See ZNALAB
ZTXTAB	54	<u>Description</u>: Converts text reference to absolute address <u>Parameters Passed</u> PAR1 byte 0 X'80' if status of block to remain unchanged not X'80' - text block set to busy 1-3 text reference to be converted <u>Parameters Returned</u> PAR1 byte 0 unused 1-3 absolute address corresponding to text reference
ZTXTRF	50	<u>Description</u>: Converts absolute address to text reference <u>Parameters Passed</u> PAR1 byte 0 unused 1 block number of text block containing absolute address 2-3 unused PAR2 byte 0 unused 1-3 address to be converted <u>Parameters Returned</u> PAR1 byte 0 unused 1-3 text reference corresponding to absolute address <u>Note</u>: This routine is of use in only a few cases since it requires to be passed the block number containing the absolute address, and only returns the offset from the start of the block. This offset can be calculated, if the text block is scanned sequentially, by subtracting the address of the start of the block
ZUERR	30	<u>Description</u>: Inserts diagnostic messages in dictionary <u>Parameters Passed</u> PAR5 bytes 0-1 unused 2-3 numeric parameter, if any (halfword binary) PAR6 byte 0 unused 1-2 message number (hexadecimal) 3 bit 0 on if text to be inserted 1 on if statement number to be inserted 2 on if numeric parameter to be inserted 3 on if dictionary reference to be inserted 4-7 severity code of message X'0' Termination X'4' Severe X'8' Error X'C' Warning

Name	Hex. Offset	Details
ZUERR	30	PAR7 either: byte 0 unused 1-3 address of text to be inserted or: byte 0-1 unused 2-3 dictionary reference to be inserted PAR8 bytes 0-1 unused 2-3 length of text to be inserted <u>Parameters Returned</u>: None <u>Notes</u> 1. When the message is printed the numeric parameter is converted to decimal and inserted into the message. PAR5 is unused if bit 2 of byte 3 in PAR6 is off 2. If bit 1 is on, the statement number is taken from the ZSTAT slot in the communications region at the time ZUERR is called 3. Bits 0 and 3 are mutually exclusive and if both are on in a call to ZUERR, the routine aborts 4. If bit 3 is on, the dictionary reference from PAR7 is inserted into the message dictionary entry. The error message phases pick up the BCD from the dictionary entry, indicated by the reference, and insert it into the message 5. If bit 0 is on, text is picked up from the address given in PAR7. This text is in compiler internal representation. The length of the text is taken from PAR8. The maximum length of text inserted into a message is 10 characters. If the length given is greater than 10 the text is truncated to 10 characters. If the text has been truncated it is enclosed in quotes in the message
ZUGC	10	<u>Description</u>: Gets scratch storage <u>Parameters Passed</u> PAR1 bytes 0-3 count of 512 byte blocks of storage required <u>Parameters Returned</u> PAR1 byte 0 unused 1-3 address of allocated scratch storage PAR2 bytes 0-1 unused 2-3 bytes of storage allocated
ZULF	70	<u>Description</u>: Writes record to object module file (PLILOAD) <u>Parameters Passed</u> PAR1 byte 0 unused 1-3 address of 72 byte area containing record to be written <u>Parameters Returned</u>: None <u>Notes</u>: 1. If an uncorrectable I/O error occurs on this data set the option LOAD is switched off, a message is produced, and the compilation continues without producing an object module 2. The last 8 bytes of the record will contain: bytes 73-76 name of object module, taken from ZPRNAM in the communications region. This is set from the first four characters of the first entry label of the external procedure 77-80 sequence number

