CRAY RESEARCH, INC.

CRAY X-MP AND CRAY-1® COMPUTER SYSTEMS

FORTRAN (CFT)
INTERNAL REFERENCE MANUAL

SM-0017

RECORD OF REVISION

PUBLICATION NUMBER SM-0017

Each time this manual is revised and reprinted, all changes issued against the previous version are incorporated into the new version and the new version is assigned an alphabetic level.

Every page changed by a reprint with revision has the revision level in the lower righthand corner. Changes to part of a page are noted by a change bar in the margin directly opposite the change. A change bar in the margin opposite the page number indicates that the entire page is new. If the manual is rewritten, the revision level changes but the manual does not contain change bars.

Requests for copies of Cray Research, Inc. publications should be directed to the Distribution Center and comments about these publications should be directed to:

CRAY RESEARCH, INC.
2520 Pilot Knob Road
Suite 310
Mendota Heights, Minnesota 55120

Revision	Description
	October 1980 - Original printing.
A	June 1981 - This reprint with revision includes implementation of the character data type and other miscellaneous changes that bring the manual into agreement with the version 1.10 release. All previous printings are obsolete.
A-01	May 1983 - This change packet brings the manual into agreement with the CFT 1.11 release. Major changes include the addition of intrinsic function processing, the #CL Text Table (TBCLTXT), the Substring Definition Table (TBSB), the Intrinsic Function Name Table (TBJ), the Intrinsic Function Attribute Table (TBK), calling sequence information, and the following subroutines: CCAT, CCLA, CCLO, CCRS, CCTB, CKRF, CLCF, CLGA, CLOF, CLOG, CLRS, CLSZ, CLTG, CPOP, FRTG, FSHD, FSUB, IPRN, IRST, MSAR, MVOP, NARG, SIN, and SOPT. The register numbers were moved to tables 4-1 and B-1. Miscellaneous technical and editorial changes are also included.
A-02	July 1983 - This change packet, along with A-01, brings the manual into agreement with the CFT 1.11 release. The default of IF optimization is changed from OPT=PARTIALIFCON to OPT=NOIFCON on the CFT control statement.
B	November 1983 - This reprint with revision incorporates revision A and change packets A-01 and A-02. No other changes have been made.

B-01 February 1984 - This change packet brings the manual into agreement with the CFT 1.13 release. The CFT release has been numbered 1.13 in conjunction with the 1.13 COS release. Major changes include the addition of reentrancy support, new instruction scheduler information, the Entry/Exit Address Table (TBEE), the Call-by-value Reference Table (TBFR), the Program Description Table (TBH), the Plus Dependency Table (TBPD), the Sequence Number Table (TBSN), the Saved Variable Table (TBSV), the following subroutines: ABRA, ARUS, ASVL, ASVM, BLCN, BTD, COPR, CRAR, CRRG, CRVR, DORP, EVOP, FLVS, FSTK, GCRF, GIXA, GSBS, ISRF, LSOM, LSOV, MCEX, NICV, NOCV, PMRT, PMST, PPDP, RBIN, RBLI, RBRG, RCCK, RSTB, SAST, SA50, SDCO, SPFH, SPFR, TPRU, ZMEM, and Appendixes D and E. Miscellaneous technical and editorial changes are also included.

B-02 December 1984 - This change packet brings the manual into agreement with the CFT 1.14 release. Major changes include the addition of the Block Definition Table (TBBK), the Register Variables to Restore After a CALL Table (TBCALL), the Character Length Table (TBCLEN), the Conjunctive Term Table (TBCT), the Disjunctive Term Table (TBDT), the Label Usage Table (TBLB), the TBT Index of Variables Not Assignable to B/T Register Table (TBNOBT), the TBX Extension Table (TBXX), and the following subroutines: AIBF, CDPR, CEXP, CFBI, CIDN, CQYL, CRMV, CRNK, CTTY, DOUN, EBSN, EBXR, EBXS, ECNT, ECNU, EDJT, EDJU, ERTX, ESBK, ESNL, ETBX, FPAR, GBAT, GCBS, GLBD, GTCB, IGXF, IVTX, LBLK, NOBTVAR, PBLK, PCIV, PCST, PLDP, RBMV, RDPT, ROSR, SDPF, SFMN, SGES, SPRN, TFBK, and VLAN. Miscellaneous technical and editorial changes are also included.

B-03 January 1986 - This change packet brings the manual into agreement with the CFT 1.15 release and supports the COS 1.15 and UNICOS† 1.0 releases. Major features affecting this revision are code and data separation, generalized segment length, vector temporary storage, vectorized search loops, DO-loop table enhancements, LOOPMARK utility, and generalized loop CIVs; routines added or changed are BOFG, CDSP, DLTB, ENST, GCBS, GDEX, GDLB, GRLD, GRRL, GRST, GRSV, LLIV, LMRK, MIAR, MSST, RBVT, RBVU, RVLR, UCIV, and UCDI. Miscellaneous technical and editorial changes are also included.

† UNICOS is derived from the AT&T UNIX System; UNIX is a trademark of AT&T Bell Laboratories.

PREFACE

This publication is part of a set of manuals written for programmers, analysts, and field engineers who have the responsibility of installing, debugging, and modifying the Cray operating system COS or UNICOS.

This manual describes the internal design of the Cray FORTRAN Compiler (CFT), Version 1.

Section 1 briefly introduces the compiler and describes CFT conventions.

Sections 2 and 3 provide a basis for understanding the compiler's operation. These sections describe the general flow of the compiler through Passes 1 and 2, respectively.

Section 4 describes table management and gives the specifications for the tables used by the compiler.

Section 5 gives the details of the major subroutines within the compiler.

Section 6 provides information about CFT I/O.

The appendix section gives additional information about the compiler's internal design, including pertinent reference information and sample code.

Related publications are:

SR-0000	CAL Assembler Version 1 Reference Manual
SM-0007	IOS Table Descriptions Internal Reference Manual
SR-0011	COS Version 1 Reference Manual
SM-0040	COS EXEC/STP/CSP Internal Reference Manual
SM-0041	COS Products Set Internal Reference Manual
SM-0042	COS Front-end Protocol Internal Reference Manual
SM-0043	COS Operational Procedures Reference Manual
SM-0044	Operational Aids Reference Manual
SM-0045	COS Table Descriptions Internal Reference Manual
SM-0046	IOS Software Internal Reference Manual

Manuals designated as internal describe the internal design of the software whereas the other manuals in the set define procedures and external features of tools needed for installing and maintaining CRI software.

All values specified in this manual are expressed in octal unless otherwise noted.

CONTENTS

FIGURES (continued)

TABLES

INDEX

COMPILER OVERVIEW 1

1.1 GENERAL DESCRIPTION

The CRAY-1 FORTRAN Compiler (CFT) is a two-pass compiler that converts statements from the FORTRAN language to the binary machine language of the CRAY-1 Computer Systems. CFT constructs CRAY-1 machine-language instruction sequences that cause the full range of CRAY-1 features and capabilities to be applied during program execution.

CFT is written in CRAY-1 Assembly Language (CAL) and executes under the Cray operating systems COS and UNICOS. It has no hardware requirements beyond those required for the minimum system configuration.

CFT analyzes and compiles source code one program unit (main program, subroutine, function, or block data subroutine) at a time. No information is retained from one program unit to the next.

During compilation, CFT constructs a number of tables in the user area. A *table* is basically a list of information kept for referencing by the compiler. Examples of the tables maintained by CFT are a list of the symbols used in the source program and a list of constants encountered.

The compiler tables are located at the high end of user memory immediately below the user Dataset Parameter Area. Space is allocated to each table as needed. Table length is variable, with most tables expanding as compilation proceeds. If the table area overflows its allotted memory, CFT requests additional memory from the operating system.

Figure 1-1 illustrates the two-pass philosophy of CFT. The input to Pass 1 of CFT is the source input dataset consisting of the source code file and the accompanying control statement. The principal output from Pass 1 is a copy of the source program translated to an internal format and a symbol table describing the attributes of the symbols encountered in the source program. The Pass 1 output becomes the input to Pass 2. Pass 2 finishes compilation and provides as its output the loader tables, suitable for loading and execution.

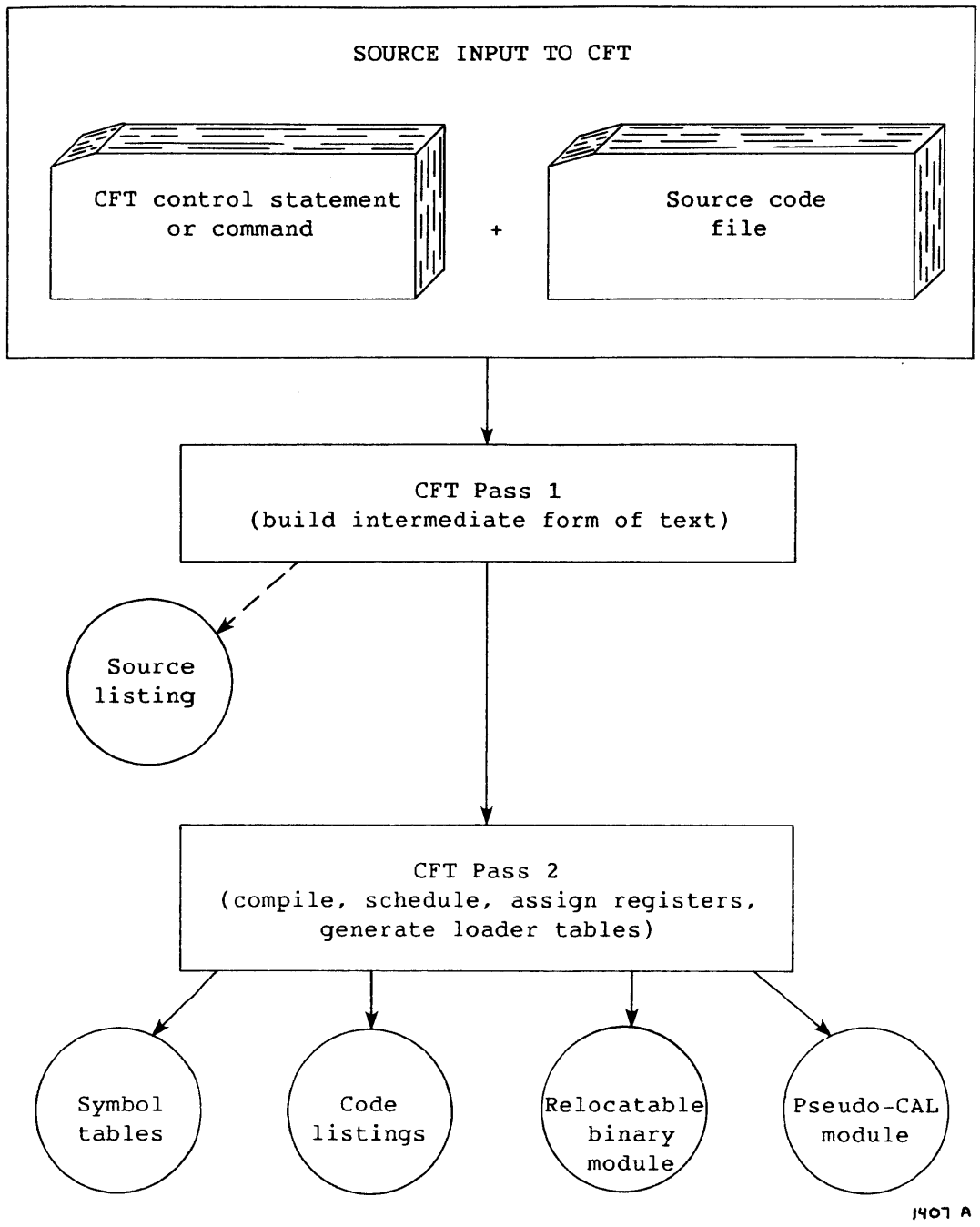


Figure 1-1. CFT's two-pass philosophy

The compiler is loaded in the user field by the operating system when it encounters a CFT command or control statement. Each user receives a copy of the compiler in the user field. The compiler itself is not re-entrant. Parameters of the CFT command or control statement specify characteristics of the compiler run, such as the datasets or files containing source statements and list output.

1.1.1 PASS 1

Basically, Pass 1 of CFT converts the source code to an intermediate form to facilitate Pass 2 activities. Figure 1-2 illustrates the general flow of Pass 1. A brief summary of Pass 1 activity follows. Section 2 of this manual provides a more detailed description of the flow through Pass 1.

After CFT reads and interprets the CFT command or control statement, it initializes the tables, presets default values, and opens the files or datasets required by the job. The preset defaults retain their values for any options not specified by the user on the CFT command or statement.

At this point, the main loop of the compiler begins. Most pointers are cleared and the tables are set to empty. The statement sequence number is initialized at 0. This procedure is executed at the start of each program unit. Then, compilation begins by calling CFT's card reader driver. The card reader driver reads the source input file statement by statement, checking for continuation lines and comment lines.

As each statement is read, it is copied into the compiler's statement buffer, where it is examined and classified by type. Then, depending on statement type, control transfers to one of the many *unique statement processors*. A *unique statement processor* is a handler for only one specific FORTRAN statement; for example, there is a unique statement processor for WRITE statements and another for DIMENSION statements.

Each FORTRAN statement is classified as either executable or non-executable. An *executable* statement specifies an action, while a *non-executable* statement is an inactive descriptor of data (declarative) or program form.

With few exceptions, statements are completely processed as they are encountered. Exceptions to this are the DO and EQUIVALENCE statements. DO statements generate table entries that trigger additional processing when the terminal statement is encountered. EQUIVALENCE statements are packed into a table as they are encountered and are processed when the first executable statement (that is, the last declarative statement) is encountered.

