IBM Technical Newsletter

File Number S360-29
Re: Form No. Y28-6800-3
This Newsletter No. Y33-6002
Date May 1, 1968
Previous Newsletter Nos. None

IBM SYSTEM/360 OPERATING SYSTEM
PL/I (F) COMPILER
PROGRAM LOGIC MANUAL

This Technical Newsletter provides replacement pages for IBM System/360 Operating System, PL/I (F) Compiler, Program Logic Manual, Form Y28-6800-3. Pages to be inserted and removed are listed below.

Pages to be Inserted	Pages to be Removed
25,26	25,26
33-36	33-36
43-46,46.1	43-46
49-52	49-52
59-61,61.1,62	59-62
117,118	117,118
129,130	129,130
135,136	135,136
139,140	139,140
209,210	209,210
243,244	243,244
255,256	255,256
287-290	287-290
302.1,302.2	-
303,304	303,304
335-338	335-338
345,345.1,346	345,346
355,356,356.1	355,356
363-365,365.1,366	363-366
373,374	373,374
413,414	413,414
421,422,422.1	421,422
425-428,428.1	425-428

A change to the text or a small change to an illustration is indicated by a vertical line to the left of the change; a changed or added illustration is denoted by the symbol • to the left of the caption.

IBM United Kingdom Laboratories Ltd., Programming Publications, Hursley Park, Winchester, Hampshire, England.

Restricted Distribution

The specifications contained in this Technical Newsletter correspond to Release 16 of IBM System/360 Operating System. Significant changes or additions will be reported in subsequent revisions or technical newsletters.

Summary of Amendments

This Technical Newsletter documents incremental improvements to the PL/I F Compiler for Release 16 of IBM System/360 Operating System. These improvements include: implementation of the UNALIGNED attribute and the STRING function; array and subscript optimization; and diagnostic message improvements.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

Module AD

Module AD performs inter-phase 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 phase AD will load either phase AH, or phases AI and AJ, or all three, to produce the required output.

The DUMP Option

The DUMP option which is specified in the PARM field of the EXEC card indicates where dumping of main storage is to take place. It may be specified in one of the following 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.

AREA is any combination of TDPSCIE:

- T text blocks
- D dictionary blocks
- P phases loaded
- S scratch storage
- C control phase
- I annotated dictionary blocks
- E annotated text blocks

The general syntax is:

```
DUMP[=([AREA],{x|(y,z)},...)]
```

A single phase name indicates dumping of storage after this single phase. A pair of phase names indicates a continuous group of phases after which dumping of storage is to occur.

The dump will appear on SYSPRINT, inserted into the normal compiler output.

If AREA is omitted the default taken is DTSP. If a program check occurs and DUMP has been specified then AREA will be given the default DTSPC.

Use of the DUMP option may cause the compiler to use about 8K bytes more core than the SIZE option specifies. This is because SIZE specifies the amount of core the compiler can use for normal compilation and does not allow for the internal compiler diagnostic dumps.

Example of an EXEC card using the DUMP option:

```
//STEP1 EXEC PROC=PL1LFC,
            PARM.PL1L='DUMP=(TE,QJ)'
```

This statement specifies compilation using the DUMP option to obtain a printout of the text blocks, the annotated text blocks, and of storage after the completion of compiler phase QJ.

Module AE

Module AE is the finalization of the READ-IN Phase control. (See Fig.4, Note[1])

Module AF

Module AF is a control section consisting of a table containing the compiler options which may be used during a compilation. The table is constructed at system generation time. The control section is brought into storage by the initialization Module AB at compilation time. A description of the use of Module AF is given in Appendix G.

Module AG

Module AG closes SYSUT3 for output, and re-opens it for input.

The closing and opening operations are performed in the following order:

CLOSE
  alter macro-type in data control block (DCB)

OPEN(INPUT)
  switch routine ZURD to point at SYSUT3 DCB

Module AH

This module produces a dump of the dictionary. It prints out the communications region in the first block, and the offsets tables for each block if the extended dictionary option is in use. The remainder of each block is printed out entry by entry. The BCD is translated for those entries containing BCD. At the end of the dump, a list of all the

dictionary codes used is given, with an explanation for each code.

The module is called by phase AD only if an I is specified in the AREA field of the DUMP option.

Modules AI and AJ

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 print in which the triples and pseudo-code items comprising the text are printed separately. This option is available between phases IA and OE inclusive.

Module AK

Module AK is the closing routine of the compiler. Its function is to release core used for dictionary, text blocks, scratch storage, and completed phases. If batch compilation is not specified, module AK closes all the files used by the compiler. If a batch compilation is specified, a check is made to determine whether any source programs are still to be compiled. Where there are none module AK closes all files. Where one or more programs remain to be compiled, the spill file only is closed, the batch delimiter card is scanned for syntax errors, and control is returned to module AA.

Module AL

This module contains the control routines for dictionary and text-block handling for the extended dictionary.

Module AM

Module AM marks phases as either wanted or not wanted, depending upon the compiler invocation options. Phases that are always loaded are marked wanted.

AM is the first compiler phase loaded after compiler initialization. It tests the relevant bits in CCCODE and marks the phases accordingly.

Module AN

This module contains the routines for dictionary and text-block handling for the normal-sized dictionary.

Module JZ

Module JZ builds the second half phase directory. A build list is constructed from the second half list held in Module AA; a BLDL is performed on this list. The phase directory is then reconstructed in Module AA for the second half of the compiler.