Name	Hex. Offset	Details
ZUPL	08	Description: Puts record out to PLILIST Data Set Parameters Passed PAR1 byte 0 unused 1-3 address of area defining print line PAR3 bytes 0-3 zero if no secondary heading to be printed or address of area defining secondary heading line Parameters Returned: None Notes: 1. Paging action is performed automatically 2. Format of area containing print line is: byte 0 x'00' 1 total length of area (binary) 2 ASA control character 3 line to be printed-EBCDIC, variable length, maximum 132 characters
ZURC	18	Description: Releases scratch storage got by ZUGC Parameters Passed PAR1 bytes 0-3 count of entries to ZUGC to be released (binary) Parameters Returned: None Note: The routine frees all the storage that was allocated by the last n calls to ZUGC, where n is given by PAR1
ZURD	0C	Description: Reads record from PLIINPUT Parameters Passed PAR1 byte 0 unused 1-3 address of area into which record is to be placed Parameters Returned PAR2 bytes 0-1 unused 2-3 length of record returned (binary)
ZUTXTC	14	Description: Gets new text block and, optionally, chains it to a current block Parameters Passed PAR1 byte 0 unused 1-3 text reference to a current block if new one is to be chained PAR2 bytes 0-2 unused 3 X'00' no chaining or X'8n' chaining where n is status required for current block Parameters Returned PAR1 byte 0 unused 1-3 text reference to new block PAR2 byte 0 unused 1-3 absolute address of start of new block Notes: 1. The new block is set to a status of 'busy' 2. See ZALTER for definition of status of blocks

APPENDIX I: PLC COMMUNICATIONS REGION

This appendix contains:

- A layout of the PLC communications region (CHAPLI)
- The CHAMGL DSECT for the merge list
- The CHAPLI DSECT for the PLC communications region

	<------------------------------------4 bytes------------------------------------>			
CFBAA7	NOT USED			PLIFTP (Footprint)
	PLISOD (Diagnostics on SYSOUT)	PLILMN (LIMEN)	PLIBRV (BREVITY)	PLIERR (Error level code)
	PLIDDN (Pointer to SOURCEDS DCB)			
	PLILDN (PointerR to listing DCB)			
	PLIMAC (Pointer to MACRO DSNAME)			
	PLIMRG (Pointer to MRGLST block)			
	PLIPDS (Pointer to merge DSNAME)			
	PLIPRT (PRINT Option)	PLILDS (Listing Option)	PLICON (CONTINUE Option)	PLILOD (LOAD Option) (Not used)
	PLINAM (PLI Module Name) (8 bytes)			
	PLIEXP (Pointer to EXPLICIT parameter list)			
	PLIXDS (Pointer to XFERDS name)			

Figure 19. PLC Communications Region

DSECT for MERGE MODULE LIST (CHAMGL)