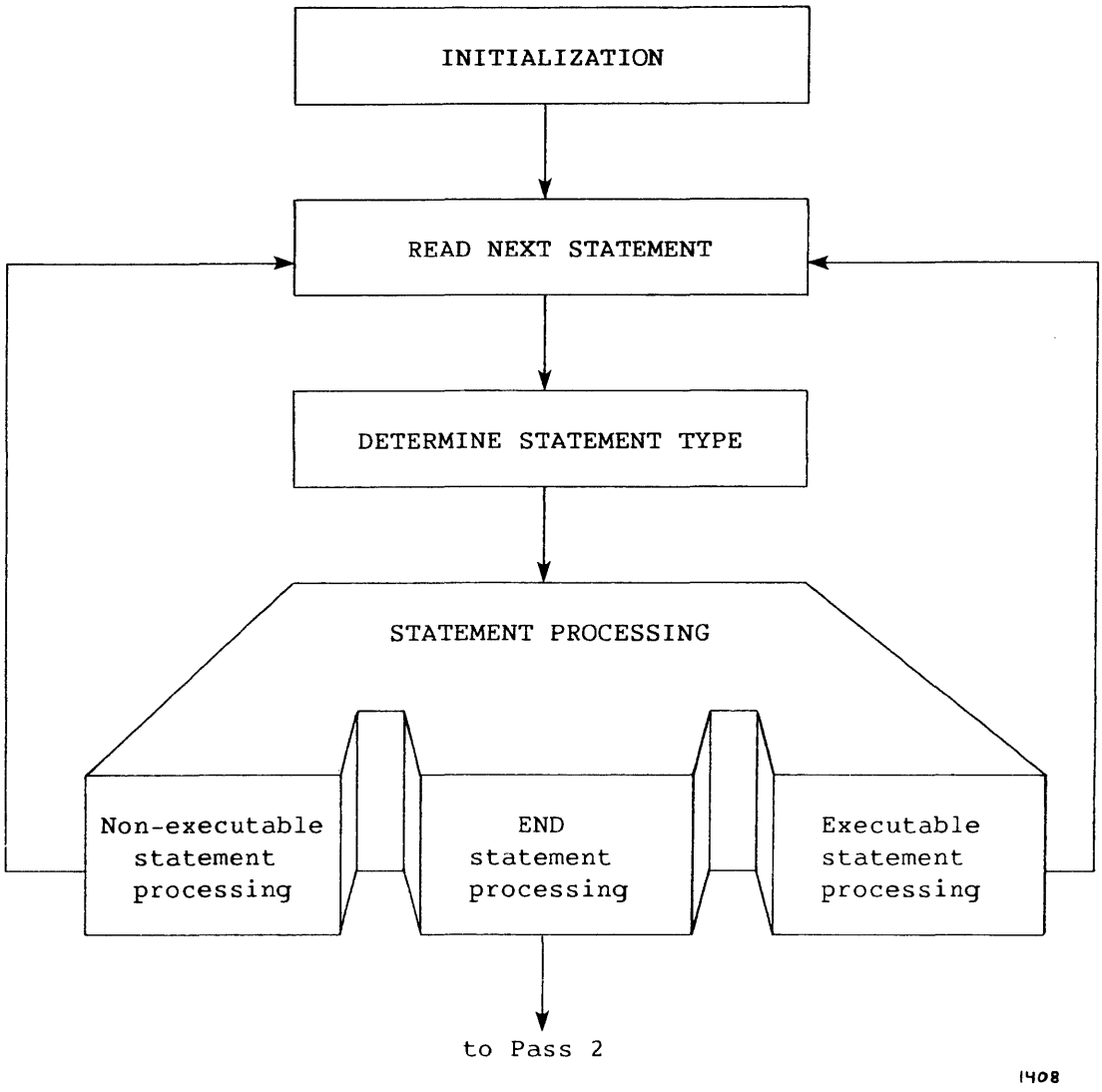


Figure 1-2. Pass 1 Overview

Many FORTRAN statements have similar syntax. For example, the syntaxes of READ, WRITE, and PRINT statements are similar, as are those of REAL, DIMENSION, and COMMON statements. In such cases, one of the unique statement processors is called to process the initial keyword. This unique statement processor then branches to an appropriate *common syntax processor* to handle the syntax held in common by the statements.

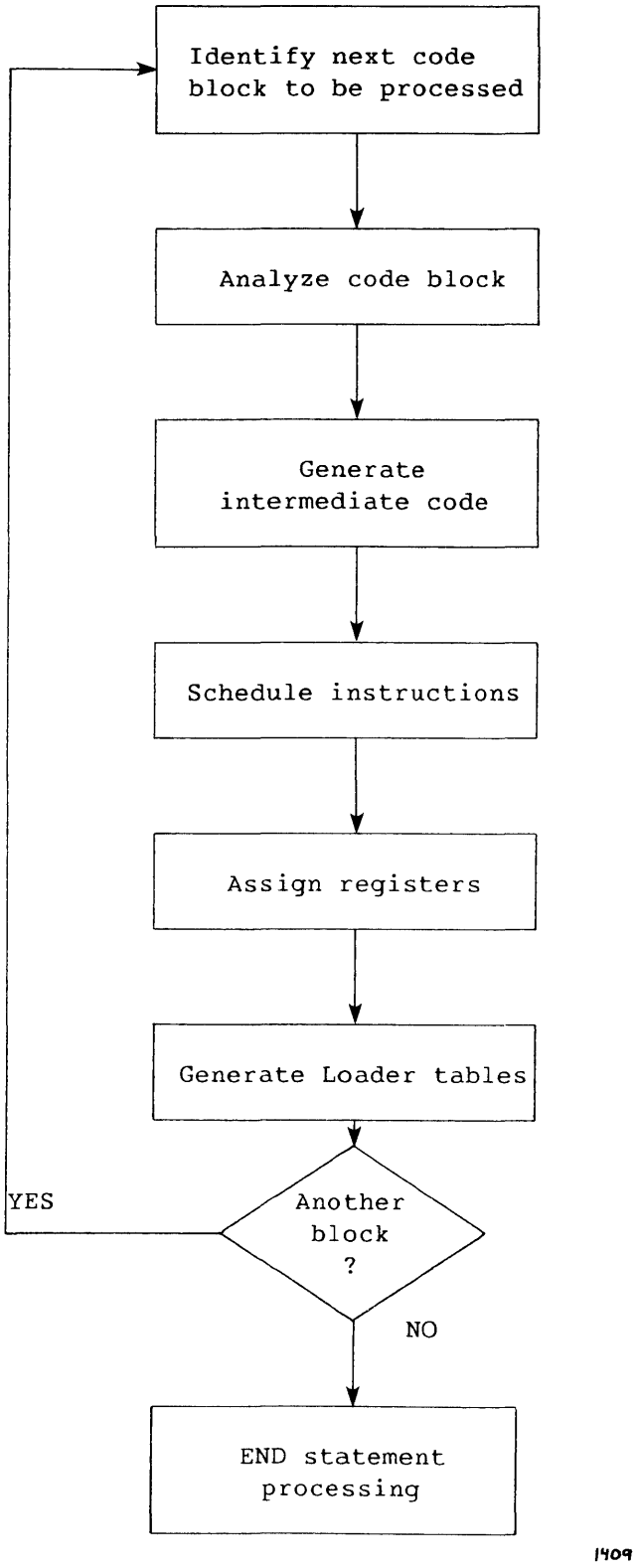


Figure 1-3. Pass 2 Overview

1.2 TABLE NAMES AND INDEXES

CFT maintains nearly 50 tables during compilation. Initially, all the tables are empty. Then, as compilation proceeds, the tables expand in memory.

Several conventions exist for naming tables. Each table has a unique name, often a mnemonic for the function the table performs. The form is TBx, where x is the table identifier. For example, a table that maintains array information is TBA, the Array Table.

The index to a table is of the form KTx. For example, the index to TBA is KTA. Pointer words to the CFT tables are maintained in register V7. The pointer word contains the first word address (FWA) and the last word address + 1 (LWA+1) for the table. (Refer to Appendix B for more information.)

Table indexes are relative to the beginning address of the appropriate table. Except for the Symbol Table, the Tag Buffer Table (TBG), and the Program Unit Name Table (TBPN), the FWA of a table is always a multiple of 100_8, while the LWA+1 changes as necessary.

Refer to the section entitled Compiler Tables for table descriptions and a summary of table management.

1.3 CFT MEMORY ORGANIZATION

The compiler is loaded for each CFT command or control statement and is reinitialized rather than reloaded for each program unit. In a multiprogramming environment, several copies of the compiler can be in memory at one time because a copy goes in the field of each CFT user.

Figure 1-4 illustrates the organization of the memory area occupied by one FORTRAN user job. CFT code is at the low end of the user field, with the various routines comprising CFT arranged in approximately alphabetical order. Immediately above that is an area allocated for two compiler tables, the Library Name Table (TBL) and the Library Macro Table (TBM). Above that is the Record Image Buffer (RIB), which contains one source line at a time, followed by the library routines. During Pass 1, the statement currently being processed by CFT is stored in the Character Buffer (CHB), which is at the low end of blank common.

PASS 1 FLOW 2

2.1 INTRODUCTION

Pass 1 of the CRAY-1 FORTRAN Compiler performs the following functions:

- Compiler initialization
- Statement-by-statement processing of the source file:
  - Read source statement
  - Determine statement type
  - Process statement
- END processing for Pass 1

The input to Pass 1 is the source input file. The main output from Pass 1 is the information contained in three tables:

- The Symbol Table (TBS) - Contains the names of all symbols in the source program
- The Tag Table (TBT) - Holds the attributes of all of the symbols listed in TBS
- The Tag Buffer Table (TBG) - Contains a copy of the source program translated into an internal tag-and-operator format

2.2 INITIALIZATION

Routine BGIN (begin compilation) handles both initialization at the beginning of processing and initialization at the start of each program unit. Control transfers to BGIN at the start of processing to initialize the compiler. Then, each time processing begins on a new program unit, control transfers to BG10, a location within the routine BGIN, to reinitialize for that program unit.

2.2.1 INITIALIZATION AT BGIN

BGIN calls the control card cracking routine CARD to read the CFT command or control statement. A copy of the CFT command or statement is stored in words 5 through 77 of the operating system's Job Communication Block. Once the CFT command or statement is decoded, CARD determines which datasets or files are required and opens them.

CARD also collects the list options and error processing options in the CFT command or statement and sets indicator bits in register T.OCW to show whether the corresponding options are on or off. Any options not specified by the user on the CFT command or control statement are set to the default values. The default options are generally contained in the block of constants at the front of the compiler and can be changed at compiler assembly time.

2.2.2 INITIALIZATION AT BG10

The main loop of the compiler begins at label BG10. Control returns to BG10 at the start of each program unit to reinitialize CFT. Pointers are cleared and table pointers are reset to indicate that the tables are empty except for the Program Unit Name Table and the Page Number Table (TBPN and TBPG), which are saved from pass to pass. Storage registers are zeroed and the statement sequence number is initialized at 0.

Compilation begins with a call to routine RNXT.

2.3 READ SOURCE STATEMENT

Routine RNXT (read next statement) is the compiler's card reader driver. RNXT reads the input dataset ($IN or its equivalent) statement by statement, checking for statement continuation and comments. An internal buffer builds one complete FORTRAN statement at a time into a buffer. If a source listing is requested, lines are written to the listing file as they are read.

Two buffer areas are used to assemble a statement. The Record Image Buffer (RIB) can accommodate one card image and contains the image of the next card to be processed. The Character Buffer (CHB) contains one complete FORTRAN statement character image including an initial line and up to 19 continuation lines.

The First Card Buffer (FCB) is maintained if no source listing was requested. The FCB receives the first line of each statement so that the line can be printed out if an error occurs during processing.

encountered, variables are assigned addresses. By the end of Pass 1, all variables are known. Variables occurring in EQUIVALENCE statements have already been given addresses by this time, since they are handled at the end of nonexecutable statement processing. Local variables, however, must be assigned addresses. ENST resolves all EQUIVALENCEs and assigns addresses to all variables in TBT.

DATA statement entries are also made in TBB, the Loader's Text Table. Then ENST begins building TBH, the Loader Program Description Table. TBH holds program name, common block name and length information. Following this are ENTRY names and EXTERNAL names.

Each statement having a statement number also has a pointer in bits 1 through 17 of the statement header entry in TGB that points to the corresponding TBT statement number entry. At the end of Pass 1, all statement number references are linked together, and unreferenced statement numbers are deleted. The index to the TBT entry for an unreferenced statement number is cleared from the TGB statement header entry. Although the statement number still exists in TBS and TBT, no pointers to these table entries exist. An unreferenced statement number is transparent to CFT Pass 2.

ENST copies intermediate code generated as a result of variable dimension declarators from TBQ to immediately after each ENTRY statement header. ENST also links all statement numbers and references. Any statement number having no reference is deleted. If a statement number is never referenced, the pointer to TBT in the statement entry header word is cleared and, although the statement number is still in TBT, there is no pointer to it. This means statement numbers cannot be used only to break up blocks of code. Finally, the Intermediate Tag Buffer (TGB) is moved to the Tag Buffer Table (TBG) and actual addresses relative to the appropriate block are filled in for all symbols. The move is done word-for-word. As each item is moved, its tag is examined and looked up in TBT and the actual address or offset is put into the tag.

At EN81, some of the tables are cleared and released in preparation for Pass 2.

The final step of pass 1, if extended memory addressing (EMA) is enabled, is to change all 602 program block tags to 607 block tags. This allows CFT to separate the code from the data and to reference up to 16 million words of local data.