48-CHARACTER SET PREPROCESSCR

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

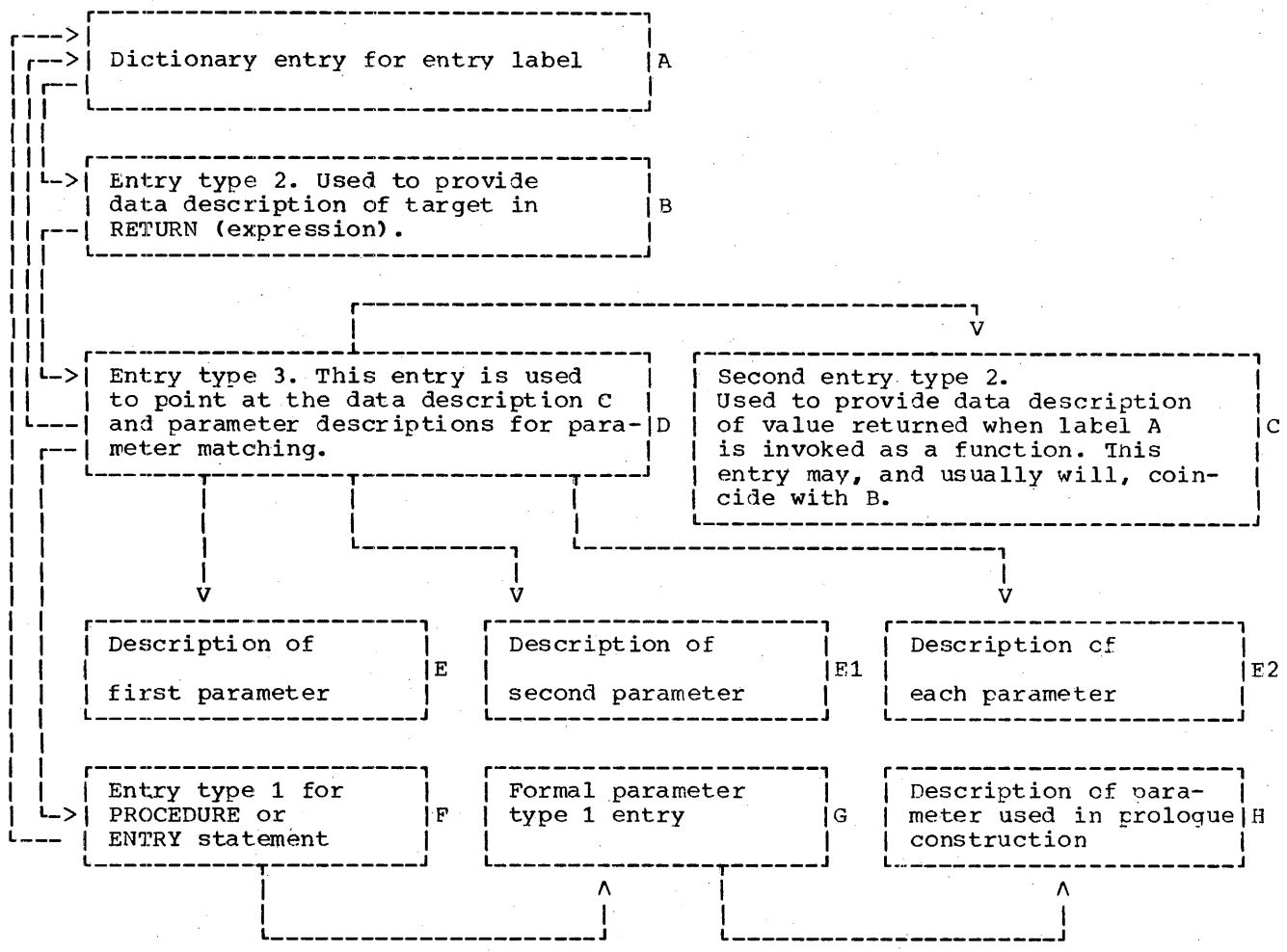
The preprocessor scans the input text for occurrences of characters peculiar to the 48-character set, and converts these to the corresponding 60-character symbols. It then puts out the adjusted text onto backing storage ready for Phase CI, the first pass of the Read-In Phase.

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.

Before the text is processed a copy of the original input is preserved. The output from the preprocessor is the transformed text, record by record, followed by the original text. The Read-In Phase processes transformed text but prints out the



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

Figure 6. Dictionary Entries for an Internal Entry Point

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 core, 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 Appendix C.2), 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.

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 Appendix C.2.

The attribute list is scanned and an options code byte is created in the entry type 1 (see Appendix C.2). A check is then made for invalid and inconsistent attributes. CHARACTER and BIT attributes are processed, and second file statements (see Appendix D.8) are created if necessary. Precision data are converted to binary, and dictionary entries are created for pictures (see Appendix C.7).

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 Appendix C.3).

For ENTRY statements, entry type 1 dictionary entries are created (see Appendix C.2), 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 Appendix C.8) 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 Appendix D.8) 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 Appendix C.8). Second file statements (see Appendix D.8) are created for adjustable length constants, and the source text is replaced by pointers to the statements.

DEFINED: second file statements (see Appendix D.8) 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 Appendix C.7) 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 Appendix C.4)
2. Structures (in this case, the 'true' level number is calculated) (see Appendix C.4)
3. Label Variables (see Appendix C.4)
4. Files (see Appendix C.7)
5. Entry Points (see Appendix C.2)
6. Parameters (see Appendix C.7)
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 core 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 Appendix C.2). 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 Appendix C.2), 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 Appendix C.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 Appendix C.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

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 core 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.

THE TRANSLATOR LOGICAL PHASE

The Translator Phase ccnsists 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 Appendix D.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 ECB 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 Appendix D.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 IL

This phase immediately precedes the main generic phase. Its function is to obtain a block of scratch storage and place the entire built-in function table in that area. The starting address of this table is then placed in a register, and control is released to the main generic processor.

Phase IM

Phase IM 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.

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.

THE 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 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.

THE 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

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, Phase LA, 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 core 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 preprocessor for LX and LY, moving constants into scratch core 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 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 LA 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 core, 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 tc 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 invcked 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 Appendix D.10).

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 one subscript which is an integer constant, and all following subscripts declared to have fixed upper and lower bounds. 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.

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

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 Appendix C.7). A code byte 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, 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 pseudo-code items) has the same form as the code produced by the Register Allocation Phase (see Appendix D.7).

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

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.

Phase RA

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 RF

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 insured 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.

THE 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

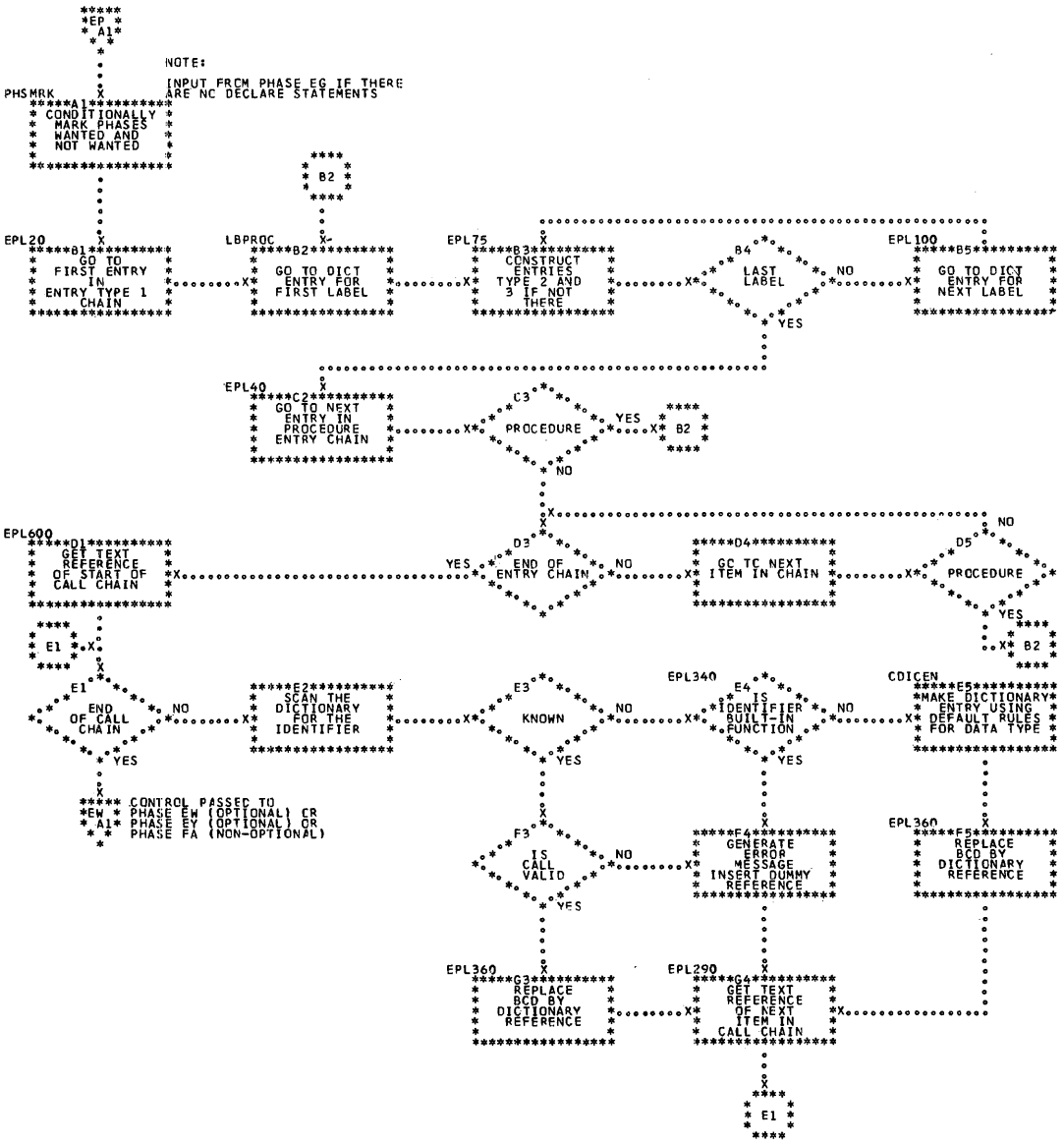
Phase TJ scans the text until no further optimization can be achieved in the final assembly.