DSECT for PLC COMMUNICATION REGION (CHAPLI)

```
LOCATION  INSTRUCTION    ADDR 1    ADDR 2   STATEMNT  SOURCE
                                            0019200        COPY CHAPLI
          63 00000                                 +  CHAPLI    DSECT                  DSECT FOR COMMUNICATION BUCKET
   00000                                           +  PLIFTM    DS     F
          63 00003                                 +  PLIFTP    EQU    PLIFTM+3        FOOTPRINT OF PATH THROUGH PLC
          00000000                                 +  PLIFT0    EQU    X'00'                PLC NOT PREVIOUSLY INTERRUPTED
          00000004                                 +  PLIFT1    EQU    X'04'                EDITOR END REQUIRED
          00000008                                 +  PLIFT2    EQU    X'08'                DATA SET CLEANUP REQUIRED
          0000000C                                 +  PLIFT3    EQU    X'0C'                PL/1 COMPILER INVOKED
          00000010                                 +  PLIFT4    EQU    X'10'                DATA SET CLEANUP REQUIRED
          00000014                                 +  PLIFT5    EQU    X'14'                ODC END REQUIRED
          00000018                                 +  PLIFT6    EQU    X'18'                DATA SET CLEANUP REQUIRED
          0000001C                                 +  PLIFT7    EQU    X'1C'                CFBAK END RTN REQUIRED
          00000020                                 +  PLIFT8    EQU    X'20'                DATA SET CLEANUP REQUIRED
          00000024                                 +  PLIFT9    EQU    X'24'                PLC CALL COMPLETE
63 00004                                           +  PLISOD    DS     XL1             DIAGNOSTICS ON SYSOUT OPTION
          00000000                                 +  PLISD1    EQU    X'00'                DIAGNOSTICS ON SYSOUT
          00000001                                 +  PLISD2    EQU    X'01'                NO DIAGNOSTICS
63 00005                                           +  PLILMN    DS     CL1             VALUE OF "LIMEN"
          000000C9                                 +  PLILM1    EQU    C'I'                 INFORMATION MESSAGES
          000000E6                                 +  PLILM2    EQU    C'W'                 WARNING MESSAGES
          000000D5                                 +  PLILM3    EQU    C'N'                 ERROR MESSAGES
          000000E7                                 +  PLILM4    EQU    C'X'                 SERIOUS ERROR MESSAGES
          000000E3                                 +  PLILM5    EQU    C'T'                 TERMINAL ERROR MESSAGES
63 00006                                           +  PLIBRV    DS     CL1             VALUE OF "BREVITY"
          000000D4                                 +  PLIBR1    EQU    C'M'                 MESSAGE ID ONLY
          000000E2                                 +  PLIBR2    EQU    C'S'                 NORMAL MESSAGE TEXT
          000000C5                                 +  PLIBR3    EQU    C'E'                 EXTENDED MESSAGE TEXT
          000000E3                                 +  PLIBR4    EQU    C'T'                 STANDARD TEXT-NO MSG ID
          000000E7                                 +  PLIBR5    EQU    C'X'                 EXTENDED TEXT-NO MSG ID
63 00007                                           +  PLIERR    DS     XL1             ERROR LEVEL CODE
          00000000                                 +  PLIER0    EQU    X'00'                NO ERRORS DETECTED
          00000004                                 +  PLIER1    EQU    X'04'                TYPE 1 ERRORS
          00000008                                 +  PLIER2    EQU    X'08'                TYPE 1 ERRORS - ERRORS
          0000000C                                 +  PLIER3    EQU    X'0C'                TYPE 2 ERRORS - SEVERE
          00000010                                 +  PLIER4    EQU    X'10'                TYPE 3 ERRORS - TERMINAL
63 00008                                           +  PLIDDN    DS     F               POINTER TO SOURCE DCB
63 0000C                                           +  PLILDN    DS     F               POINTER TO LISTING DCB
63 00010                                           +  PLIMAC    DS     F               POINTER TO MACRO DATA SET NAME
63 00014                                           +  PLIMRG    DS     F               POINTER TO FIRST BLOCK OF MERGE LIST
63 00018                                           +  PLIPDS    DS     F               POINTER TO MERGE DATA SET NAME
63 0001C                                           +  PLIPRT    DS     XL1             PRINT OPTION
          00000000                                 +  PLIPR0    EQU    X'00'                NO PRINT
          00000041                                 +  PLIPR1    EQU    X'41'                PRINT - NO ERASE
          00000061                                 +  PLIPR2    EQU    X'61'                PRINT WITH ERASE
63 0001D                                           +  PLILDS    DS     XL1             LISTING DATA SET OPTION
          00000000                                 +  PLILS0    EQU    X'00'                LISTING DATA SET
          00000001                                 +  PLILS1    EQU    X'01'                LISTING ON SYSOUT
63 0001E                                           +  PLICON    DS     XL1             CONTINUATION OPTION
          00000000                                 +  PLICN1    EQU    X'00'                NO CONTINUATION
          00000003                                 +  PLICN2    EQU    X'C3'                CONTINUE COMPILATIONS
63 0001F                                           +  PLILOD    DS     XL1             LOAD OPTION
          00000000                                 +  PLILD1    EQU    X'00'                LOAD - CONVERSION REQUIRED
          00000001                                 +  PLILD2    EQU    X'01'                NO LOAD - COMPILE ONLY
63 00020                                           +  PLINAM    DS     CL8             NAME OF CURRENT OBJECT MODULE
63 00028                                           +  PLIEXP    DS     A               POINTER TO EXPLICIT PARAM LIST
63 0002C                                           +  PLIXDS    DS     A               POINTER TO XFERDS NAME
```

APPENDIX J: ODC -- INPUT RECORD FORMAT

This appendix contains:

- card image formats of records which are input to ODC
- a list of tables and DSECTs used by ODC