Statement numbers encountered during Pass 2 are not limited to those that were included in the original source code. During Pass 1 processing, CFT inserts made-up statement numbers for processing logical IF and block IF statements when it is necessary to jump from a block. CFT also inserts made-up statement numbers at both the beginning and the end of a DO loop when processing a DO. A made-up statement number is also inserted immediately following an ENTRY other than the primary. These made-up statement numbers go through Pass 2 analysis the same as any programmer-defined statement numbers.

3.2.2 MARK CONSTANT INCREMENT VARIABLES

The section of code beginning at AB20 builds TBZ. TBZ contains an entry for each variable defined within the code block. The information in TBZ is used later in Pass 2 in handling constant increment variables and in building the Cross Reference Overflow Table (TBV).

Starting at AB40 is a section of code that finds and marks all constant increment integers in TBG. In loop mode, a *constant increment variable* (CIV) is a variable that is incremented by an invariant expression at only one point in the loop. An example of a CIV is the DO control variable in a DO block where the index is an integer. CIVs are located for two reasons: subscript optimization in general and vectorization in particular. The type of a CIV can be INTEGER or REAL.

In a replacement statement, a CIV can be either a function of itself (for example, I=I+1) or a function of another variable that is a CIV (for example, I=J+1, where J is a CIV). The only operators allowed in a CIV expression are + and -. One operand may be variant, but all others must be invariant within the loop.

When a CIV is identified, a flag is set in the TBZ entry for the variable. All references in TBG made to that variable also have a flag set to indicate that that variable is a CIV.

3.2.3 ANALYZE ARRAY REFERENCES FOR DEPENDENCIES

Routine ADEP (analyze dependencies) checks for dependencies within arrays. Vectorization is inhibited if a dependency exists; however, the programmer can override this with the CDIR$ IVDEP directive.

ADEP builds the Plus Dependency Table (TBPD). TBPD is used in code generation to move a vector load before a vector store.

ADEP is a double loop. The outer loop, which drives ADEP, takes each successive definition entry from the Defined Variable Table (TBZ). A definition entry is one in which the defined item appears on the left-hand side of a replacement statement, or the item is used in an input statement or is an argument in a subroutine or function that might have side effects. For each definition entry, the inner loop of the routine searches the entire block for references to the item defined. ADEP compares the definition with subsequent definitions and other references made to it in the block, looking for dependencies (EQUIVALENCE overlapping, for example). If ADEP finds an ambiguous dependency with a condition for safe vectorization, the condition is entered into the Conjunctive Term Table (TBCT). If ADEP finds an unambiguous dependency or an ambiguous dependency with no condition for safe vectorization, the Vector Loop flag (VLF) is turned off. The VLF is global to a block and is located in register S7.

ADEP builds TBY, which is used by the optimizer in load-and-store operations. Each variable within a loop has a definition entry followed by an entry for each reference made to the variable. ADEP proceeds through the entire block, even if not in loop mode or if a dependency has been found, because TBY must be completely built for the instruction scheduler and the load/store generation routines.

3.2.4 PROMOTE CONSTANTS WITHIN SUBSCRIPT EXPRESSIONS

Basically, routine PCON (promote constants) has as its task the cleanup of all subscripts within the code block. PCON cycles through the code block looking for array references. Each time PCON finds an array reference, it cycles through the reference looking for constants within subscripts. Each constant occurring within a subscript is multiplied by the appropriate dimension multiplier and then added into the initial term of a subscript expression where it acts like a bias.

For example, if the dimension is A(10,10,10), the subscript

```
A(I,3,J+2)
```

is processed as follows:

Pass 1 expands the subscript to

```
@ A + 0 + 1 * (I-1) + 10 * (2) + 100 * (J+1).
```

PCON extracts the constants, leaving

```
@ A + 119 + 1 * (I) + 100 * (J).
```

As a result of PCON, an array reference looks like a base address plus a constant plus terms that involve variables within the subroutine for the rest of the subscript. The constant is the sum of all constants from all subscripts. PCON calls routine SVEC to locate possible scalar temporary definitions in a vector loop. SVEC then sets the vector array flag (VAF) in subsequent references to the scalar temporary within the TBG block.

3.2.5 EXAMINE ARRAY REFERENCES AND FUNCTION REFERENCES

Array and function references are examined by routine EAFR. For each statement in a block, EAFR does a backward scan looking for array and function references. Within a statement, it scans from right to left so that it can sort out the parentheses within the statement.

Array references are checked by EAFR to determine whether they are vectorizable. An array element is a candidate for vectorization if its subscripts meet the following general rules:

1. The variant subscripts can contain only linear references to a CIV.

2. The only operators allowed in the variant subscript are +, -, and * on either side of the CIV; otherwise, +, -, *, /, and ** are allowed.

If all of these conditions are met, then EAFR sets a flag called the vector array flag (VAF) for this particular array reference. This flag is set on a term-by-term basis.

EAFR also sets the variant subscript flag (DSF) in the array tag if the subscript has any variants. A vectorizable array has this flag set, but a non-vectorizable array might also have a variant subscript. A subscript is *invariant* if it is not changed within a loop and there are no stores anywhere into the array. A subscript is *variant* if it is changed within the loop or if there is any store into the array.

EAFR looks at function references to ensure all arguments are proper vector arguments. It checks a flag in the function tag for a vector version available for the function. If a vector version exists, EAFR looks at each argument of the function to see if it is invariant and to see if its VAF is set. If either condition is met, then the function reference is vectorizable.

3.2.6 TRANSFER TO VECTOR CONTROL

If processing is in loop mode, control is transferred next to routine VCTL (vector control). VCTL consists of three main sections.

The first section of VCTL copies several flags including the VAF from TBG into TBZ.

The second section of routine VCTL searches each tag in order. Whenever VCTL finds a condition that turns off vectorization, it turns off the vector loop mode flag (VLF) and returns to the Compile Block routine (CBLK). If the tag is that of a variant (the variant bit is set from ABLK), then it is a vector.

A scalar temporary, even though it is not an array, has had its VAF set by GVEC; VCTL thus treats it as an array. VCTL also looks for recursive sums.

When VCTL is done, it generates the necessary calculations at the beginning of the loop to set the vector length register and then returns to CBLK. CBLK generates vector instructions because the VLF is set.

The third part of VCTL generates the incrementation for the CIVs found in the code block. It is called when CBLK finishes the loop. The information in TBZ is used for this purpose.

3.3 GENERATE INTERMEDIATE CODE

As a result of Pass 1, the number of different FORTRAN statements in a program is reduced to very few. By the end of the pass, CFT has restructured the program unit so that it contains the following types of statements: replacement statements, CALLs, IFs, GO TOs, and ENTRY statement headers.

Generating intermediate code is driven by routine CBLK (compile block). CBLK handles CALL statements and replacement statements. IF statements are handled by the code beginning at IF50 and GO TO statements cause control to transfer to GT30; however, this transfer does not occur until after CBLK generates the code for all expressions associated with the statements.

All expressions are handled by CBLK as general-purpose expressions even if they are basic 1-term expressions (for example, each argument of a CALL statement or the expression in the parentheses in an IF statement).

CBLK calls PBLK to find the next statement to be compiled. CBLK tries to compile the located statement by finding the innermost set of parentheses. The innermost set of parentheses is compiled, the parentheses are removed, and the process is repeated until all parentheses have been removed. Final statement processing occurs when the parentheses processing ends. Then, any remaining stores are compiled and IF statements or GO TO statements are completed.

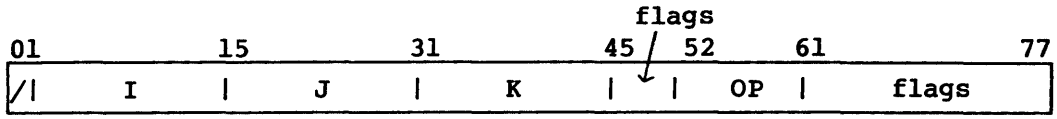
When CBLK finds an open parenthesis, it begins a series of forward scans within the parentheses. By definition, the first open parenthesis found in a backward scan is the innermost. CBLK puts itself in a loop and calls routine OLEV (operator level), which makes repeated scans through the expression, looking for operators in precedence order.

CBLK calls PTRI (process triad) for each operator OLEV finds in the expression. PTRI checks for index processing, and either calls CTRI or extracts the index increment.

When CBLK determines it is processing an intrinsic function reference, it evaluates each of the function parameters and calls routine INFN (intrinsic function generator) to expand the skeleton for the function and generate instructions. INFN returns to CBLK.

Intermediate code is generated one word at a time in PIB, the Pseudo Instruction Buffer. Most of the code inserted in PIB is generated by routine CTRI. As each instruction is generated, one word is stored into PIB. Register A7 contains the address where the next instruction can be stored; A7 is incremented by 1 after each store to prepare for the next instruction.

The format of a stored instruction is:



Field	Bits	Description
I	1-14	I field of instruction, pseudo register number, or parameter number
J	15-30	J field of instruction, pseudo register number, or parameter number
K	31-44	K field of instruction, pseudo register number, or parameter number
Flags:	45-50	
	45	If set, I field is result or is unused; if clear, I field is operand pseudo register or constant.
	46	If set, J field is result or is unused; if clear, J field is operand pseudo register or constant.
	47	If set, K field is result or is unused; if clear, K field is operand pseudo register or constant.
	50	If set, I is both an operand and a result (as in shift operations)
OP	52-60	Opcode for instruction to be generated

Two special cases exist for the FLG and OP fields. If FLG bits 45, 46, and 47 are set and OP=005, the entry represents an entry or exit sequence. If FLG=0, OP=0, and I=0, the entry represents a special-case instruction sequence.

Flags:	61-77	
	75	If set, instruction can be delayed to postamble
	77	If set, J and K are invariants and instruction can be removed to preamble

When instructions are generated, pseudo registers are assigned instead of real registers. A *pseudo register* is an imaginary register not corresponding to any *hard (real) register*. Pseudo

4.3.2 TAG DEFINITIONS

Each entity (variable, statement number, external name, etc.) within a statement is converted to a descriptive tag during Pass 1 processing. The first three octal digits (9 bits) of a tag describe the basic type of entity.

Tags are used in many compiler tables. Table 4-2 describes tag types.

Table 4-2. TGB Tag Descriptions

Tag	Description
100	Pseudo tag. Used primarily in DO statement processing when a tag is desired but no memory needs to be assigned. Only used in TGB.
101	Statement number tag
102	External function tag
103	Inline function tag
104	Statement function tag; arithmetic statement function.
106	Subroutine entry name tag
107	Function entry name tag; also used for implied-DO variables in DATA statements.
110-577	Dummy argument tags. Assigned in consecutive order; 110 is assigned the first dummy argument allowing for over 300 arguments per subroutine or function, or Pointee tags. Assigned in consecutive order; 110 is assigned the first pointee tag.
600*q*	Constant tag; refers to a constant rather than a variable. The digit immediately following the tag (*q*) is the subtype for the type of constant. Subtypes are as follows: *q* value — Explanation 0 — Refer to TBB; the offset field is an index into TBB. 1 — Constant can be machine-generated using an 071 machine instruction 4 — Immediate constant; 22-bit constant is supplied. 6 — Shifted constant; 22-bit constant entered in S register and shifted left 51 bits.

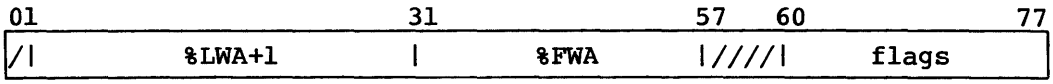
Table 4-2. TGB Tag Descriptions (continued)

Tag	Description
601	Used for dummy argument addresses. The offset field gives the offset from the address in B01 to the address passed in for a dummy argument.
602	Program block; positive relocation with respect to the origin of the current program. The program block is used for generated code, static variables and arrays, and constants.
603 (#TB)	In static mode, #TB holds temporary variables local to a code block; space in #TB is reused from block to block. #TB is not used in stack mode.
604 (#CL)	In static mode, #CL holds argument lists and the space into which passed-in argument lists are copied for multiple-entry routines. In stack mode, #CL holds argument list headers only (the headers are built as compile-time constants).
605 (#ST)	#ST is not used in static mode. In stack mode, all stacked entities except the B/T save area are in #ST (including entities in #TB and most entities in #CL in static mode). The offset field gives the offset from the run time address in B03 to the first word of the stacked entity.
606 (#RG)	Used for variables globally assigned to the B and T registers
607 (#DA)	Holds data (constants and static arrays and variables not in common)
	Tags 601, 605 (#ST), and 606 (#RG) are used internally by CFT; no loader tables are generated for them. Tags 602, 603 (#TB), 604 (#CL), and 607 (#DA) are treated as local blocks by the loader; a program unit using one of these blocks is assigned a unique (nonshared) area of memory.
610-777	User-declared common block tags. Assigned in consecutive order; 610 is assigned the first common block. This assignment allows up to 120 common blocks per subroutine or function. The loader treats these tags as common blocks; references to the same common block by two program units are treated as references to a single (shared) area of memory.

4.TBBK TBBK - BLOCK DEFINITION TABLE

TBBK describes the characteristics of each statement in a block, such as the beginning of the block, the end of the block, and factors that can inhibit vectorization.