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.

Chart EP. Phase EP Overall Logic Diagram



- Chart EW. Phase EW Overall Logic Diagram

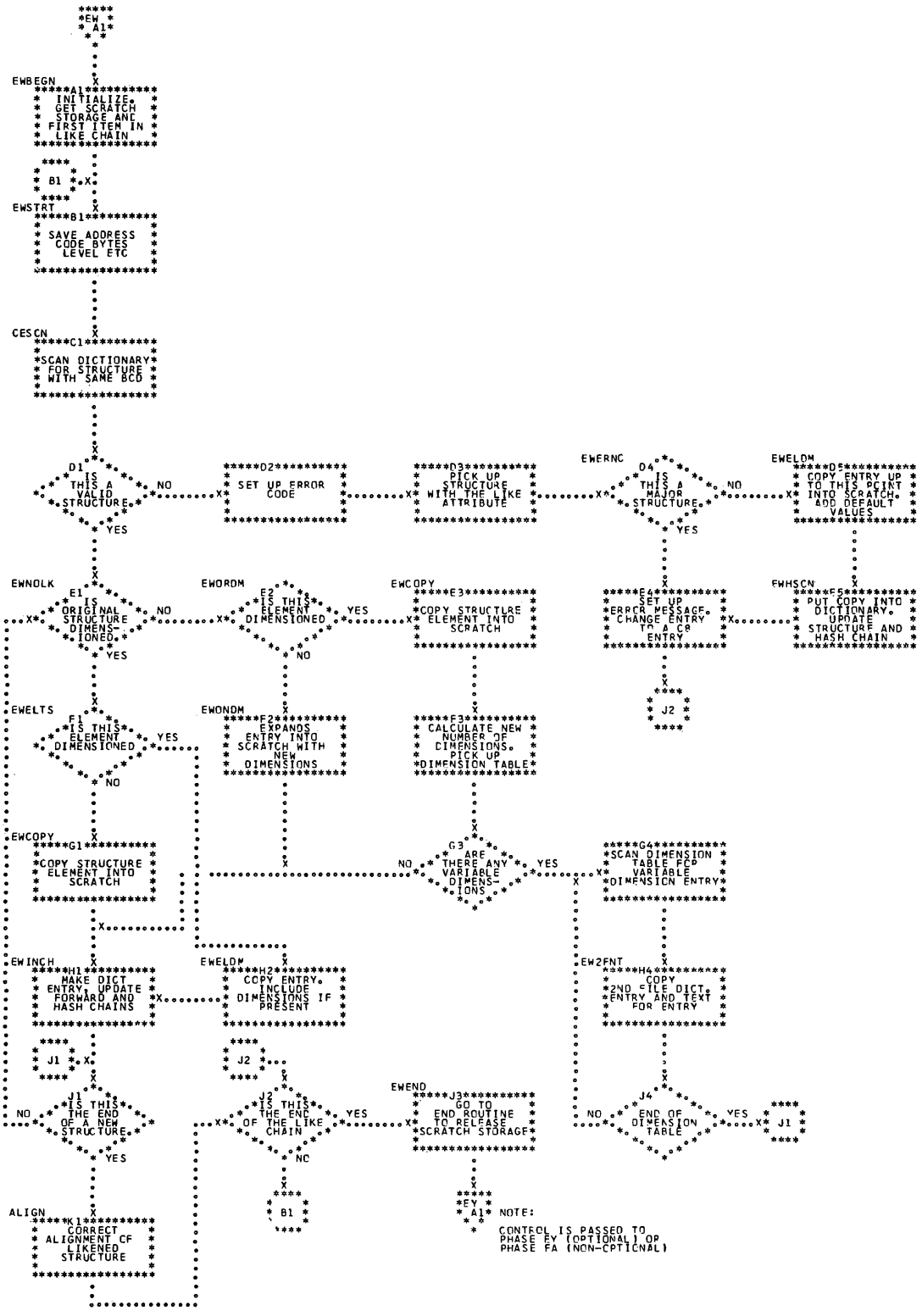


Table ED. Phase ED, Initialization

Statement or Operation Type	Main Processing Routine	Subroutines Used
Sets up routines in scratch core 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.
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 EL1. Phase EL Routine/Subroutine Directory

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.
CDAT56 (EK)	Processes USES attribute.
CDAT57 (EK)	Processes SETS 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 (cont'd)

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	Post-processor.
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 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.