ESD Input Record (Card Image)

```
┌───┬─────┬─────────┬───────┬───────┬───────┬─────────────────────────────┬────────────────┐
│ 1 │ 2-4 │  5-10   │ 11,12 │ 13,14 │ 15,16 │            17-72            │     73-80      │
└─┬─┴──┬──┴───┬─────┴───┬───┴───┬───┴───┬───┴─┬───────────────────────────┴──┬─────────────┘
  │    │      │         │       │       │     │                              │
  │    │      │         │       │       │     │                              └───── Not used
  │    │      │         │       │       │     │
  │    │      │         │       │       │     └───── ESD Data -- see below
  │    │      │         │       │       │
  │    │      │         │       │       ├───── Blank if all ESD items are LD
  │    │      │         │       │       │
  │    │      │         │       │       └───── ESD IDENTIFIER of first ESD item (other
  │    │      │         │       │              than LD)
  │    │      │         │       │
  │    │      │         │       └───── Blank
  │    │      │         │
  │    │      │         └───── Number of bytes of ESD data
  │    │      │
  │    │      └───── Blank
  │    │
  │    └───── ESD
  │
  └───── 12-9-2 (0000 0010)
```

ESD Data Item

```
┌──────────────┬───┬────────┬────┬────────┐
│     1-8      │ 9 │ 10-12  │ 13 │ 14-16  │
└─┬────────────┴─┬─┴─┬──────┴──┬─┴─┬──────┘
  │              │   │         │   │
  │              │   │         │   │     Zero - if length is on END card.
  │              │   │         │   ├─────Length of control section (if type is: SD,PC,CM)
  │              │   │         │   ├─────Identifier of SD entry containing name
  │              │   │         │   ├─────Blank if type is ER
  │              │   │         │   └─────Length of pseudo-register (PR)
  │              │   │         │
  │              │   │         └─────Blank - Alignment Factor for type PR
  │              │   │
  │              │   └─────24 bit address (SD,PC,LD,LR)
  │              │
  │              └───── Type - Hex (00=SD,01=LD,02=ER,03=LR,04=PC,05=CM,06=PR)
  │
  ├───── Name -- when type is:  SD,LD,LR,ER,CM,PR
  └───── Blank -- when type is:  PC or blank CM
```

Text Input Record (Card Image)

```
┌─┬────┬─┬───┬────┬─────┬─────┬─────┬───────────┬──────────┐
│1│ 2-4│5│6-8│9-10│11,12│13,14│15,16│   17-72   │  73-80   │
└┬┴─┬──┴┬┴─┬─┴─┬──┴──┬──┴──┬──┴──┬──┴──┬────────┴──┬───────┘
 │  │   │  │   │     │     │     │     │           │
 │  │   │  │   │     │     │     │     │           └----- Not used
 │  │   │  │   │     │     │     │     │
 │  │   │  │   │     │     │     │     └----- Text data (machine
 │  │   │  │   │     │     │     │            language code)
 │  │   │  │   │     │     │     │
 │  │   │  │   │     │     │     └----- ESD Identifier of SD for control
 │  │   │  │   │     │     │            section of this text
 │  │   │  │   │     │     │
 │  │   │  │   │     │     └----- Blank
 │  │   │  │   │     │
 │  │   │  │   │     └----- Number of bytes of text data
 │  │   │  │   │
 │  │   │  │   └----- Blank
 │  │   │  │
 │  │   │  └----- 24 bit address of first byte of text data
 │  │   │
 │  │   └----- Blank
 │  │
 │  └----- TXT
 │
 └----- 12-9-2 (0000 0010)
```