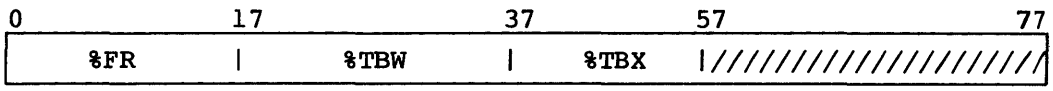
Format before compilation:



Field	Bits	Description
%LWA+1	1-30	%LWA+1 statement in TBG (before compilation)
%FWA	31-56	%FWA statement in TBG (before compilation)
Flags:	60-77	
BSR	60	Statement has search exit
BNM	61	Statement cannot have vectorizable minus dependency
BOT	62	Statement not part of loop
BVL	63	Vectorizable function reference
BLS	64	ELSE block
BUB	65	Unconditional branch
BBR	66	Change in flow of control
BET	67	Block is executed every time
BVF	70	Nonvectorizable function reference
BLF	71	Block in loop
BFE	72	Forward entry within group
BND	73	End of group
BXE	74	Enter from outside group
BEX	75	Calls function or subroutine

Field	Bits	Description
BSP	76	Sub-block has a plus dependency
BSM	77	Sub-block has a minus dependency

Format after compilation:

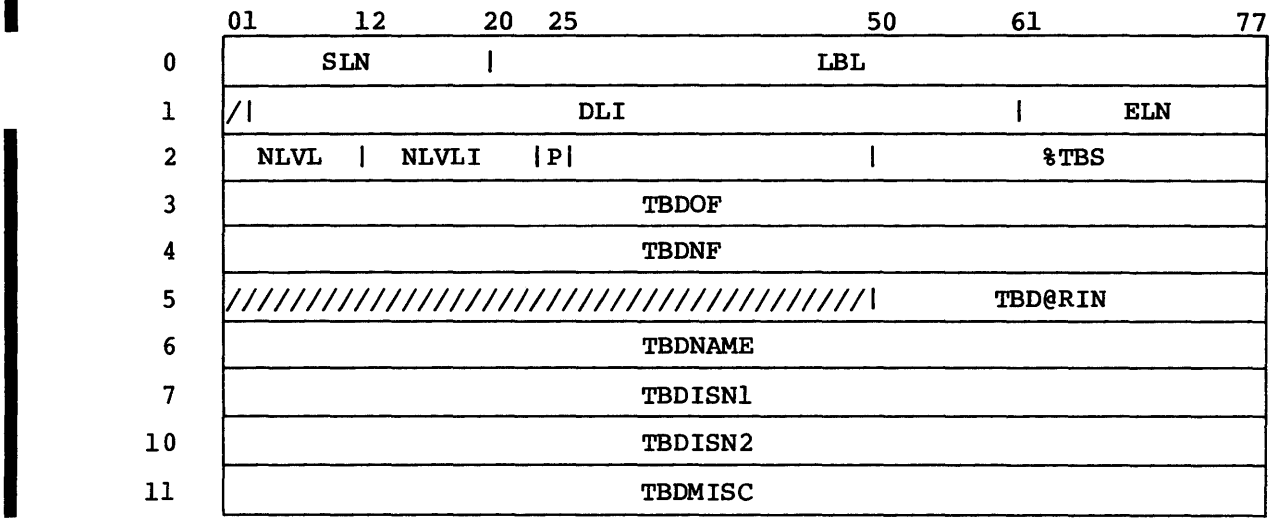


Field	Bits	Description
%FR	0-16	Length of TBFR (after compilation)
%TBW	17-36	Length of TBW (after compilation)
%TBX	37-56	Length of TBX (after compilation)

4.TBD TBD - DO LOOP TABLE

TBD contains a 10-word entry for each DO loop encountered by the compiler.

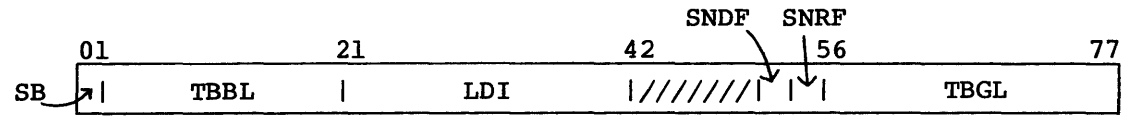
Format:



Field	Word	Bits	Description
SLN	0	0-17	Starting line number; keyed to the source listing line number.
LBL	0	20-77	Label that ends the DO loop; 6 ASCII characters, right-justified, and zero-filled.
DLI	1	1-60	DO-loop index; 8 compressed (6-bit) ASCII characters, left-justified. A bias of 40B is used for compression.
ELN	1	61-77	Ending line number, keyed to the source listing line number
NLVL	2	0-11	Maximum depth of DO loops nested in this one, ignoring implied DOs.
NLVLI	2	12-25	Maximum depth of DO loops nested in this one, including implied DOs.
POTVECT (P)	2	26	Set if this loop is potentially vectorizable (that is, if this is an innermost DO loop or all loops nested in this one are unrolled).

Field	Word	Bits	Description
%TBS	2	50-77	Index into TBS for the label that is the start of the DO loop. The TBS entry for the end-of-loop label is always the next TBS entry.
TBDOF	3	0-77	Each bit set indicates an optimization done on the loop.
TBDNF	4	0-77	Each bit set indicates a reason the loop did not vectorize.
TBD@RIN	5	0-77	Address of where SETTBD was called to fill in this TBD entry. This address is used so that NOVECTOR messages can print parcel addresses.
TBDNAME	6	0-77	Holds an 8-bit ASCII character name to be inserted into a NOVECTOR message.
TBDINS1	7	0-77	Holds an 8-bit ASCII character sequence number to be inserted into a NOVECTOR message.
TBDINS2	10	0-77	Holds a second 8-bit ASCII character sequence number to be inserted into a NOVECTOR message.
TBDMISC	11	0-77	Contains the number of a dependency message. Used only if the TBDNDEP1 or TBDNDEP2 bit is set in TBDNF.

Format of a TBT primary entry for a statement label during Pass 2:



Field	Bits	Description
SB	0	Sign bit; set for statement numbers with secondary entries. This bit is set between Pass 1 and Pass 2 when the initial jump instruction is generated to the label and stored at the offset specified by the secondary entry. This bit is set for other labels when the statement number definition is compiled and entered in TBB.
TBBL	1-20	TBB last reference index. TBBL is the index of the last reference to this label in TBB relative to T.PBS. TBBL is updated for each reference compiled and is the head of the chain of references in TBB.
LDI	21-41	Label definition index; initially an index relative to the LWA+1 of TBG pointing to the statement header of the statement where the label is defined. When the statement number definition is compiled, it becomes the index of the definition in TBB.
SNDF	54	Statement number defined flag; set until the label definition is compiled and entered in TBB.
SNRF	55	Statement number referenced flag; set for referenced statement numbers.
TBGL	56-77	TBG last reference index. TBGL is the index to the last reference of the label in TBG relative to the LWA+1 of TBG. TBLG serves as the head of the chain of references in TBG.

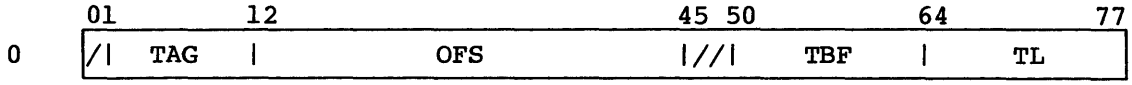
A TBT secondary entry exists for statement numbers whose parcel address must be read into a register at run time. User-defined statement numbers appearing in ASSIGN statements or END= or ERR= branches of I/O statements require secondary TBT entries. Compiler generated labels also have secondary entries when associated with the first word address of a jump table generated for alternate return subroutine calls and computed GOTOs.

4.TBZ TBZ - DEFINED VARIABLE TABLE

During Pass 2, each variable that is defined in a block is entered in TBZ. Each entry consists of two words. The first word, word 0, contains the variable tag. The second word, word 1, contains the tag location index, tag definition index, and a number of flags pertaining to constant integer analysis.

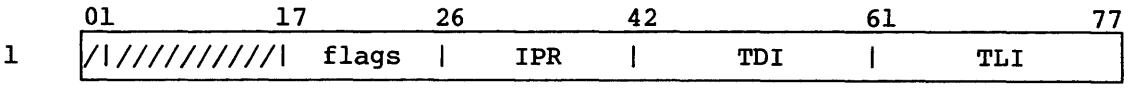
TBZ is cleared at the beginning of each new block that does not have a drop-through entry.

Word 0 Format:



Field	Bits	Description
TAG	1-11	Tag
OFS	12-44	Offset
TBF	50-63	Tag Buffer flags (refer to 4.TGB)
TL	64-77	Type and length

Word 1 Format:

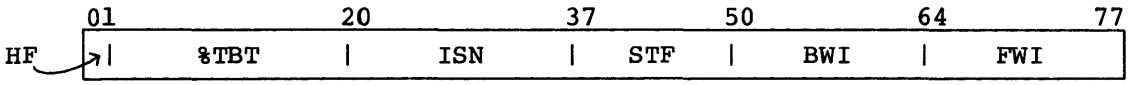


Field	Bits	Description
Flags:	17-25	
INPRF	17	Processed, set if update has been compiled
INCNF	20	Conditional increment
INTYP	21	Set for S register increment
INTRP	22	Trip count
INSLF	23	Self reference

Field	Bits	Description
Flags (continued):		
INAMB	24	Ambiguous increment if clear
INSUB	25	Subtract increment
IPR	26-41	Increment pseudo register
TDI	42-60	Tag definition index. For a replacement definition, TDI equals the block index of beginning of next statement in TBG; otherwise, TDI=TLI.
TLI	61-77	Tag location index; equal to block index of tag location in TBG.

4.TGB TGB - TAG BUFFER

During Pass 1, a TGB statement entry begins with a statement header word. This header word has its sign bit set, whereas none of the entry words do; the header word's sign bit is set so that during Pass 2, where things are processed on a statement-by-statement basis, a quick search will locate the head of each statement unit. The statement header word has the following format:



Field	Bits	Description
HF	0	Flag set to indicate header word
%TBT	1-17	Index into statement number entry in TBT, if one exists. If header is for an entry (ENF set), the field is the index into TBH of the entry name.
ISN	20-36	Internal sequence number in binary, as it appears on FORTRAN source listing on left margin
STF	37-47	Statement type flags, as follows:

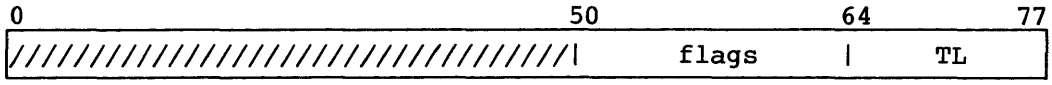
Bit	Flag	Description
37	ENF	ENTRY statement
40	DBF	Beginning of a DO
41	CSF	Conditional statement
42	RPF	Replacement statement
43	CAF	CALL statement
44	ISF	IF statement
45	GTF	GO TO statement
46	CNF	CONTINUE statement
47	IDF	Ignore Vector Dependency flag

If the header is for an entry (ENF set), bits 40-47 give the number of arguments associated with this entry.

BWI	50-63	Backward offset relative to the location of this header to previous statement header in TGB
FWI	64-77	Forward offset relative to the location of this header to next statement header in TGB

Following the header word for a statement is a 1-word entry for each of the elements in the statement. This entry can be a tag, an operator, or a separator. A tag is derived from the TBT entry for the corresponding symbol. It contains an index to the TBT entry. Operators and separators are translated to 6-bit codes that reflect processing precedence.

Tag Buffer flags are as follows:



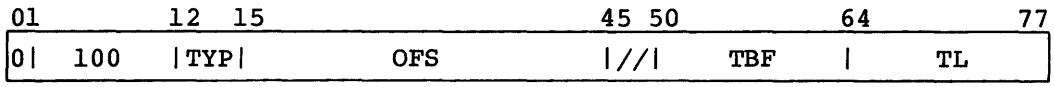
Field	Bit	Description
Flags:	50-63	
DAF	50	Dummy Argument flag
EQF	51	Equivalence flag If DAF and EQF are set, the tag is a pointer reference
FNF	52, 53-55	Function flag; if flag is set, the symbol name is a function and bits 53-55 are as follows. 53 Call-by-value flag (CBV) 54 Function/subroutine Call flag (FSC) 55 Single/multiple Result Function flag (SMR) If flag is clear and bits 53-55 are zeros, then the symbol name is a simple variable. Otherwise, the symbol name is an array; bit 52 is clear and bits 53-55 contain the number of dimensions in the array (up to 7). If bits 51 and 52 are set, the function has side effects and cannot be optimized.
INF	56	Internal Statement Function flag or
RDF	56	Defined flag; set if variable is defined (assigned a value). This bit is set on a block-by-block basis.
IVF	57	Variant/invariant flag; set if variable is a variant within the block. This bit is set on a block-by-block basis.
CIF	60	Constant Increment Variable (CIV) flag
VAF	61	Vector Array flag or Function flag; set if this array reference or function call can be

Field	Bit	Description
VAF (continued)		vectorized. Vectorization is possible if the subscripting is acceptable, if the vector array or function is in a vectorizable form, or if it is a known vector library routine with a vector argument.
SAF	62	Intrinsic function special processing bit; bit is checked only in function headers with VAF set. If SAF is also set, a call is made to SPFH from CBLK and to SPFR to handle special processing for intrinsic functions such as SHIFT and CSMG.
		or
KSF	62	Known Sign flag; set if the sign of a number is known. Used for constant tags (600) only.
MAF	63	Sign bit, if known (if bit 62 is set)
		or
SCF	63	Subsequent ambiguous reference
TL	64-77	Type and length

Pseudo entry

A pseudo tag is used to represent a temporary value which is not to be allocated a memory location. The pseudo tag (100) is used only in TGB and TBG. Fields conform to function and variable. Pseudo entry for a function is only used for RETURN statements, while that for a variable is used throughout.