• Chart MD. Phase MD Overall Logic Diagram

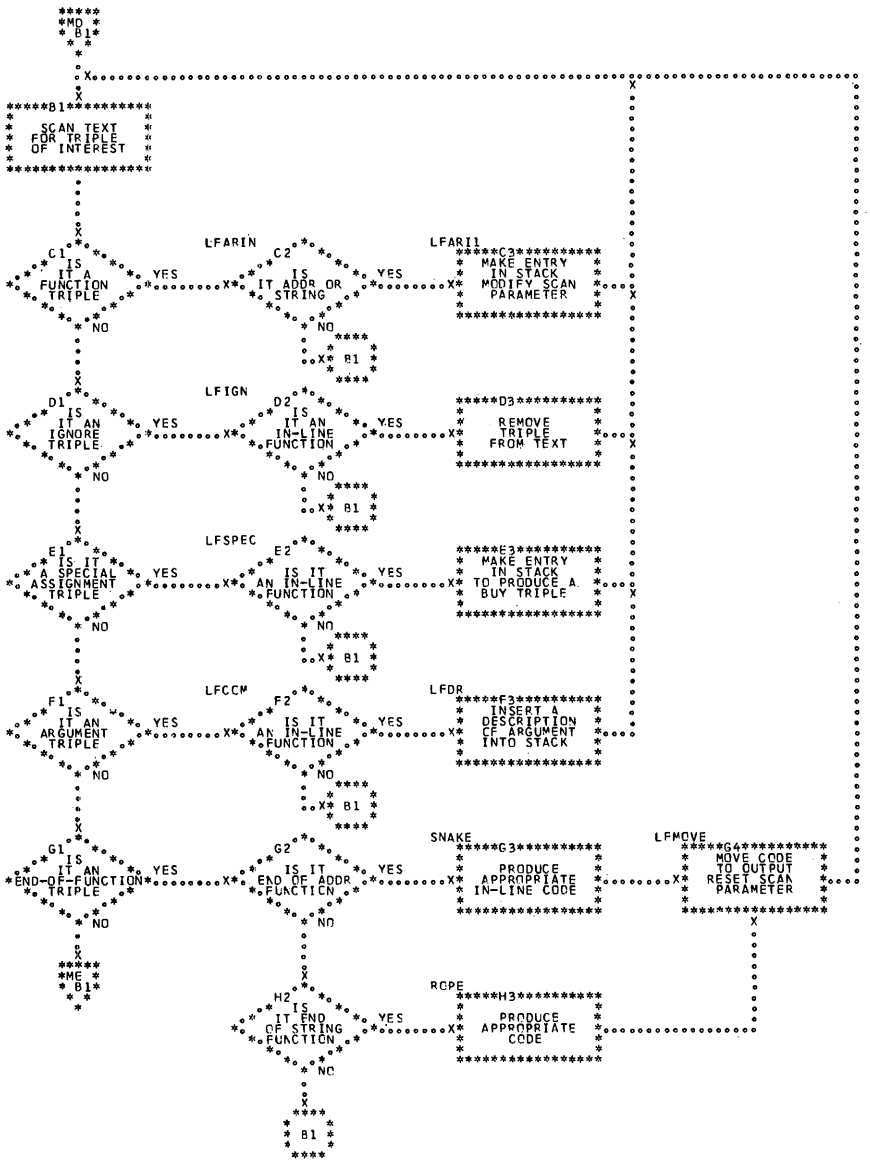


Chart ME. Phase ME Overall Logic Diagram

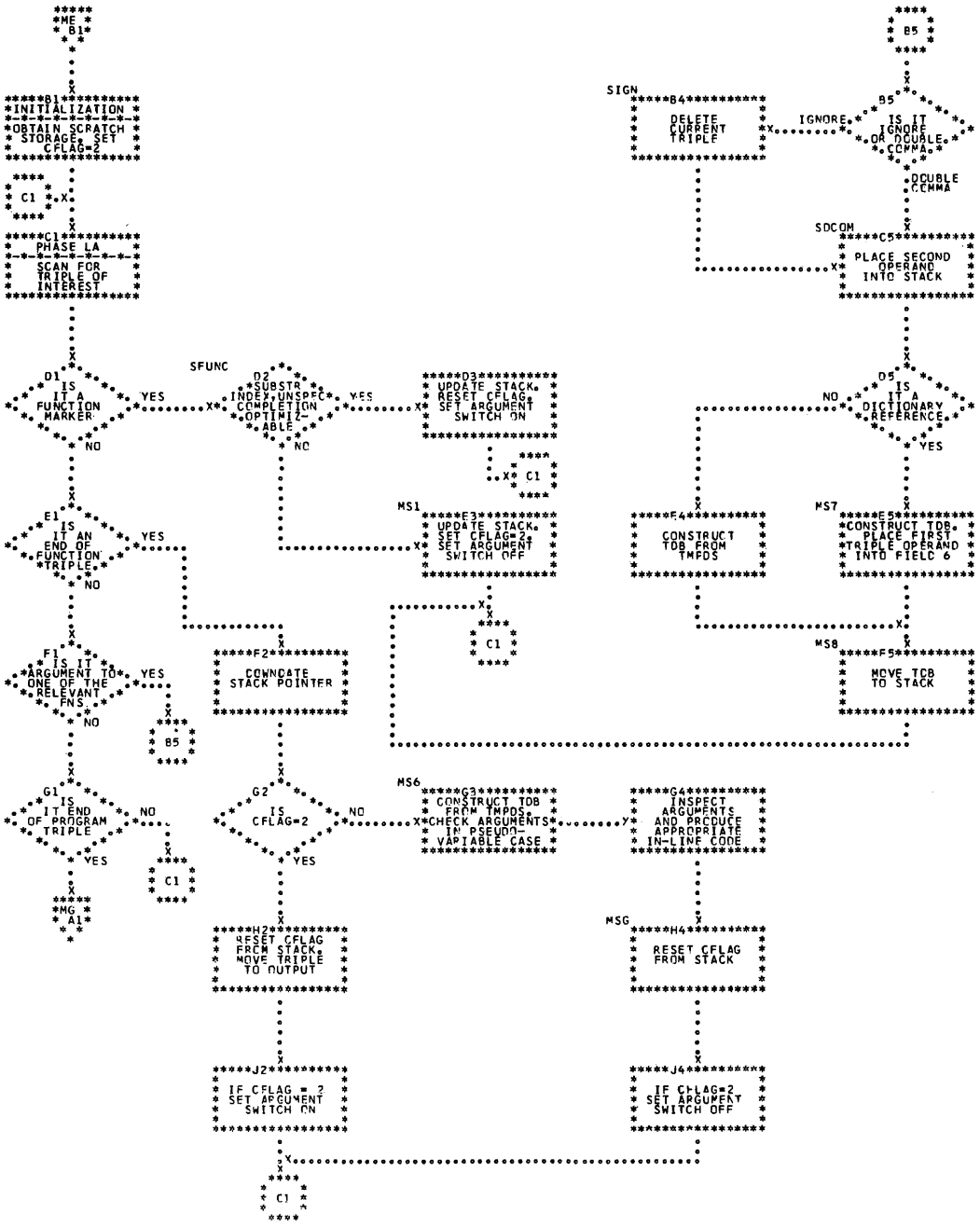


Table MB1. Phase MB Routine/Subroutine Directory

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 rescan 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-varaible 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 LA (SCAN)	None
Builds up function stack	LFARIN	None
Builds up argument stack	LFCOM	None
Moves generated code to output block	LFMOVE	MV3(LA)
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 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
Checks subscript range	SBSBRN	None