RLD Input Record (Card Image)

```
┌─┬────┬─────┬─────┬─────┬──────────────────┬──────────┐
│1│ 2-4│ 5-10│11-12│13-16│      17-72       │  73-80   │
└┬┴─┬──┴─┬───┴──┬──┴──┬──┴─┬────────────────┴──┬───────┘
 │  │    │      │     │    │                   │
 │  │    │      │     │    │                   └----- Not used
 │  │    │      │     │    │
 │  │    │      │     │    └----- RLD data - see below
 │  │    │      │     │
 │  │    │      │     └----- Blank
 │  │    │      │
 │  │    │      └----- Number of bytes of RLD data
 │  │    │
 │  │    └----- Blank
 │  │
 │  └----- RLD
 │
 └----- 12-9-2 (0000 0010)
```

```
┌───┬───┬─┬─────┐
│1,2│3,4│5│6,7,8│ RLD data item
└┬──┴┬──┴┬┴┬────┘
 │   │   │ │
 │   │   │ └--Assigned address of address constant
 │   │   │
 │   │   └-Flag field -- (TTTTLLSTn)
 │   │          TTTT=type                           S=Direction of Relocation
 │   │          0000=non-branch                     0=positive (+)
 │   │          0001=branch                         1=negative (-)
 │   │          0011=pseudo register cumulative length Tn=type of next RLD item
 │   │            LL=length of address constant       0=next RLD item has a dif-
 │   │            01=2 bytes                            ferent R or P pointer;
 │   │            10=3 bytes                            they are present in the
 │   │            11=4 bytes                            next item.
 │   │                                                1=next RLD item has the
 │   │                                                  same R and P pointers,
 │   │                                                  hence they are omitted.
 │   │
 │   └---Position pointer (P) - ESDID of SD for control section that contains the
 │                              address constant
 │
 └-------Relocation pointer (R) - ESDID of CESD entry for the symbol being referred to.
                                  Zero (00) if type= PR cumulative length
```

END Input Record - Type 1 (Card Image)

```
┌───┬─────┬───┬─────┬───────────┬─────┬──────────────┬───────┬────────────────────────────┐
│ 1 │ 2-4 │ 5 │ 6-8 │   9-14    │15,16│    17-28     │ 29-32 │           33-80            │
└─┬─┴─┬───┴─┬─┴─┬───┴─┬─────────┴─┬───┴──┬───────────┴─┬─────┴─┬──────────────────────────┘
  │   │     │   │     │           │      │             │       │
  │   │     │   │     │           │      │             │       └--Not used
  │   │     │   │     │           │      │             │
  │   │     │   │     │           │      │             └--Control section length for con-
  │   │     │   │     │           │      │                trol section whose length was
  │   │     │   │     │           │      │                not specified in SD ESD item.
  │   │     │   │     │           │      │                byte 29 is binary zero if length
  │   │     │   │     │           │      │                is present.
  │   │     │   │     │           │      │
  │   │     │   │     │           │      └-Blank
  │   │     │   │     │           │
  │   │     │   │     │           └-ESDID of SD item for this control section that
  │   │     │   │     │             contains the address specified in bytes 6-8.
  │   │     │   │     │
  │   │     │   │     └-Blank
  │   │     │   │
  │   │     │   └-24 bit address of entry point (optional)
  │   │     │
  │   │     └-Blank
  │   │
  │   └-END
  │
  └-12-9-2 (0000 0010)
```

Load Data Set Error Record (Card Image)

```
┌──────┬─────────────┬────┬───────────────────────────────────────────────────────┐
│ 1-4  │    5-12     │ 13 │                        14-80                          │
└──┬───┴──────┬──────┴─┬──┴───────────────────────────────────────────────┬───────┘
   │          │        │                                                  │
   │          │        │                                                  └───── Not used
   │          │        │
   │          │        ├───── X'00' No errors
   │          │        │
   │          │        ├───── X'04' Warning messages
   │          │        │
   │          │        ├───── X'08' Error messages
   │          │        │
   │          │        ├───── X'0C' Severe error messages
   │          │        │
   │          │        └───── X'10' Terminal error messages
   │          │
   │          └───── Double precision fixed-point number resulting
   │                 from issuing a REDTIM macro instruction
   │
   └───── C' VID' Identifies load data set error record
```

TABLES AND DSECTS USED BY ODC

The ODC routine uses the following system DSECTS and associated tables. Of these, CHAPLI and CHAMGL are used exclusively by PL/I, and can be found in Appendix I of this manual. The rest may be found in IBM System/360 Time Sharing System, System Control Blocks Program Logic Manual, Form Y28-2011.