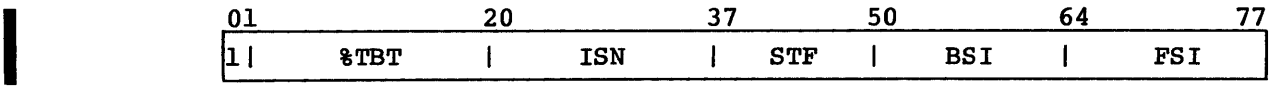
Format:



Field	Bits	Description
TAG	1-11	Tag; 100 (constant tag).
TYP	12-14	Constant tag subtype
OFS	15-44	Offset
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Statement number definition entry

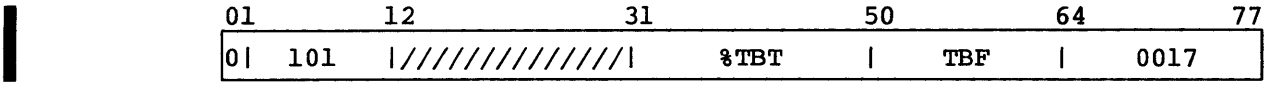
Format:



Field	Bits	Description
%TBT	1-17	Index into TBT
ISN	20-36	Internal sequence number
STF	37-47	Statement Type flags
BSI	50-63	Backward statement index
FSI	64-77	Forward statement index

Statement number reference entry

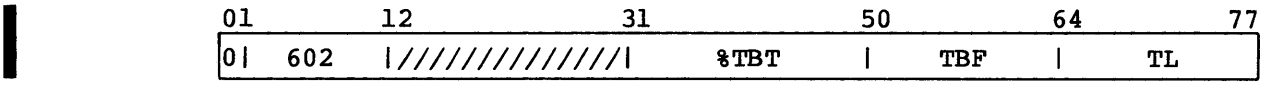
Format:



Field	Bits	Description
TAG	1-11	Tag; 101 (statement number tag).
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length; 0017.

Format number reference entry

Format:

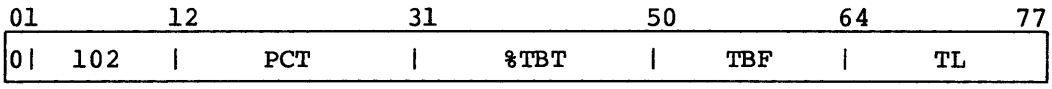


Field	Bits	Description
TAG	1-11	Tag; 602 (program block, positive relocation tag).
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length; 2017 for 24-bit ASCII value, 2077 for 64-bit ASCII value.

External function entry

(An external intrinsic function has an entry in TBL and TBM.)

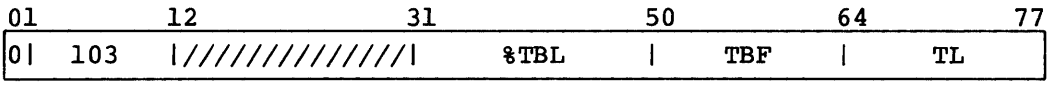
Format:



Field	Bits	Description
TAG	1-11	Tag; 102 (external function tag).
PCT	12-30	Parameter count; number of arguments to the function (taken from TBM).
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Intrinsic function entry

Format:

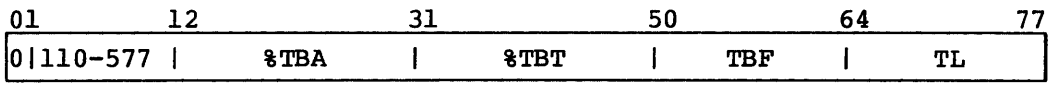


Field	Bits	Description
TAG	1-11	Tag; 103 (intrinsic function tag).

Field	Bits	Description
%TBL	31-47	Index into TBL entry
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Dummy argument entry (Pass 1)

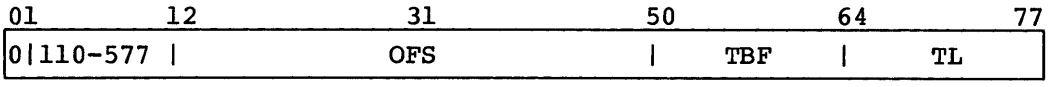
Format:



Field	Bits	Description
TAG	1-11	Tag; a number from 110 to 577 (dummy argument tag).
%TBA	12-30	Index into TBA (for array references)
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Dummy argument entry (Pass 2)

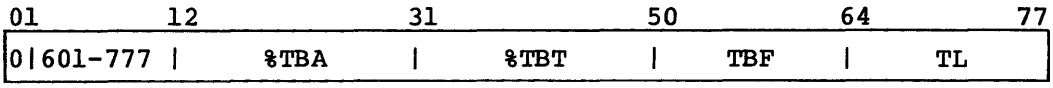
Format:



Field	Bits	Description
TAG	1-11	Tag; a number from 110 to 577 (dummy argument tag).
OFS	12-47	Offset in block
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Variable in program and common block (Pass 1)

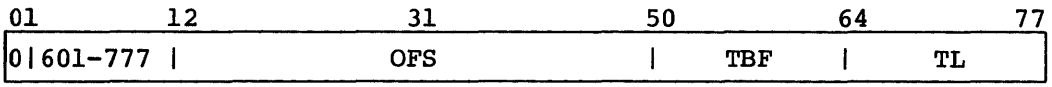
Format:



Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag 607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
%TBA	12-30	Index into TBA (for array references)
%TBT	31-47	Index into TBT
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Variable in program and common block (Pass 2)

Format:



Field	Bits	Description
TAG	1-11	Tag; one of the following: 601 Offset from B01 (dummy argument addresses) 602 Program block, positive relocation 603 (#TB) Temporary block tag 604 (#CL) Argument list block tag 605 (#ST) Offset from B03; stack tag. 606 (#RG) B/T register tag

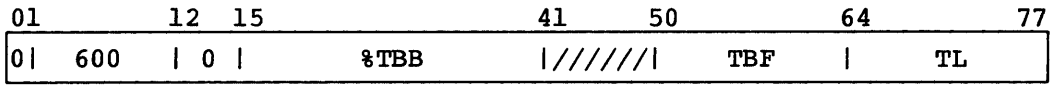
Field	Bits	Description
TAG (continued)		607 (#DA) Data block tag 610-777 Common block tag (assigned in ascending order)
OFS	12-47	Offset in block
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Constant entry

An entry is made in TGB for each constant encountered. Four constant tag subtypes are available. Their formats follow.

A subtype 0 constant tag gives TBB entry information.

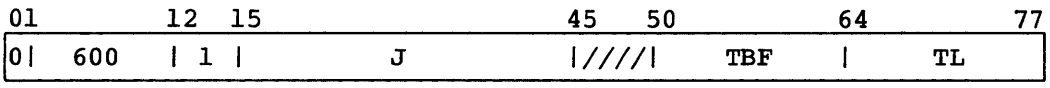
Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 0 (constant in TBB).
%TBB	15-41	Index into TBB
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

A subtype 1 constant tag signals a machine-generated constant. (The 071 machine instruction allows generation of a number of different constants.)

Format:



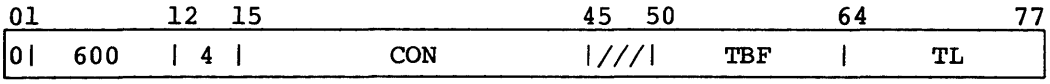
Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 1 (machine-generated constant).
J	15-44	J holds a value that corresponds to the opcode's J value as follows:

Opcode	Constant	J value
071i4x	0.5	4
071i5x	1.0	5
071i6x	2.0	6
071i7x	4.0	7

TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

A subtype 4 constant tag is used for a 22-bit immediate constant.

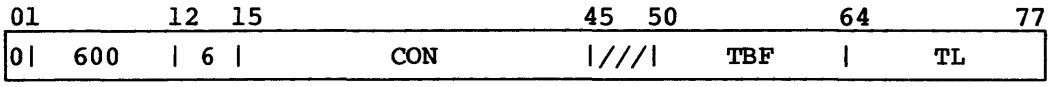
Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 4 (immediate).
CON	15-44	22-bit immediate constant (preceded by 2 sign bits)
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

A subtype 6 constant tag contains a 22-bit intermediate shifted constant. The 22-bit constant is shifted left 51 places. This subtype is used to generate floating-point constants.

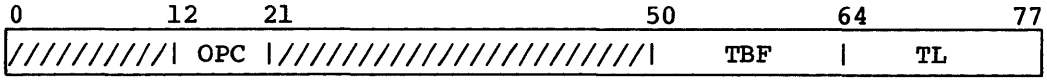
Format:



Field	Bits	Description
TAG	1-11	Tag; 600 (constant tag).
SUB	12-14	Constant tag subtype; 6 (shifted constant).
CON	15-44	22-bit intermediate shifted constant
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

Operator entry

Format:



Field	Bits	Description
OPC	12-20	TGB representation of operator or separator in 6-bit code form. OPC codes are given in table 4.TGB-1.
TBF	50-63	Tag Buffer flags
TL	64-77	Type and length

The following bits are used when the OPC field is 53, 54, or 55.

Field	Bits	Description
	20-37	Size of the parenthesis group if OPC is 53 or 54
	34-47	%TBW of subscript increment if OPC is 53 or 54 Number of arguments if OPC is 53 or 54 Number of subscripts if OPC is 53, 54, or 55 Argument number if OPC is 53, 54, or 55

SUBROUTINES 5

This section includes brief descriptions of the major subroutines that comprise the CFT compiler. These routines are listed in alphabetical order according to routine name.

ROUTINE: ABLK - Analyze block

PASS: 2

DESCRIPTION: ABLK is the main driver for Pass 2. It divides TBG into segments, called blocks, which are then compiled into code, one at a time.

During Pass 1, all extraneous statement numbers are deactivated. Thus, at the beginning of Pass 2, each statement number in TBG is a target.

Each time ABLK is called, it searches through TBG to find the next block boundary. A block boundary is an ENTRY statement, a loop begin, or a statement referenced from outside the block.

ABLK does the following:

- Builds TBZ, the list of variables defined within the block
- Sets the variant bit in all references to variant elements, including variables used in EQUIVALENCE statements and common or dummy arguments (if there is an external reference to them)
- Locates all constant increment variable (CIV) variables and creates a TBZ entry for each CIV

The ABLK sequence is:

1. ABLK
2. Analyze dependency conditions (ADEP)
3. Promote constants from subscripts (PCON)
4. Set Vector Array flag in all probable scalar temporary vectors (SVEC)
5. Examine array or function references (EAFR)
6. Vector loop control (VCTL)
7. Compile block (CBLK)

ROUTINE: BDST - BLOCK DATA statement processor

PASS: 1

DESCRIPTION: Routine BDST processes BLOCK DATA statements and enters SRST at SR04.

ROUTINE: BFST - BUFFER IN and BUFFER OUT statement processor

PASS: 1

DESCRIPTION: This routine processes BUFFER IN and BUFFER OUT statements. It also does some syntax checking and branches to routine IOST (at IO66).

ROUTINE: BGIN - Begin compilation

PASS: 1

DESCRIPTION: Routine BGIN initializes CFT and calls the control statement cracking routine, CARD.

The code beginning at BG10 performs partial initialization at the start of Pass 1 for each block compiled.

ROUTINE: BKST - BACKSPACE statement processor

PASS: 1

DESCRIPTION: Routine BKST is the BACKSPACE statement processor. Upon recognition of 'BACKSPACE', control is passed to RW01 in RWST for further processing.

ROUTINE:	BLCN - Blank count
PASS:	1,2
DESCRIPTION:	Routine BLCN returns the number of bits occupied by the leading blanks in the right-justified, blank-filled, 8-bit ASCII symbol in S1. The caller can use this bit count to left-justify the symbol.

ROUTINE:	BLFL - Blank fill a word
PASS:	1,2
DESCRIPTION:	This routine blank fills an ASCII word in register S1.

ROUTINE:	BOFG - Check branches out of loop
PASS:	2
DESCRIPTION:	BOFG checks all branches out of a potential vector loop for being candidates for vector search. If a vector search does not occur, the vector loop flag is cleared. If a vector search occurs, the address is entered into the search table.

ROUTINE:	BTD - Convert binary value to ASCII decimal value
PASS:	1,2
DESCRIPTION:	BTD converts a binary value (S1) to the ASCII representation of its decimal value (returned in S1). The contents of A0-A7 and S7 are saved, the external library routine $BTD does the conversion, and A0-A7 and S7 are restored.

ROUTINE:	CADR - Compile address
PASS:	2
DESCRIPTION:	CADR compiles the address of a tag. It compiles the code needed to load an address into an A register.

DESCRIPTION: (continued)

Certain conditions must be met before optimization occurs. Optimization must first be enabled by specifying OPT=FULLIFCON or OPT=PARTIALIFCON on the CFT control card. (Optimization is disabled by specifying OPT=NOIFCON on the CFT control card (default) or using the CDIR$ NOIFCON compiler directive.) If the optimization level is partial-IF conversion (OPT=PARTIALIFCON), the replacement expression cannot involve division or an external function reference. The last requirement for optimization is that the type of the replacement variable must be integer, logical, or real.

ROUTINE: CCTB - Convert character constant operand

PASS: 2

DESCRIPTION: CCTB converts a character constant operand to a Boolean operand when the character constant operand is used as an arithmetic operator operand.

ROUTINE: CDIR - Compiler directive processor

PASS: 1

DESCRIPTION: CDIR processes the CDIR$ directives during Pass 1.

ROUTINE: CDPR - Compiler directive processor

PASS: 2

DESCRIPTION: CDPR processes the ALIGN, BL, BLOCK, CODE, CVL, FASTMD, NOBL, NOCODE, NOCVL, NODOREP, NOIFCON, NORECURRENCE, NOVECTOR, RESUMEDOREP, RESUMEIFCON, ROLL, SAFEIF, SLOWMD, and UNSAFEIF compiler directives during Pass 2.