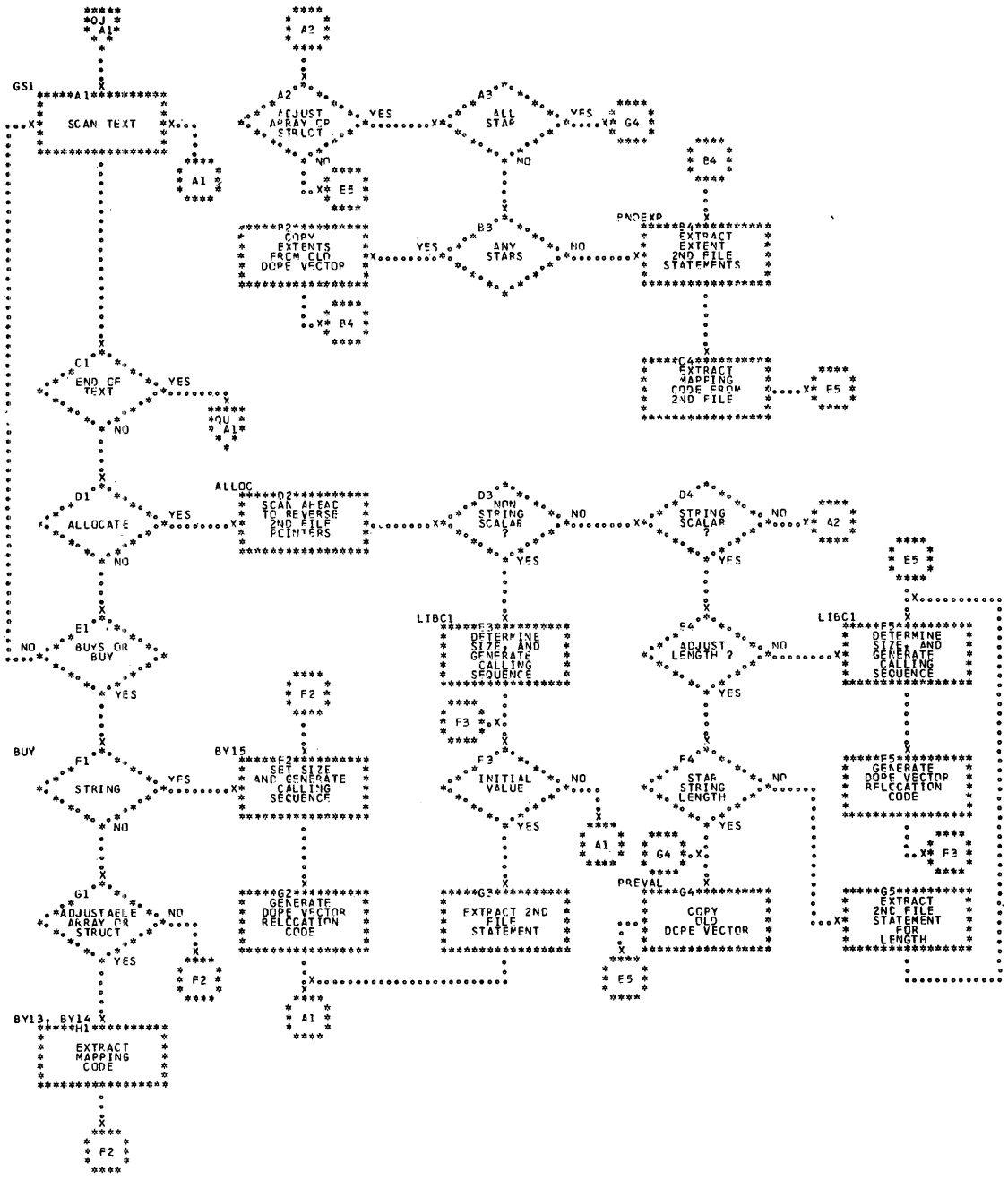
- Table MS1. Phase MS Routine/Subroutine Directory

Routine/Subroutine	Function
SBASS	Updates scan pointer over an assignment.
SBCOBI (MT)	Converts subscript to binary integer.
SBERR (MT)	Puts error message into dictionary.
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.
SBOPT	Calculates element offset in optimizable cases.
SBSBRN (MT)	Checks subscript range.
SBSCAN	Branches to LA 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.
SBTRID	Scans for comma, subscript prime, or subscript triple.
SBXOP (MT)	Handles special index feature.
SCAN	Controlling scan of text.
UTTEMP (MT)	Allocates workspace.

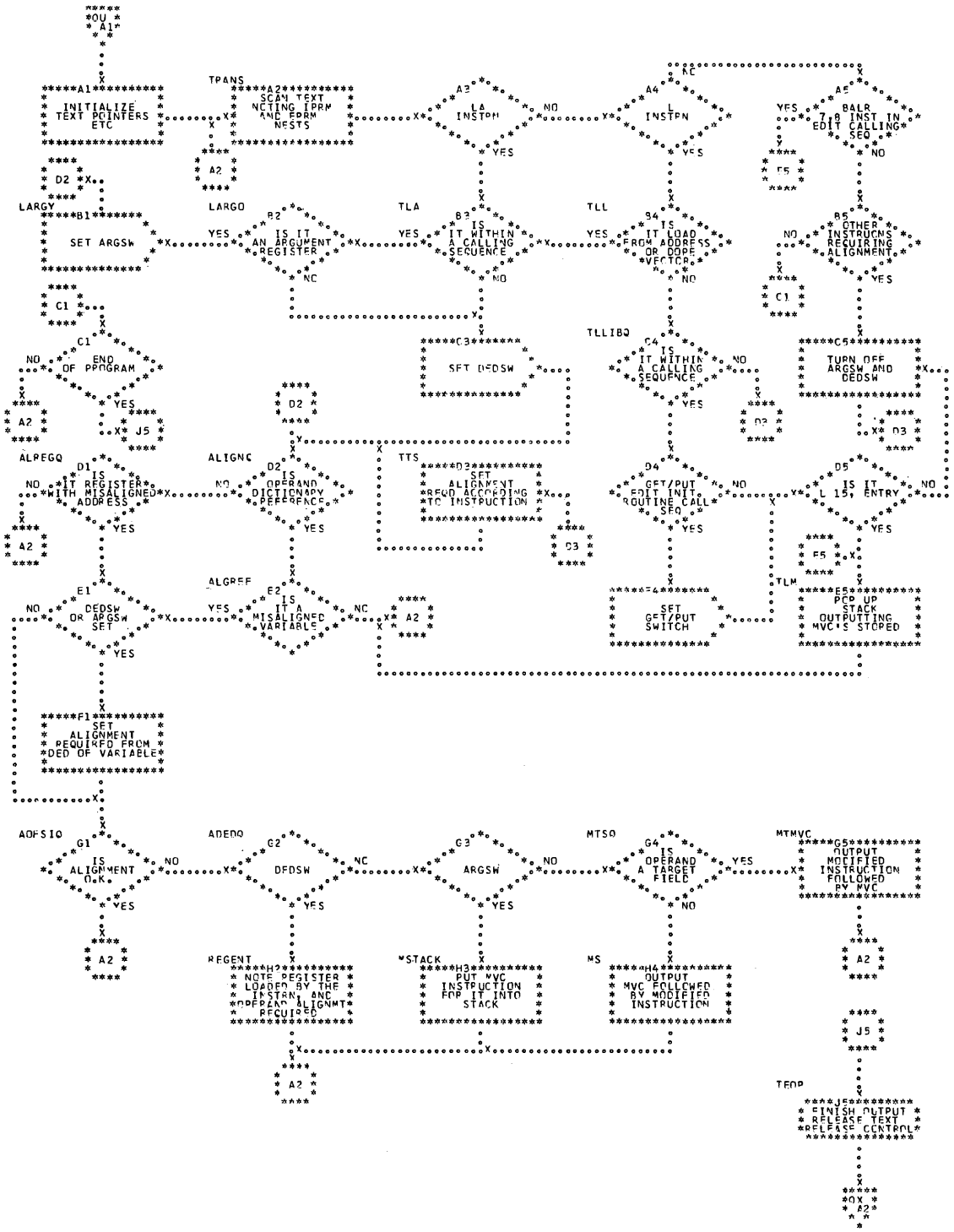
Table NA. Phase NA Pseudo-Code Branches, ON, Returns