CHADCB - to open and close data sets

CHATDT - to find the job library and determine whether a data set exists

CHAPLI - for communication with PLC

CHAMGL - to access the input merge list

CHAPOD
CHAPOM
CHAPOE
CHADHD — to search the partitioned data set directory in order to build a merge list based on the members of the job library

APPENDIX K: COMPILER OUTPUT MODULES

The PL/I compiler is contained in six link-edited output modules.

This appendix provides:

- a table of the six compiler output modules, indicating their functional names and the name of the VCON CSECT in each that holds the addresses of the modules within that output module, and
- a table organized by output module name, relating logical phases, physical phases, and modules.

Compiler Output Modules

Output Module	Functional Name	VCON CSECT
CFBAC	Control Output Module	
CFBAD	Main Output Module	IEMAU
CFBAE	First Preprocessor Output Module	IEMAW
CFBAF	Second Preprocessor Output Module	IEMAX
CFBAG	Optimization Output Module	IEMAY
CFBAH	Interphase Dumping Output Module	IEMAZ

Compiler Phases and Modules

Output Module	Logical Phase	Physical Phase	Modules	Description
CFBAC			AA	Controls running of compiler
			AC	Writes records on intermediate file PLIMAC
			AE	End of read-in phase
			AF	Compiler default options
			AG	Closes PLIMAC for output, reopens for input
			AK	Closing phase of compiler
			AL	Controls dictionary compilation
			XZ	Builds conversational diagnostics
CFBAD			AB	Performs detailed initialization
			AM	Phase marking
	Read-In		CA	Read-in phase common routines
			CC	Read-in phase common routines
			CE	Keyword Tables

Output Module	Logical Phase	Physical Phase	Modules	Description
CFBAD (cont)	Read-In (cont)	CI	CG,CI	Read-in first pass
			CK	Keyword Tables
		CL	CL, CM	Read-in second pass
			CN	Keyword tables
		CO	CO, CP	Read-in third pass
			CR	Keyword tables
		CS	CS, CT	Read-in fourth pass
		CV	CV,CW	Read-in fifth pass
	Dictionary	ED	ED	Initialization, subroutine package for Declare Pass 2
		EG	EF, EG	Initialization
		EI	EH,EI,EJ	First pass over DECLARE statements
		EL	EK,EL,EM	Second pass over DECLARE statements
		EP	EP	Constructs dictionary entries for PROCEDURE, ENTRY and CALL statement
		EW	EV,EW	Constructs dictionary entries for LIKE attributes
		EY	EX,EY,EZ	Constructs dictionary entries for ALLOCATE and for explicitly qualified based variables.
		FA	FA,FB	Checks context of source text
		FE	FE,FF	Changes BCD to dictionary references
		FI	FI	Checks validity of dictionary references
		FK	FK	Rearranges attributes
		FO	FO,FP	Constructs dictionary entries for ON-conditions
		FQ	FQ	Checks validity of PICTURE chain for PL/I functions and the TRANSLATE and VERIFY functions
		FT	FT,FU	Dictionary house-keeping
		FV	FV,FW	Merges second file statements into text
		FX	FX,FY,FZ	Processes identifiers for cross reference and attribute listing
		F1	F1	Determines if syntax check should terminate compilation
	Pretranslator	GA	GA	Constructs DECLARE and OPEN control blocks
		GB	GB,GC	Modifies I/O statements

Output Module	Logical Phase	Physical Phase	Modules	Description
CFBAD (cont)	Pretranslator (cont)	GK	GK	Checks parameter matching
		GO	GO	Preprocessor for second check on parameters
		GP	GP,GQ,GR	Second check on parameters
		GU	GU,GV	Processes CHECK condition statements
		HF	HF,HG	Processes structure assignments
		HK	HK,HL	Processes array assignments
		HP	HP	Processes items defined using iSUBs
	Translator	IA	IA,IB,IC	Stacks operators and operands
		IG	IG	Processes array and structure arguments and built-in functions
		IK	IK	First part of preprocessor for generic functions
		IL	IL	Second part of preprocessor for generic functions
		IM	IM,IN, IP,IQ	Processes generic functions
		IT	IT	Processes function triples
		IX	IX	POINTER and AREA checking
		JD	JD	Evaluates constant expressions
	Aggregates	JI	JI,JJ.	Structure pre-preprocessor
		JI	JI,JK,JL	Structure preprocessor
		JK	JK,JL,JM	Structure processor
		JP	JP	Checks DEFINED chains
		JZ	JZ	Checks for abort condition
	Pseudo-Code	LB	LB,LC	Generates triples to initialize AUTOMATIC and CONTROLLED scalar variables
		LD	LD	Constructs dictionary entries for initialized STATIC scalar variables and arrays
		LG	LG,LH	Expands DO loops
		LR	LR	Fixed decimal expression optimization and initialization for phase LS
		LS	LS,LT,LU	Converts expression triples to pseudo-code
		LV	LV	Provides string handling facilities
		LW	LW	Initialization for phase LX