ROUTINE: CDSP - Code and data separation

PASS: 1

DESCRIPTION: CDSP separates code and data. CDSP operates at the end of pass 1, just prior to analysis of branch statements (ABRA). CDSP performs the following functions:

- Converts 602 program block tags to 607 data block tags in TBT and TBG to allow addressing of up to 16 million words in EMA mode, and to ensure that data references are to the data block and not to the program block throughout code generation
- Puts constants, DATA initialization, and local storage in the data block by adjusting loader table TXT header block indices in TBB
- Puts any secondary entries and NAMELIST tables in the data block by adjusting loader table TXT header block indices in TBB
- Puts any externals passed as parameters in the data block by adjusting loader table XRT header block indices in TBE
- Puts any BRT loader table entries in the data block by adjusting BRT header block indices in TBR and creates a BRT header for program block entries. TBR entries at this point are for secondary entries and/or NAMELIST table.
- Puts pointers in the data block by converting 602 tags to 607 tags in TBC
- Changes the program block tag (602) in T.PB to the block data tag (607)
- Calculates the size of the data block for the loader table PDT and checks for data block overflow, and sets the current parcel address (T.PA) and the program base address (T.PBS) to zero.

ROUTINE: CE*xx* - Check EQUIVALENCE overlap

PASS: 2

DESCRIPTION: These routines check for EQUIVALENCE overlap, looking for EQUIVALENCE dependencies between two tags.

CE*xx* checks all references against the definitions in TBZ. CEOV selects exact matches for the references in TBZ and then begins an analysis of the subscripts. Because all constants are packed together at the end of the subscript expression during Pass 1, all scans proceed from right to left. If the constant add-ins do not match, a flag is set.

ROUTINE: CEXP - Constant expression evaluation

PASS: 1,2

DESCRIPTION: CEXP examines the tag buffer for a simple constant expression inside parentheses. If the tag buffer contains ($constant_1$ *operator* $constant_2$) where the constant tags are of the form 06004... and the operator must be KAP, KAS, or KAD, this expression is evaluated to a single constant and stored back into the tag buffer.

ROUTINE: CFBI - Correct forward and backward indices

PASS: 1,2

DESCRIPTION: CFBI examines a section of the tag buffer to ensure that all statement header forward and backward indices are correct.

ROUTINE: DD*xx* - Implied DO processor

PASS: 1

DESCRIPTION: These routines process implied DOs during the second section of DATA statement processing. They calculate and maintain trip counts and increment the DO variable pseudo register in TBY.

ROUTINE: DETB - Build Debug Symbol Table

PASS: 2

DESCRIPTION: DETB builds the Debug Symbol Table for the loader.

ROUTINE: DLTB - Initialize and print DO-loop table

PASS: END processing for Pass 1; 2

DESCRIPTION: DLTB inserts and retrieves information about a DO loop into/from TBD and prints the table of loops encountered.

ROUTINE: DMST - DIMENSION statement processor

PASS: 1

DESCRIPTION: DMST processes DIMENSION statements and then sets up for routine DCLR.

ROUTINE: DORP - DO-loop replacement

PASS: 1

DESCRIPTION: DORP checks for the replacement of a 1-line DO-loop with a call to a $SCILIB routine which performs the same operation more efficiently. When DORP is called from STTR, DOST expanded the DO-loop preamble and body in TBG. DORP parses the TGB for an equation match and evaluates the equation to determine if a $SCILIB routine can do the same operation. If a $SCILIB routine is chosen, the DO-loop preamble is rewritten and the $SCILIB call macro is expanded by OP81. Control returns to STTR where the DO terminator label is checked. If the 1-line DO-loop cannot be replaced, control returns to STTR before TGB is altered and DO-loop termination continues normally.

ROUTINE: DOST - DO statement processor

PASS: 1

DESCRIPTION: DOST processes DO statements and does the following:

- Checks syntax
- Calls AT44 to process the DO list
- Builds the termination package in TBR
- Calls OP81 to expand the Pass 1 macro for a DO statement
- Processes implied DOs in I/O statements

DO22 is called from STTR to expand TBR into the DO termination macro.

ROUTINE: DOUN - DO-loop unrolling

PASS: 1

DESCRIPTION: DOUN attempts to unroll an inner DO-loop with known iteration counts. The following conditions must be met before a DO-loop is unrolled.

ROUTINE: ECNT - Enter conjunctive term

PASS: 2

DESCRIPTION: ECNT copies the term from TBDT to TBCT if it is not already in TBCT.

ROUTINE: ECNU - Enter simple term

PASS: 2

DESCRIPTION: ECNT forms and enters a simple relational term into TBCT.

ROUTINE: ECST - ENCODE statement processor

PASS: 1

DESCRIPTION: This routine processes ENCODE statements and then sets up for routine IOST.

ROUTINE: EDJT - Enter disjunctive term

PASS: 2

DESCRIPTION: EDJT forms and adds one condition for safe vectorization of a dependency to the conditions already found. Terms are simplified and duplicates are removed before entry.

ROUTINE: EDJU - Enter test for differing CIVs

PASS: 2

DESCRIPTION: EDJU forms and adds one condition for safe vectorization of subscripts with CIVs which do not match.

ROUTINE: EFST - ENDFILE statement processor

PASS: 1

DESCRIPTION: EFST processes ENDFILE statements. Upon recognition of 'ENDFILE', control is passed to RW01 in RWST for further processing.

ROUTINE: EHOL - Enter Hollerith string

PASS: 1

DESCRIPTION: EHOL enters a Hollerith/character string in TBB (or TBE for a DATA statement). The string can be longer than one word. It is packed, eight characters per word, and any trailing H, L, or R is processed. Character or Hollerith constants with an H suffix are padded from the right with blanks to a word boundary. A zero word in TBB terminates a character or Hollerith string that appears in an argument list.

ROUTINE: EIDL - Examine implied DO-loop list

PASS: 1

DESCRIPTION: Routine EIDL examines the implied DO-loop list. It examines subscript expressions in DATA statements to see if subscripting is linear; if it is, the implied DO-loop is compressed into a loader DUP Table.

ROUTINE: ELWD - Enter last word

PASS: 1,2

DESCRIPTION: ELWD is the main table manager. It is used in creating table entries for all sequential tables. ELWD enters a value from register S4 into the last word of a table (KTx in A1).

When making an entry, ELWD detects whether the entry would cause the table to overflow. If this is the case, ELWD calls routine MTAB to expand the table. If there is a memory move, all the table pointer words (in V7) of the moved tables are adjusted.

ROUTINE: EMPR - Error message processor

PASS: 1,2

DESCRIPTION: EMPR is the error message processor. It prints out any error messages, with the message number as its argument.

Normally, for a fatal error during Pass 1, EMPR exits to STTR; for a fatal Pass 2 error, it exits to ABLK. It can optionally return to the caller on fatal errors.

NOVECTOR messages are issued by LMRK, not by EMPR.

EMPR always returns for nonfatal errors. Before returning, EMPR restores all registers.

ROUTINE: ENST - END statement processor

PASS: 1

DESCRIPTION: ENST terminates Pass 1 and processes FLOWEXIT, $END, or implied RETURN.

DESCRIPTION: (continued)

EN01 begins termination of Pass 1 and sets up for Pass 2. It performs the following functions:

- Allocates actual addresses for blocks
- Processes DATA statements
- Initializes the loader tables
- Links statement number references in TGB
- Counts the number of declared task common blocks and prepares the required number of reserved B registers
- Incorporates TBQ into the entries in TGB
- Moves TGB to TBG; as TGB is moved, the tags are transformed to contain actual offset addresses rather than pointers into TBT.
- If EMA mode is enabled, converts 602 program block tags to 607 data block tags to separate code from data and to allow reference to up to 16 million words of local data.

If fatal errors have occurred, the last three steps are skipped. After ABLK has compiled the last block, the LDR tables are closed and written to the binary file. The symbol table is printed if specified and control then transfers to BG10.

When RNXT detects end of file, control transfers to EN78. CFT issues logfile messages and exits.

ROUTINE: EQST - EQUIVALENCE statement processor

PASS: 1

DESCRIPTION: Routine EQST processes EQUIVALENCE statements. EQUIVALENCE processing is done in three steps. First, as EQUIVALENCE statements are encountered they are packed up, 8 characters per word, and moved from CHB to TBR. Control then returns to RNXT. Further processing is postponed until all other declaratives are processed, when the first executable statement or statement function is found. Finally, storage is assigned.

When the first nondeclarative is encountered, control transfers to EQ10. At this point, TBP contains an entry for each common block entity consisting of the %TBT for that entity. EQ replaces each common block entry in TBP with an entry of the

DESCRIPTION: (continued)

form described in 4.TBP. TBH is updated reflecting each common block length. The corresponding TBT entry for each TBP entry is updated with %TBP for that entity.

Next, the packed equivalenced entities are extracted one by one from TBR. A corresponding TBP entry is created for each equivalenced entity. This ends the actions occurring at EQ10.

At the end of Pass 1 (at EN10), a pass is made through TBP to perform two functions for each equivalenced entity: (1) Detect a common block lengthened through an equivalence, in which case, TBH is updated; (2) Assign storage to equivalenced entities that are part of the static or stack blocks. The information in TBP is saved until the end of Pass 1 when all storage allocations are done.

At the end of Pass 1, all variable tags within a block are represented by a base tag and an offset field. This means that during Pass 2, CFT can accurately determine whether there is a conflict between EQUIVALENCE variables.

ROUTINE: ERTX - Invalidate old TBX entries

PASS: 2

DESCRIPTION: ERTX marks TBX entries which are no longer valid due to a transfer of control.

ROUTINE: ESBK - Enter new sub-block

PASS: 2

DESCRIPTION: ESBK enters the statement description into TBBK.

ROUTINE: ESNL - Enter statement number reference

PASS: 2

DESCRIPTION: ESNL checks the the statement number reference for a previous definition in the block. If a previous definition is found, the block is terminated and loop is compiled. If a previous definition is not found, the reference is entered into TBBK.

ROUTINE: ESTB - Enter Symbol Table

PASS: 1

DESCRIPTION: Routine ESTB makes an entry in the Symbol Table (TBS). ESTB is called only after SSTB has determined that the symbol name in question is not yet in TBS. SSTB makes an alphabetical search of TBS and locates the slot that the new symbol name should occupy. ESTB then makes the entry at that point in TBS.

ESTB checks to make sure that there is space in TBS for each new entry. If ESTB finds that more space is required, it calls for a memory move upward or downward, depending on whether the new symbol belongs in the upper or lower half of TBS. (Refer to the section on Table Management in this publication for a detailed description.) Then ESTB completes making the new entry.

ROUTINE: ETBX - Make TBX entry

PASS: 2

DESCRIPTION: ETBX generates the TBX entry and a parallel TBXX entry with a Valid or Invalid flag.

ROUTINE: FRTG - Locate argument tag

PASS: 1

DESCRIPTION: FRTG locates the argument tag if it is a single tag and returns the argument tag to the caller.

ROUTINE: FSHD - Find statement header

PASS: 2

DESCRIPTION: FSHD is given the address of a TBG entry and finds the statement header of the entry.

ROUTINE: FSTK - Force compiler-generated variables onto stack

PASS: 1

DESCRIPTION: FSTK is called in stack mode to force compiler-generated variables onto the run time stack (see Appendix D for the stack frame format). This procedure is done by inserting the stack block tag into each static-tagged TBT entry without a corresponding TBS entry (that is, it moves anonymous variables to the stack).

Two classes of compiler-generated variables are not forced by FSTK. The first class occurs when the space into which argument lists are copied for multiple-entry routines is assigned to the stack by EN10. The second class is function result tags made into special cases by EN84.

ROUTINE: FSUB - Find substring

PASS: 2

DESCRIPTION: FSUB locates the substring for the character operand.

ROUTINE: GBAT - Generate B to A register transfer instruction

PASS: 2

DESCRIPTION: GBAT adds a 024*ijk* (A*i* B*jk*) instruction to PIB. TBX is searched for a matching entry to the B pseudo register in S1. If a matching entry is found, the pseudo register is returned. If a matching entry is not found, a register tag entry is made and entered into TBX. An 024*ijk* (A*i* B*jk*) instruction is generated using a new pseudo register. The resulting A pseudo register is then returned in S1.

ROUTINE: GCBS - Get common or data block base pseudo register

PASS: 2

DESCRIPTION: GCBC adds instructions to PIB to address the base (0 word) of a common block or data block. GCBS is used by Pass 2 addressing routines to address extended memory common blocks and data blocks.

S4 contains the tag buffer entry. The offset in S4 is zeroed to create a new table entry. TBX is searched for a matching entry. If a match is found, the matching pseudo register is returned. If a matching entry is not found, a new pseudo register is assigned to S4 and the new entry is added to TBX. Using the new TBX entry, a 020 load instruction is generated and the new pseudo register is returned. If extended memory addressing is used, all 020 loads with common block or data block relocation will be changed to an extended memory 01*h* 1*ijkm* load. The *ijkm* field is 24 bits long to allow for very large addresses.

DESCRIPTION: (continued)

If the offset in S4 is positive and longer than 22 bits, the offset is left in the new TBX entry and the original TBX entry is zeroed. This allows the 01*h* 1*ijkm* load to be generated with the very large offset.

ROUTINE: GCRF - Generate code-and-result tag

PASS: 2

DESCRIPTION: GCRF generates the code-and-result tag for a 22-bit constant deferred in Pass 2.

ROUTINE: GDEX - Get dimension extent

PASS: 1

DESCRIPTION: GDEX finds the permissible range of a subscript in an array in order to estimate the maximum likely trip count for a loop.

ROUTINE: GDLB - Get dimension lower bound

PASS: 1

DESCRIPTION: GDLB determines the lower bound of an array dimension in order to check for negative values being used in subscript calculations.