Statement or Operation Type	Main Processing Routine	Subroutines Used
Initializes text block	NAINIT	SCINIT (LA)
Scans text for next triple of interest to user	NASC1, NASC2, NASC3	SC1, SC2, SC3 (all in LA)
Processes STOP statements	STOP	NAUT1
Processes EXIT statements	EXIT	NAUT1
Processes IF statements	IF	NAUTD, NAUT16, NAUT21, ZSTUT1
Processes ON statements	ON	NAUTD, NAUT6, NAUT16, SC5 (LA)
Produces Library call at end of each PROCEDURE or BEGIN block in source text	PROCP, BEGINP	NAUT1
Processes RETURN statements	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 GO TO statements	GOTO	NAUTD
Processes GOLN triples	GOLN	NAUTD
Processes GOOB statements	GOOB	NAUT5, NAUTD, NAUT16, SC5 (LA)
Processes SIGNAL statements	SIGNAL	NAUTD, NAUT6, NAUT16, NAUT8, NAUT10, NAUT21
Processes REVERT statements	REVERT	NAUTD, SC5 (LA)

• Chart QJ. Phase QJ Overall Logic Diagram



• Chart QU. Phase QU Overall Logic Diagram



•Chart QX. Phase QX Overall Logic Diagram

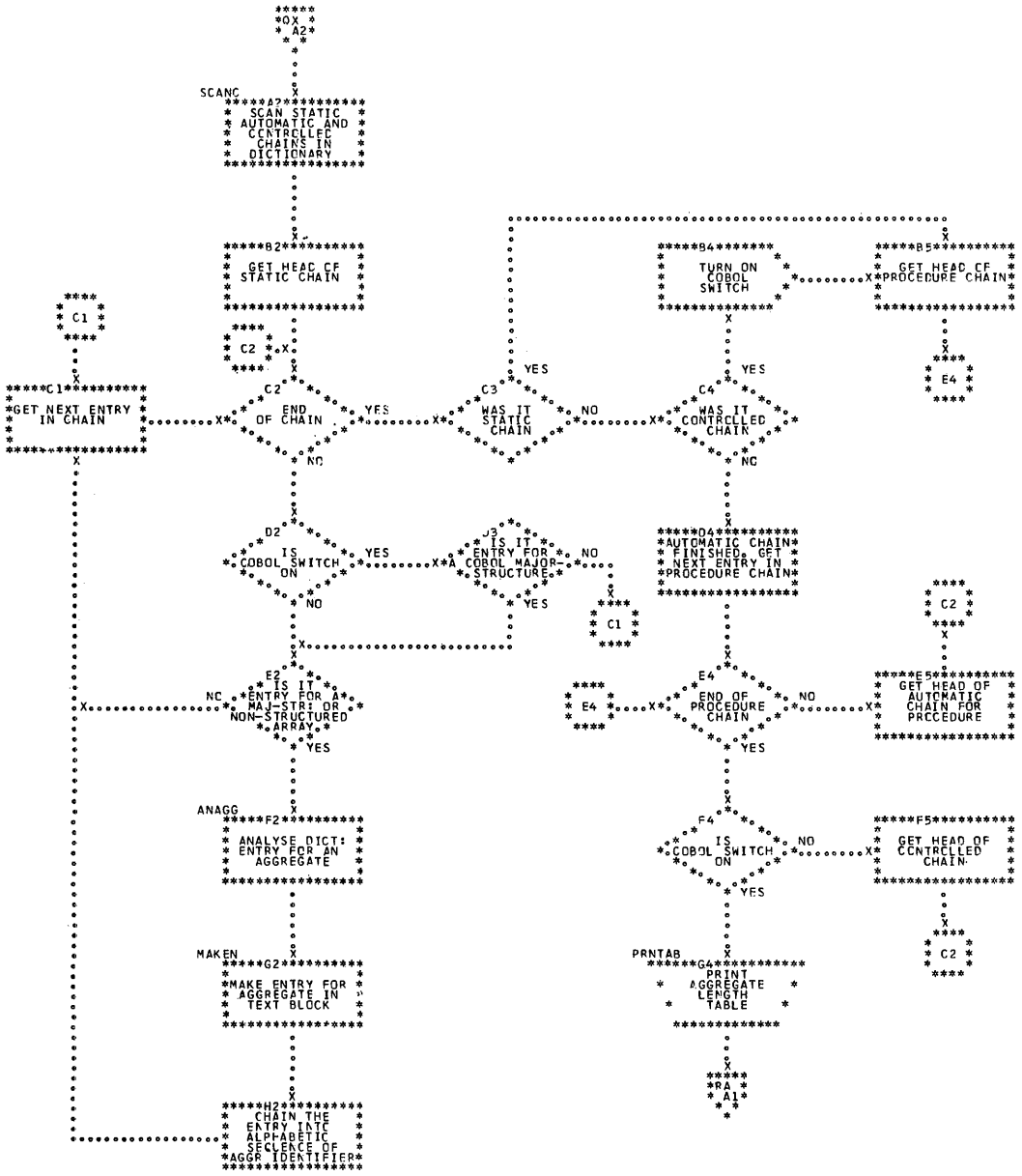


Table PA. Phase PA DSAs in STATIC Storage

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 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 terminal 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.

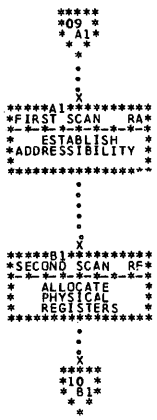
• Table QX. Phase QX Print Aggregate Length Table

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.

Chart 09. Register Allocation Logical Phase Flowchart



This appendix relates the logical phases, physical phases, and modules contained within the physical phases. The compiler name is IEMAA.