Output Module	Logical Phase	Physical Phase	Modules	Description
CFBAD (cont)	Pseudo-Code (cont)	LX	LX,LY	Converts string triples to pseudo-code
		MA	MA	Constructs pseudo-code for functions
		MB	MB,MC	Constructs pseudo-code for pseudo-variables
		MD	MD	Scans for ADDR and STRING functions and generates code for each
		ME	ME	Constructs pseudo-code for in-line functions
		MG	MG,MH	Constructs pseudo-code for in-line functions
		MI	MI,MJ	Constructs pseudo-code for in-line functions
		MK	MK	Constructs pseudo-code for in-line functions
		ML	ML	Processes generic entry names
		MM	MM,MN,MO	Processes CALL and function procedure invocations
		MP	MP	Reorders BUY and SELL statements
		MS	MS,MT	Constructs pseudo-code for subscripts
		NA	NA	Generates pseudo-code for branches RETURN triples, etc.
		NG	NG	Generates library calling sequences for DELAY and DISPLAY statements
		NJ	NJ,NK	Generates library calling sequences for executable RECORD-oriented input/output statements
		NM	NM,NN	Generates library calling sequences for executable STREAM-oriented input/output statements
		NT	NT	Preprocessor for NU
		NU	NU,NV	Generates library calling sequences for data/format lists
		OB	OB,OC	Processes compiler functions and pseudo-variables
		OD	OD	Pseudo-code assignment
		OE	OD,OE,OF	Constructs pseudo-code for assignments
		OG	OG,OH	Generates library calling sequences
		OM	OM,ON,OO	Generates pseudo-code for data type conversions in-line

Output Module	Logical Phase	Physical Phase	Modules	Description
CFBAD (cont)	Pseudo-code (cont)	OP	OP,OQ	Generates pseudo-code for further in-line conversions
		OS	OS,OT,OU	Converts constants to required internal form
	Storage Allocation	PA	PA	Puts eligible DSA's into STATIC
		PD	PD	First STATIC storage allocation phase
		PH	PH	Second STATIC storage allocation phase
		PL	PL,PM	Constructs symbol tables and DEDs
		PP	PP,PO	Sorts AUTOMATIC chain
		PT	PT,PU,PV	Allocates AUTOMATIC storage
		QF	QF,QG,QH	Constructs prologues
		QJ	QJ,QK,QL	Allocates DYNAMIC storage
		QU	QU	Aligns misaligned operands and processes halfword binary operands
		QX	QX	Lists lengths of aggregates
	Register Allocation	RA	RA,RB,RC	Processes addressing mechanisms
		RD	RD	Flags branches for optimization
		RF	RF,RG,RH	Allocates physical registers
	Final Assembly	TF		Assembly first pass
		TJ	TJ,TK	Optimization
		TO	TO,TP,TQ	Produces ESD cards
		TT	TT,TU	Assembly second pass
		UA	UA,UB,UC	Final assembly initial values, first pass
		UD	UD,UB,UC	Generates RLD and TXT cards to set up dope vectors for STATIC DSAs
		UE	UE,UB,UC	Final assembly initial values, second pass
		UF	UF,UG,UH	Produces listings
		UI	UI,UG,UH	Completes final assembly listings
	Error Editor	XA	XA	Determines whether there are diagnostic messages to be printed.
			XA,XB	Dummy module