ROUTINE: GIXA - Generate index address

PASS: 2

DESCRIPTION: GIXA generates pseudo registers for an address on a stack and computes the address.

ROUTINE:	GLBD - Get label definition
PASS:	2
DESCRIPTION:	GLBD generates an internal label for use in Pass 2 code generation.

ROUTINE:	GMEM - Get memory
PASS:	1,2
DESCRIPTION:	Routine GMEM gets more memory from the operating system. COS inserts the memory between the tables and the I/O buffers. GMEM moves the tables and adjusts the table pointer words in V7.

ROUTINE:	GRLD - Generate reductions and load secondary registers
PASS:	2
DESCRIPTION:	GRLD initializes vector reductions, performs the initial load of items kept in secondary registers for the duration of a loop, and sets up scalar recurrence variables.

ROUTINE:	GRRL - Generate reload of item from secondary register
PASS:	2
DESCRIPTION:	GRRL reloads an item that was assigned to a secondary register during the execution of a loop.

ROUTINE: GRST - Generate reset of secondary register

PASS: 2

DESCRIPTION: GRST resets a secondary register for a recurrence variable.

ROUTINE: GRSV - Generate save of recurrence variable

PASS: 2

DESCRIPTION: GRSV saves the value of a recurrence variable, one of whose stores is conditional.

ROUTINE: GSBS - Get stack base tag

PASS: 2

DESCRIPTION: GSBS finds the pseudo register holding the address of the stack base.

ROUTINE: GTCB - Get task common block base pseudo register

PASS: 2

DESCRIPTION: GTCB adds instructions to PIB to address the base (word 0) of a task common block. GTCB is used by Pass 2 addressing routines to address task common blocks.

DESCRIPTION:
(continued)

The corresponding common block attribute entry (COMTAG) is obtained based on the S4 tag. If the B pseudo register field is nonzero, a call is made to GBAT to generate a B to A register transfer. If the B register field is 0, the base address of the common block must be calculated. A new TBX entry is created by zeroing the offset. TBX is searched for a matching entry. If a match is found, the matching pseudo register is returned. If a matching entry is not found, a new pseudo register is assigned and the new TBX entry is added to TBX. A 100-load instruction is generated and the base pseudo register is returned.

If the S4 offset is positive and longer than 22 bits, a check is made for extended memory addressing. If extended memory addressing is not requested, an error message is issued. If EMA is being used, the offset for the original TBX entry is zeroed and a 020 load instruction is generated (it will become a 01*h ijkm* extended memory load). An A register add is then generated to add the original task common block base PR to the very large offset PR. The pseudo register used as the result of the add is then returned.

ROUTINE: LBLK - Locate sub-block definition

PASS: 2

DESCRIPTION: LBLK locates the TBBK entry and the TBG address.

ROUTINE: LDIV - Integer divide processor

PASS: 2

DESCRIPTION: LDIV generates a function call to process the 64-bit integer divide. LDIV is called by CTRI.

ROUTINE: LGCL - Logical and relational operator processor

PASS: 2

DESCRIPTION: Routine LGCL generates code to process relational and logical operators. It is called by CTRI and it may call CTRI or INFN to generate code.

ROUTINE: LGST - LOGICAL statement processor

PASS: 1

DESCRIPTION: This routine processes LOGICAL statements and then sets up a call to routine DCLR.

ROUTINE: LLIV - Load CIV value

PASS: 2

DESCRIPTION: LLIV loads the value of a constant increment variable (CIV) for a loop.

ROUTINE: LMRK - LOOPMARK processor

PASS: END processing after pass 2

DESCRIPTION: LMRK draws brackets around DO loops in the source code listing and prints messages for loops that did not vectorize. LMRK does not use EMPR to issue messages.

ROUTINE: LSOM - Load store overlap move

PASS: 2

DESCRIPTION: LSOM inserts a vector-to-scalar transfer before a vector instruction which could cause memory overlap on the CRAY X-MP Computer System.

ROUTINE: LSOV - Load/store overlap check

PASS: 2

DESCRIPTION: LSOV checks for vector load/vector store overlaps on the CRAY X-MP Computer System.

ROUTINE: LT*xx* - Loader Table generator

PASS: 2

DESCRIPTION: The LT*xx* routines build the loader tables.

LTST initializes the loader tables called by END at the start of Pass 2. It takes all the instructions from RASN and packs them into the format required by the loader. It adds indicator bits used by the loader to indicate what to relocate, whether an instruction references an external, common block information, and so on.

LTGN builds the loader tables for each block. It is called after routine RASN has assigned registers.

DESCRIPTION: (continued)

LTGN exits to ABLK to fetch the next block. It packs each instruction into TBB. Since the actual target address for jumps may not be known, all jump instructions are linked in TBB. LTGN also builds the External Reference Table (TBE) and the Block Relocation Table (TBR) for variable references and jumps. If the generated code listing is requested, LTGN calls the appropriate OUT*xx* routine to format the output.

LTFU can be called to force pass instructions until a word boundary is reached. It is normally used at the end of a routine or before an entry.

LTND is called at the end of Pass 2 to terminate the loader tables. The actual number of B and T registers needed to be saved/restored is inserted in all of the EXIT/ENTRY sequences. Actual statement label addresses are inserted in jump instructions.

ROUTINE: MAP - Map block names and lengths

PASS: 2

DESCRIPTION: MAP prints the block names and lengths list in the symbol table, if requested.

ROUTINE: MCEX - Special case handling for scheduler

PASS: 2

DESCRIPTION: MCEX expands the instruction sequence after the sequence is scheduled.

ROUTINE: MIAR - Change variables to short integers

PASS: 2

DESCRIPTION: MIAR changes the type of all variables that are used only as subscripts or to update a CIV to short integer, forcing those variables into A registers.

ROUTINE: MMEM - Manage memory

PASS: 1,2

DESCRIPTION: MMEM is a general memory management routine that does memory moves through the vector registers. It can move a block of memory either up or down. The source and destination blocks specified may overlap.

ELWD determines that a move is necessary, MTAB determines which way and where to move, and MMEM does the actual move.

ROUTINE: MSAR - Move S to A register

PASS: 2

DESCRIPTION: MSAR generates the TBW entry to move the S to the A register. MSAR returns the PR found in TBW if one exists; otherwise, the next entry in TBW is returned and generates code moving the S to the A register.

ROUTINE: MSST - Save store preceding vector search branch

PASS: 2

DESCRIPTION: MSST saves stores preceding a vector search branch until after the branch, at which point the vector length (VL) for the store is known.

ROUTINE: MTAB - Move table

PASS: 1,2

DESCRIPTION: MTAB moves tables in memory. It is called by ELWD or ESTB. Each time an addition must be made to a table (a table has overflowed), MTAB looks at adjacent table pointer words in V7 to see what kind of table move is necessary.

DESCRIPTION: (continued)

First, MTAB looks above the current table for any gaps; if there are, tables are moved upward to fill the gaps (rather than moving down and taking more memory). If no gaps exist above this table, MTAB moves the table and all tables below it down 100_8 words of memory and allocates 100_8 words to the table that overflowed. If the downward move gets close to the top of TGB or PIB, GMEM is called to get more memory. MTAB calls MMEM to do the actual move. Table pointers in V7 are updated by MTAB.

ROUTINE: MVOP - Move operands

PASS: 1,2

DESCRIPTION: MVOP moves operands and uses the pointers set up by SOPT.

ROUTINE: NARG - Return number of arguments

PASS: 2

DESCRIPTION: NARG is a function returning the number of arguments in a tag-and-operator-format subroutine or function call.

ROUTINE:	OUT*xx* - Pseudo-CAL output generator
PASS:	2
DESCRIPTION:	OUT*xx* generates the pseudo CAL output. It converts instructions to ASCII and writes them on the output or pseudo CAL file, if requested.

ROUTINE:	OUTBB - Output BLOCK BEGINS
PASS:	2
DESCRIPTION:	This routine writes the "BLOCK BEGINS" message.

ROUTINE:	PAST - PAUSE statement processor
PASS:	1
DESCRIPTION:	Routine PAST processes PAUSE statements. PA02 is the code common for processing STOP and PAUSE. PA10 is common for STOP and PAUSE.

ROUTINE:	PBLK - Select and prepare compilation of next statement
PASS:	2
DESCRIPTION:	PBLK invalidates the TBX entries, if necessary, and selects the next statement to be compiled. At the end of the block, the block is transferred to SKED. Prepare the compressed index sub-blocks and compile any statement number definitions.

ROUTINE: PCIV - Process conditional constant increment variables

PASS: 2

DESCRIPTION: PCIV examines the conditional block for CIV references and moves the load of CIV from the conditional block.

ROUTINE: PCON - Promote constants

PASS: 2

DESCRIPTION: PCON promotes the constants that have been retained thus far intact in subscript expressions. It scans through the subscript references in TBG and collects all possible constants into a single offset term.

ROUTINE: PCST - Process conditional store

PASS: 2

DESCRIPTION: PCST invalidates the TBX entries made unusable by a conditional store.

ROUTINE: PEXP - Process exponent

PASS: 2

DESCRIPTION: PEXP is the exponentiation processor. It is called from CBLK if OLEV finds an exponentiation operation. It searches for the last ** in a sequence (for example, in the expression A**B**C, the B**C would be processed first).

This routine handles special casing of **2, **3, and **4 by calling routine CTRI to do the multiplies. Otherwise, PEXP builds the name of the external routine that does the operation and calls CTRI; CTRI then calls PE90 to generate the actual CALL (through CB32 and CB42A).

ROUTINE: RBRG - Initialize register times

PASS: 2

DESCRIPTION: RBRG assigns initial times to registers.

ROUTINE: RBVT - Schedule store of vector temporary

PASS: 2

DESCRIPTION: RBVT delays the store of a vector temporary variable until after completion of the vector loop.

ROUTINE: RBVU - Issue store of vector temporary

PASS: 2

DESCRIPTION: RBVU stores the value of a vector temporary variable; the store can take place either in or outside of a loop.

ROUTINE: RCCK - Register chain check

PASS: 2

DESCRIPTION: RBRG determines if the result of one operation is needed for the evaluation of a specified pseudo register.

ROUTINE: RDPT - Remove duplicate terms

PASS: 2

DESCRIPTION: RDPT compares two terms and removes duplicate additive expressions from both terms. Duplicate expressions are replaced by nulls.

ROUTINE: RDST - READ statement processor

PASS: 1

DESCRIPTION: RDST processes READ statements. It sets unit to 100 if the READ is in short form and goes to IOST.

ROUTINE: REST - REAL statement processor

PASS: 1

DESCRIPTION: REST processes REAL statements and then sets up for routine DCLR.

ROUTINE: RNXT - Read next statement

PASS: 1

DESCRIPTION: Routine RNXT reads the next statement from the input file, checks for comment and continuation cards, and (if necessary) drives the source output listers.

RNXT places the statement in the Character Buffer (CHB). It skips comment lines and concatenates continuation lines. Columns 1 through 6 and 72 through the end of line are discarded. All blanks are removed from CHB except on FORMAT statements. Hollerith/character text characters are flagged by setting their sign bits. RNXT converts lowercase characters to uppercase.

FORMAT and END statements are special cases. RNXT also checks to ensure all parentheses are matched pairs. See section 2 for a detailed description of RNXT.

End of statement is indicated by a zero word.

ROUTINE: ROSR - Convert an invariant **CIV

PASS: 1

DESCRIPTION: ROSR reduces the strength of an invariant **CIV by converting it to a multiply operation.

ROUTINE: RPST - Replacement statement processor

PASS: 1

DESCRIPTION: This routine processes the replacement statement. It sets up and transfers to AT36 to convert to tag-operator format.

If array bounds checking is in effect and the left-hand side of the replacement statement is an array, RPST sets up for OP02 to do the actual bounds checking.

ROUTINE: RSTB - Restore table pointers

PASS: 1,2

DESCRIPTION: RSTB restores a dynamic table pointer to a previous state given the current and previous pointers.

ROUTINE: RTC - Real-time clock

PASS: End of compilation

DESCRIPTION: This routine calculates compile time and formats the logfile message.

ROUTINE: RTST - RETURN statement processor

PASS: 1

DESCRIPTION: RTST processes the RETURN statement.

ROUTINE: RVLR - Restore value to VL register

PASS: 2

DESCRIPTION: RVLR restores the value of the VL register.

ROUTINE: RWST - REWIND statement processor

PASS: 1

DESCRIPTION: This routine processes the REWIND statement. RW01 is common for REWIND, ENDFILE, and BACKSPACE. Subroutines at IO10 and IO20 are called and control is passed to IO67 for final processing.

ROUTINE: SYMADD - Add symbol to cross-reference

PASS: 1

DESCRIPTION: SYMADD updates cross-reference tables TBU and TBV.

ROUTINE: SYTB - Print Symbol Table

PASS: 2

DESCRIPTION: SYTB prints out the Symbol Table in either the short form or the full cross-reference form, if requested.

ROUTINE: TFBK - Transfer between sub-blocks in a conditional loop

PASS: 2

DESCRIPTION: TFBK controls compilations of various sub-blocks in a conditional vector loop. TFBK inserts a transfer around a block and the definitions for the generated statement numbers of the loop. TFBK replaces the statement number definitions in a second block with the generated statement numbers.

ROUTINE: TPRU - Tally PR usage

PASS: 2

DESCRIPTION: TPRU makes a list of PRs to be used after the end of the instruction group.

ROUTINE: TRAN - Do type conversion

PASS: 2

DESCRIPTION: This routine compiles code for type conversion. If the two operands are not the same type, TRAN generates code converting the operand of lowest type to match the code of the higher type operand.

ROUTINE: TRUN - Truncate after each floating-point operation

PASS: 2

DESCRIPTION: TRUN compiles code to do truncation after each floating-point operation, if TRUNC=*nn* is specified.