PHYSICAL PHASE	MODULES	DESCRIPTION
Compiler Control		
	AA	Controls running of compiler
	AB	Performs detailed initialization
	AC	Writes records on intermediate file SYSUT3
	AD	Performs interphase dumping as specified in the DUMP option
	AE	End of read-in phase
	AF	Controls system generation compiler options
	AG	Closes SYSUT3 for output, reopens for input
	AH	Format annotated dictionary dump
	AI,AJ	Format annotated text dump
	AK	Closing phase of compiler
	AL	Controls extended dictionary compilation
	AM	Phase marking
	AN	Controls normal dictionary compilation
	BX	48-character set preprocessor
	JZ	Builds second half phase directory
Compile-time Processor Logical Phase		
	AS	Resident phase for compile-time processor
	AV	Initialization phase for compile-time processor
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
BM	BM,BN	Error message printout phase
	BO,BV	Contain the diagnostic messages
BW		Cleanup phase for compile-time processor
Read-In Logical Phase		
	CA	Read-In phase common routines
	CC	Read-In phase common routines
	CE	Keyword tables
CI	CG,CI	Read-In pass 1
	CK	Keyword tables
CL	CL,CM	Read-In pass 2
	CN	Keyword tables
CO	CO,CP	Read-In pass 3
	CR	Keyword tables
CS	CS,CT	Read-In pass 4
CV	CV,CW	Read-In pass 5
Dictionary Logical Phase		
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 statements

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
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

Pretranslator Logical Phase

GA	GA	Constructs DECLARE and OPEN control blocks
GB	GB,GC	Modifies I/O statements
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 Logical Phase

IA	IA,IB,IC	Stacks operators and operands
IG	IG	Processes array and structure arguments and built-in functions
IL	IL	Preprocessor for generic functions
IM	IM,IN, IP,IQ	Processes generic functIc functions
IT		Processes function triples
IX	IX	POINTER and AREA checking
JD	JD	Evaluates constant expressicns

Aggregates Logical Phase

JI	JI,JJ	Struccure pre-preprocessor
JI	JI,JK,JL	Structure preprocessor
JK	JK,JL,JM	Structure processor
JP	JP	Checks DEFINED chains

Pseudo-Code Logical Phase

LA	LA	Utility scanning phase
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	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
LX	LX,LY	Converts string triples to pseudc-code
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	Pre-processor for NU
NU	NU,NV	Generates Library calling sequences for data/format lists
OB	OB,OC	Processes compiler functions and pseudo-variables
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
OP	OP,OQ	Generates pseudo-code for further in-line conversions
OS	OS,OT,OU	Converts constants to required internal form

Storage Allocation Logical Phase

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	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
QX	QX	Lists lengths of aggregates

Register Allocation Logical Phase

RA	RA,RB,RC	Processes addressing mechanisms
RF	RF,RG,RH	Allocates physical registers

Final Assembly Logical Phase

TF	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		Constructs the third phase list
XA,XC		Controls the printing of messages
XF		Message address blocks
XG,YY		Contain the diagnostic messages

This appendix describes the formats of dictionary entries during the compilation of a source program. The appendix is organized in the following 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. Uses of the OFFSET 1 and OFFSET 2 slots in DATA, LABEL, and STRUCTURE dictionary entries
7. Dictionary entries for:

   label constants
   data constants
   formal parameters
   FILE entries
   TASK and EVENT data
   internal library functions
   parameter descriptions
   ON conditions
   PICTURES
   expression evaluation workspace
   dope vector skeletons
   symbol table entries
   AUTOMATIC chain definitions
   DED dictionary entries
   FED dictionary entries
   temporary dope vectors
   BCD entries
   second file statements

8. Dimension tables

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

5. CODE BYTES FOR DATA, LABEL, AND STRUCTURE DICTIONARY ENTRIES

The First Code Byte - Other 1

Bit No.	Description	Set By
1	Symbol or requires load constant if label constant	Phase EL, FT, or NU
2	Defined on	Phase EL
3	Mentioned in CHECK list	Phase FO
4	Needs DVD	Various
5	Last member in structure	Phases EL or EW
6	Variable dimensions	Phase EL
7	* dimensions	Phases EL and FT
8	* 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
1	Dynamically defined	Phase EL
2	CONTROLLED major structure with varying strings	Phase EY
3	NORMAL = 0, ABNORMAL = 1	Phases EI and FT
4	Reserved	
5	Formal Parameter	Phase EI
6	INTERNAL = 0, EXTERNAL = 1	Phase EI
7 and	00 = AUTOMATIC or DEFINED or simple parameter	Phase EL
	01 = STATIC	Phase EL
8	11 = CONTROLLED	Phase EL

The Third Code Byte - Other 3

Bit No.	Description	Set by
1	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
2	Needs DED	Phase NU
3	Needs no storage for the item itself	Phase GP
4	Correspondence defined	Phase FV
5	Chameleon	Phase GP
6	Sign bit for first offset	Phase PH for STATIC and Phase PT for AUTOMATIC
7	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
8	As above but for second address slot	Phase PH

The Fourth Code Byte - Other 4

Bit No.	Description	Set by
1	Usage (i): An explicit alignment declaration has been made	Phase EL (for EW)
	Usage (ii): A constant has been produced for this structure or array	Phase JK
2 and 3	00 = Not temporary 01 = Temporary type 2 10 = Temporary not sold 11 = COBOL temporary	Phase GP, HF, HK, IM, or LB
4	Member of defined structure	Phase FV
5	Packed = 0 Aligned = 1	Phase EL
6	Major structure	Phase EL
7	No dope vector initialization	Phase GP
8	A temporary type 2 which has been incorporated in work-space 1 or RDV required. For COBOL temporaries this bit means RDV required	Phase OB

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:

Bit Number	Description
1	May be passed as an argument
2	May have an array as an argument
3	Must have an array as an argument
4	Is a pseudo-variable
5	Indicates to which of the two tables the offset refers
6	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
1	0 string 1 numeric
2	0 correct 1 error
3	0 not sterling 1 sterling
4	0 short 1 long
5	Not used
6	0 decimal 1 binary

7	0 fixed 1 floating
8	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
15-16	Dictionary reference of next item in the symbol table for this block
17-18	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'

- 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

+ Go Out Of Block

- 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	CONTROLLED	CELL
5	ATTRIBUTES	STRING	REPLY		COMPLEX	NORMAL	SECONDARY	BASED
6	PAGESIZE	SKI			PRECISION 1	USES		OFFSET
7	IDENT	LINE			PRECISION 2	SETS		
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		UNALIGNED	
E	NOLOCK	KEYFROM		,			UNALIGNED	
F	IGNORE	FORMAT LIS		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.

• Table 3. 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 to 01		Spare

APPENDIX G: SYSTEM GENERATION

For full details of the system generation process, see IBM System/360 Operating System: System Generation, Form C28-6554.

During the system generation process, a control section named IEMAF is assembled (see Figure 13) containing a table consisting of five fixed-point values aligned on full-word boundaries, immediately followed by a bit string field that is twelve bytes in length. The five fixed-point values are related to the compiler options LINECNT, SIZE, SORMGIN (start), SORMGIN (end), and CONTROL COLUMN (PAGECTL), respectively. Bits 1 to 39, and 43 to 46 in the string are used to specify the default status of the options. Bits 47 to 91 in the string are used to specify if an option keyword is to be deleted or not. A "1" in the bit string means "yes" and a "0" means "no". The remaining bits in the string are spare bits not currently in use. Figure 14 shows the bit identification table associated with the control section.

```
IEMAF    START
         DC F'60'
         DC F'99999'
         DC F'2'
         DC F'72'
         DC F'0'
         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 '     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 '     SPARE
         DC B'0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0 '     SWITCHES
```

Figure 13. The IEMAF Control Section

```
         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
```

APPENDIX I: DIAGNOSTIC MESSAGES

The messages produced by the PL/I (F) Compiler are explained in the publication IBM System/360 Operating System, PL/I (F) Programmer's Guide, Form C28-6594. The following table associates a message number with the particular phase and module in which the corresponding message is generated.