Output Module	Logical Phase	Physical Phase	Modules	Description
			XA,XC	Controls the printing of messages
			XF	Message address blocks
			XG,YY	Contain the diagnostic messages
CFBAE			BX	48-character set preprocessor
	Compile-Time Processor		AS	Resident phase for compile-time processor
			AV	Initialization phase for compile-time processor
		BM	BM,BN	Error message printout phase
			BO,BP,BV	Diagnostic messages
		BW		Clean-up phase for compile-time processor
CFBAF		BC	BC,BE,BF	Initial scan and translation phase for compile-time processor
		BG	BG,BI,BJ	Final scan and replacement phase for compile-time processor
CFBAG	Optimization	KA	KA,KB	Table handling and initialization
		KC	KC,KC1	DO temporaries
		KE	KE,KE1	DO MAP build
		KG	KG,KG1	DO examine
		KG	KJ	SUBS TABLE build
		KN	KN	Initialization
		KO	KO,KP,KQ	Subscript optimization 1 and 2
		KT	KT	SCAN utility
		KU	KU,KV	Merge patches and loop control
CFBAH			AD	Performs interphase dumping as specified in the DUMP option
			AH	Format annotated dictionary dump
			AI,AJ	Format annotated text dump
			AT	Tracing routine

INDEX

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
(USA Only)

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

IBM Technical Newsletter

File No.	S360-29
Re: Order No.	GY28-2051-0
This Newsletter No.	GN28-3161
Date:	September 15, 1970
Previous Newsletter Nos.	None

IBM System/360 Time Sharing System
PL/I Compiler Program Logic Manual

This Technical Newsletter, a part of Version 8, Modification 0, of IBM System/360 Time Sharing System, provides replacement pages for the subject publication. Pages to be inserted and/or removed are as follows:

11-14
29-30.1
51,52
105,106
221,222
255,256
421,422
425-428
435-442
451,452
465-474

A change to the text is indicated by a vertical line to the left of the change. A changed illustration is indicated by the symbol (•) to the left of the caption.

Summary of Amendments

The major change reflected in this TNL is the addition of module F1 for handling the syntax check option.

Other changes include the addition of several new messages, as well as minor technical corrections.

Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Dept. 643, Neighborhood Road, Kingston, N. Y. 12401

PRINTED IN U S A

IBM Technical Newsletter

File No. S360-29
Re: GY28-2051-0
This Newsletter No. GN28-3191
Date: September 30, 1971
Previous Newsletter Nos. GN28-3161

IBM SYSTEM/360 TIME SHARING SYSTEM
PL/I COMPILER PROGRAM LOGIC MANUAL

This Technical Newsletter, a part of Version 8, Modification 1, of IBM System/360 Time Sharing System, provides replacement pages for the subject publication. Pages to be inserted and/or removed are as follows:

iii-iv	225-236	
ix-2	237-238	(remove)
5-6	363-366	
11-16.1	435-442	
219-222	463-466	

A change is indicated by a vertical line to the left of the change.

Summary of Amendments

1. Module AT has been added to provide a debugging facility known as TRACE.

2. The PRV data set has been eliminated. ODC no longer resolves nonstandard QCONs, since the dynamic loader recognizes QCONs.

3. Module CFBAK has been added to give the user the option of transforming implicit calls to explicit calls.

4. PLC recognizes two new operands to the PLI command, EXPLICIT and XFERDS, which indicate the need for a call to CFBAK. PLC also recognizes a new PLCOPT option, NOCONV, which indicates that only CFBAK, and not ODC or the compiler, should be invoked.

5. If the default value PLIPACK is set to Y, ODC packs all CSECTs of the object module on external storage. If PLIPACK is set to P, noncommon static external CSECTs smaller than 4096 bytes, text CSECTs, and static internal CSECTs are packed. If PLIPACK is any other value or no value, no CSECTs are packed.

IBM Corporation, Dept. 643, Neighborhood Road, Kingston, N. Y. 12401

PRINTED IN U.S.A.

6. Other changes include the addition and deletion of messages, as well as minor technical corrections.

Please file this cover letter at the back of the manual to provide a record of changes.