ROUTINE: UCIV - Udpate CIV value

PASS: 2

DESCRIPTION: UCIV computes the value of a constant increment variable (CIV) for the current iteration of a loop.

ROUTINE: UDCI - Update conditional CIV value

PASS: 2

DESCRIPTION: UDCI computes the value of a constant increment variable (CIV) defined in prior conditional code.

ROUTINE: VCTL - Vector loop control

PASS: 2

DESCRIPTION: This routine provides vector loop control.

ROUTINE: VE*xx* - VECTOR/NOVECTOR directive processor

PASS: 1

DESCRIPTION: VE00 processes the VECTOR compiler directive. VE01 processes the NOVECTOR directive.

ROUTINE: VLAN - Vector loop analysis

PASS: 2

DESCRIPTION: VLAN analyzes the loop for vector hazards such as vector temporaries defined in conditional code, transfers out of the loop, and dependencies.

ROUTINE: WRST - WRITE statement processor

PASS: 1

DESCRIPTION: This routine processes WRITE statements. It sets unit to 101 for the short form of WRITE and branches to IOST.

ROUTINE: XC00 - Execute code

PASS: 1

DESCRIPTION: This routine interpretively executes the code compiled by CX00 for constant expressions.

ROUTINE:	XX00 - Set of interpreters for instructions compiled by CX00
PASS:	1
DESCRIPTION:	This routine consists of a set of interpreters for the instructions compiled by CX00. It is driven by XC00.

ROUTINE:	ZMEM - Clear a block of memory
PASS:	1,2
DESCRIPTION:	ZMEM clears a block of memory. ZMEM is called to clear the tag buffer between compilation units and clear additional tag buffer space when the tag buffer grows. ZMEM is also called to clear scratch tag buffer space after restructuring IF statements.

CFT I/O 6

CFT converts a user program written in FORTRAN to the binary machine language of the CRAY-1 and CRAY X-MP Computer Systems. Under COS, the compiler is loaded and begins processing when a CFT control statement is encountered in a user job deck. Under UNICOS, the compiler is loaded and begins processing when a CFT command line is received.

CFT requires two types of input: the user program to be compiled and a user control statement or command that gives instructions for controlling compilation. The output provided by CFT includes the user's compiled FORTRAN program in relocatable binary and a printable record of the compilation.

Figure 6-1 illustrates the I/O datasets used by CFT during compilation under COS. The dataset names $IN, $OUT, and $BLD are defaults; different dataset names are used if they are specified in the CFT control statement.

Figure 6-2 illustrates the I/O files used by CFT during compilation under UNICOS. The file names *filename*.f, *filename*.l, and *filename*.o are defaults; different file names are used if they are specified in the CFT command.

6.1 INPUT TO CFT

A user's FORTRAN program submitted to CFT must comply with certain program specifications. The syntactic and notational guidelines outlined in the CRAY-1 FORTRAN (CFT) Reference Manual must be observed.

A user may select compiler options through the CFT control statement or command and through use of compiler directives. (Refer to the CRAY-1 FORTRAN (CFT) Reference Manual for descriptions of both the CFT statement and CFT directives.)

6.2 OUTPUT FROM CFT

The output obtained from CFT is dependent on the options selected by the user. The principal output obtained is the compiled FORTRAN program in relocatable form. Optional output consists of a printable record of the compilation, including the FORTRAN source code, the assembly language equivalent generated by CFT, the cross-reference lists, the Debug Symbol Table, and more.

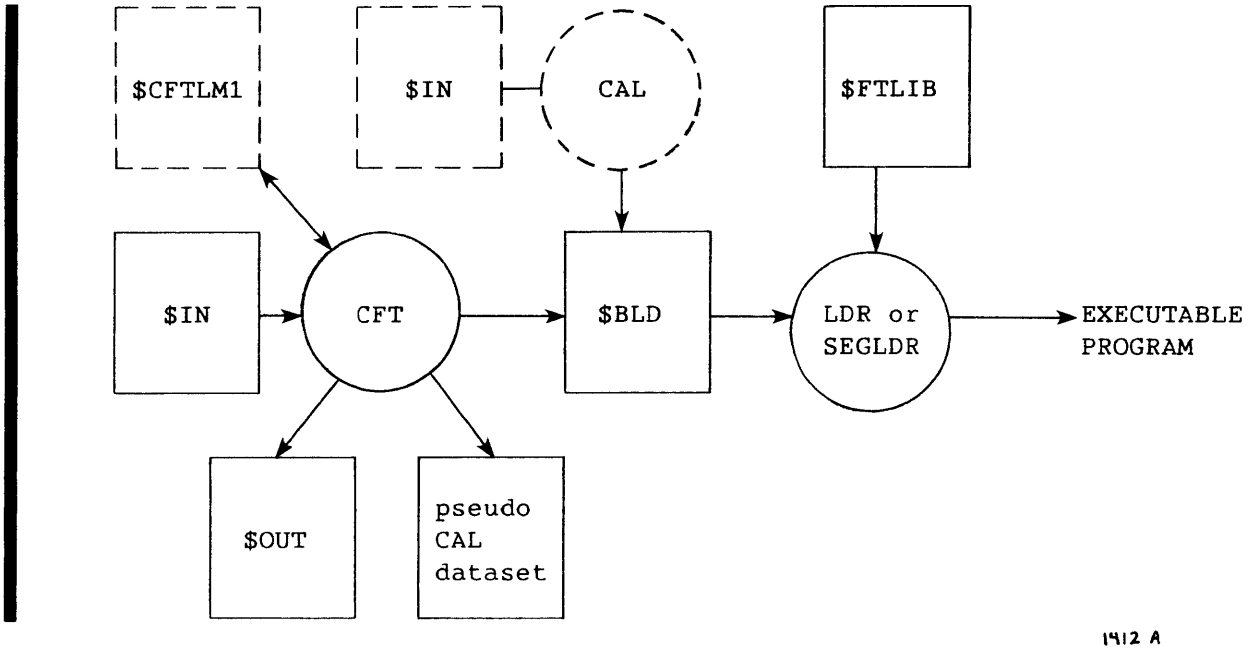


Figure 6-1. I/O Datasets Used During Compilation Under COS

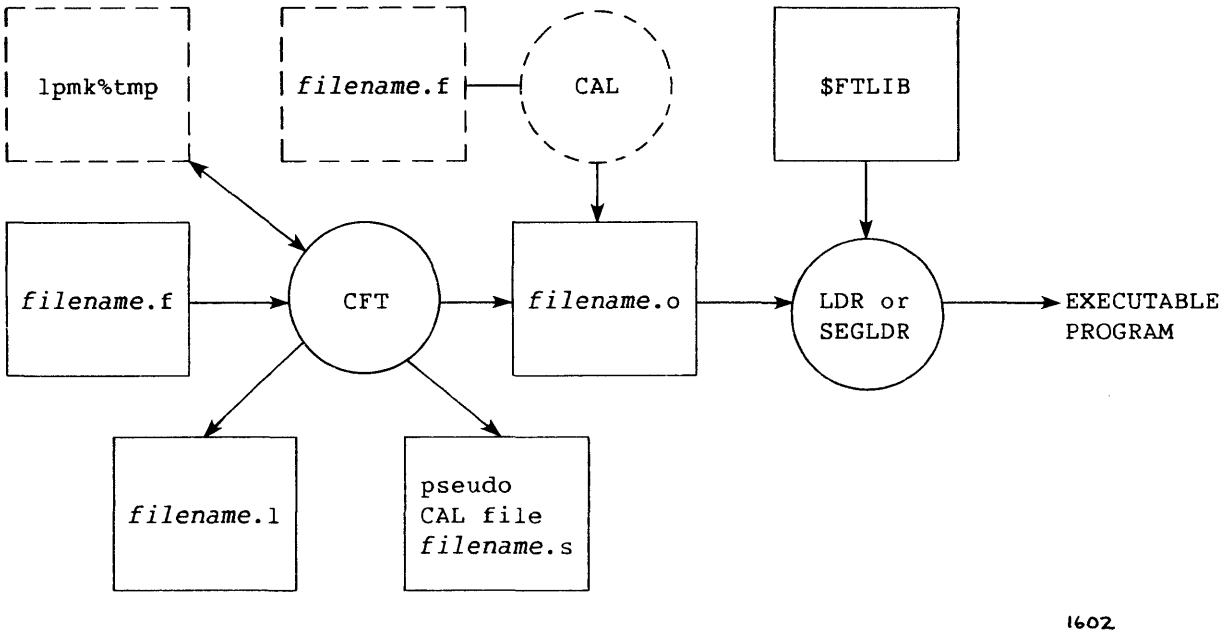


Figure 6-2. I/O Files Used During Compilation Under UNICOS

6.3 I/O DATASETS/FILES

As many as five I/O datasets or files are in use during compilation. These are:

- The text input, default $IN under COS, default *filename*.f under UNICOS
- The source output listing, default $OUT under COS, default *filename*.l under UNICOS
- The binary load-and-go module, default $BLD under COS, default *filename*.o under UNICOS
- The pseudo-CAL output (if requested by the user), default *filename*.s under UNICOS
- A scratch dataset or file used by the LOOPMARK utility (if requested by the user), default $CFTLM1 under COS, default lpmk%tmp under UNICOS

After cracking the CFT control statement or command, CFT opens the datasets or files it will need, using system default parameters. If the datasets or files are already open, the system ignores the redundant OPEN request.

The compiler references a set of library routines to aid the user in manipulating dataset or file buffers. These logical I/O routines do all the driving of the buffers, make the system calls to fill the buffers, and keep track of the IN and OUT pointers. The I/O routines read or write strings of words or characters and make the system calls that allow physical I/O to be transparent to the user.

CFT first writes out the generated code in binary to a dataset or file. This is used as input to the loader LDR or SEGLDR. Using the FORTRAN support library $FTLIB, the loader produces an executable program version of the information in the binary file.

CFT also produces a source listing containing a copy of the source input, compiler messages, and other information supplied by the compiler. Optionally, the user can also obtain a pseudo-CAL listing of the code in a form almost completely acceptable to the assembler. This allows the user to manually optimize portions of the code using CAL.

CFT uses the standard $SYSLIB routines $RCW and $WCW to perform all I/O.

DEBUGGING AIDS C

The organization of CFT generally allows errors to be isolated. This appendix gives some information and methods that may be useful in debugging code.

Because CFT is almost completely reinitialized at the start of each program unit compilation, it is highly unlikely that a compilation bug in one routine was caused by compiler failure in a previous routine.

During compilation, CFT makes two passes through the source code.

PASS 1

Pass 1 reads the input deck, translating it into an internal notation, and then produces the source listing. Any syntax errors detected are listed immediately after the line in error. Generally, Pass 1 bugs can be reproduced by combining the line causing the error with any declaratives that refer to variables used in that line. The statements preceding or following usually have no effect on syntax bugs; an exception to this includes bugs related to blocking sequences (DO-CONTINUE or IF-ENDIF).

PASS 2

Pass 2 is the code generation pass. The optimizer breaks the program unit into code blocks and processes these blocks one at a time. Generally, if CFT makes an error in one block, all of the other blocks can be deleted from a test program and CFT will still fail.

An easy way to determine the block structure of a routine is to use the ON=B option on the COS CFT statement, or the -e b option on the UNICOS CFT command. This procedure is especially useful if the error is an operand range error occurring during compilation.

The method CFT uses to divide a program unit into code blocks is relatively straightforward. A code block begins either:

- With a statement that has a label referenced elsewhere in the program unit, or
- At an entry statement, or
- After a loop ends.

A code block ends just before the next one begins or on a statement that causes a backward branch to the start of the current block.

Some I/O statements that have unusual side effects also cause a new block to start. Examples are NAMELIST and BUFFER IN. Also, IF statements often force the start of a block because they implicitly jump around code.

The block that immediately precedes an innermost DO loop is usually considered to be part of the DO-loop preamble for optimization purposes.

Pass 2 bugs generally consist of:

- Variables or intermediate results being saved in B or T registers across subroutine calls by mistake, or
- Calculations being mistakenly removed from a DO loop, or
- Loop quantities not being properly saved for relooping.

INDEX

READER COMMENT FORM

FORTRAN (CFT) Internal Reference Manual SM-0017 B-03

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

DATE _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

STAPLE

READER COMMENT FORM

FORTRAN (CFT) Internal Reference Manual SM-0017 B-03

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

DATE _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

2520 Pilot Knob Road
Suite 350
Mendota Heights, MN 55120
U.S.A.

Attention:
PUBLICATIONS

STAPLE

PUBLICATION CHANGE NOTICE

March 1986

TITLE: FORTRAN (CFT) Internal Reference Manual

PUBLICATION NO. SM-0017 **REV.** **CHANGE PACKET NO.** B-03

This change packet brings the manual into agreement with the CFT 1.15 release. Please make the following changes to your manual.

Replace:
Title page through ix
1-1 through 1-4
1-7 and 1-8
2-1 and 2-2
2-17
3-3 through 3-8
4-9 and 4-10
4.TBBK-1 and 4.TBBK.2
4.TBD-1

Add:
4.TBD-2

Replace:
4.TBT-2.1
4.TBZ-1 through 4.TGB-10
5-1 and 5-2
5-7 and 5-8

Add:
5-8.1

Replace:
5-11 and 5-12

Add:
5-12.1

Replace:
5-23 and 5-24
5-27 through 5-32
5-35 through 5-38

Add:
5-38.1 and 5-38.2

Replace:
5-43 through 5-46

Add:
5-46.1

Replace:
5-53 and 5-54
5-59 through 5-62
5-69 through 5-72
6-1 through 6-4 (all of section 6)
C-1 and C-2
Index