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
IEM0043I	Read In	CC
IEM0044I	Read In	CC
IEM0045I	Read In	CC
IEM0046I	Read In	CC
IEM0048I	Read In	CG
IEM0049I	Read In	CI
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
IEM0056I	Read In	CT
IEM0057I	Read In	CC
IEM0058I	Read In	CC
IEM0059I	Read In	CP
IEM0060I	Read In	CP
IEM0061I	Read In	CP
IEM0062I	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
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
IEM0115I	Read In	CL
IEM0116I	Read In	CI
IEM0118I	Read In	CL
IEM0128I	Read In	CO
IEM0129I	Read In	CL
IEM0130I	Read In	CL
IEM0131I	Read In	CO
IEM0132I	Read In	CO
IEM0133I	Read In	CO
IEM0134I	Read In	CP
IEM0135I	Read In	CP
IEM0136I	Read In	CO
IEM0138I	Read In	CP
IEM0139I	Read In	CP
IEM0141I	Read In	CP
IEM0142I	Read In	CO

IEM0143I	Read In	CO
IEM0144I	Read In	CO
IEM0145I	Read In	CO

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
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
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
IEM1040I	Translator	IM
IEM1051I	Translator	IM
IEM1056I	Translator	IM
IEM1057I	Translator	IM
IEM1058I	Translator	IM
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
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 Preprccessor	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	LA
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
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
IEM1612I	Pseudo-code	LW
IEM1613I	Pseudo-code	LS,LT,LU
IEM1614I	Pseudo-code	LW
IEM16151	Pseudo-code	ME
IEM1616I	Pseudo-code	ME
IEM1617I	Pseudo-code	MB
IEM1618I	Pseudo-code	MB
IEM1619I	Pseudo-code	MB

IEM1620I	Pseudo-code	MB
IEM1621I	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
IEM1670I	Pseudo-code	MP
IEM1671I	Pseudo-code	MP
IEM1680I	Pseudo-code	MS
IEM1687I	Pseudo-code	MS
IEM1688I	Pseudo-code	MS
IEM1689I	Pseudo-code	MS
IEM1691I	Pseudo-code	MS
IEM1692I	Pseudo-code	MS
IEM1693I	Pseudo-code	MS
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
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
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
IEM2352I	Storage Allocation	PD
IEM2560I	Storage Allocation	QU
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
IEM2833I	Final Assembly	TF
IEM2834I	Final Assembly	TF
IEM2835I	Final Assembly	TF
IEM2836I	Final Assembly	TF
IEM2837I	Final Assembly	TF
IEM2849I	Final Assembly	TJ
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
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
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
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
IEM3863I	Compiler Control	AA
IEM3864I	Compiler Control	AA
IEM3865I	Compiler Control	AA
IEM3872I	Compiler Control	AA
IEM3873I	Compiler Control	AA
IEM3874I	Compiler Control	AA
IEM3875I	Compiler Control	AA
IEM3876I	Compiler Control	AA
IEM3877I	Compiler Control	AA
IEM3878I	Compiler Control	AA
IEM3880I	Compiler Control	AA
IEM3887I	Compiler Control	AA
IEM3888I	Compiler Control	AA
IEM3889I	Compiler Control	AA
IEM3890I	Compiler Control	AA
IEM3891I	Compiler Control	AA
IEM3892I	Compiler Control	AA
IEM3893I	Compiler Control	AA
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
IEM3911I	Compiler Control	AB
IEM3912I	Compiler Control	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
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
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 Prccessor BC
IEM4361I Compile-time Prccessor BC
IEM4364I Compile-time Prccessor BC
IEM4367I Compile-time Prccessor BC
IEM4370I Compile-time Processor BC
IEM4373I Compile-time Prccessor BC
IEM4376I Compile-time Prccessor BC
IEM4379I Compile-time Prccessor BC
IEM4382I Compile-time Prccessor BC
IEM4283I Compile-time Processor BC
IEM4391I Compile-time Prccessor BC
IEM4394I Compile-time Processor BC
IEM4397I Compile-time Processor BC
IEM4400I Compile-time Prccessor BC
IEM4403I Compile-time Prccessor BC
IEM4406I Compile-time Prccessor BC
IEM4407I Compile-time Prccessor BC
IEM4409I Compile-time Processor BC
IEM4412I Compile-time Processor BC
IEM4415I Compile-time Prccessor BC
IEM4421I Compile-time Processor BC
IEM4433I Compile-time Prccessor BG
IEM4436I Compile-time Processor BG
IEM4439I Compile-time Processor BG
IEM4448I Compile-time Prccessor BG
IEM4451I Compile-time Prccessor BG
IEM4452I Compile-time Processor BG
IEM4454I Compile-time Prccessor BG
IEM4457I Compile-time Prccessor BG
IEM4460I Compile-time Prccessor BG
IEM4463I Compile-time Prccessor BG
IEM4469I Compile-time Prccessor BG
IEM4472I Compile-time Processor BG
IEM4473I Compile-time Prccessor BG
IEM4475I Compile-time Processor BG
IEM4478I Compile-time Prccessor BG
IEM4481I Compile-time Prccessor BG
IEM4484I Compile-time Prccessor BG
IEM4499I Compile-time Prccessor BG
IEM4502I Compile-time Processor BG
IEM4504I Compile-time Processor BG
IEM4505I Compile-time Prccessor BG
IEM4506I Compile-time Prccessor BG
IEM4508I Compile-time Processor BG
IEM4511I Compile-time Processor BC
IEM4514I Compile-time Prccessor BG
IEM4517I Compile-time Prccessor BG
IEM4520I Compile-time Prccessor BG
IEM4523I Compile-time Processor BG
IEM4526I Compile-time Processor AS
IEM4529I Compile-time Prccessor BC BG
IEM4532I Compile-time Processor AS
IEM4535I Compile-time Processor AS
IEM4538I Compile-time Processor BC
IEM4547I Compile-time Processor AV
IEM4550I Compile-time Prccessor BG
IEM4553I Compile-time Prccessor BG
IEM4559I Compile-time Prccessor BG
IEM4562I Compile-time Processor BG
IEM4570I Compile-time Prccessor BG
IEM4572I Compile-time Processor BG
IEM4574I Compile-time Prccessor BG
IEM4576I Compile-time Prccessor BG

IEM4578I Compile-time Processor BG
IEM4580I Compile-time Processor BG

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.

The action on encountering a BUY statement is similar to that for the 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.

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.

Phase QX

Phase QX is the 'AGGREGATE LENGTH TABLE' printing phase. It is entered only if the ATR (attribute list) option is 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 an aggregate with the CONTROLLED attribute, 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 aggregate length is not known at compilation the word "adjustable" is printed in the 'length in bytes' column. In the case of a DEFINED aggregate, the word 'DEFINED', and not the aggregate length, appears in the 'length in bytes' column.

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

THE REGISTER ALLOCATION LOGICAL PHASE

The purpose of the Register Allocation Phase is to insert 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.