CREF

CREF produces a sequence-numbered assembly listing followed by one to three tables, one showing cross references for all operand-type symbols (labels, assignments, etc.), another showing cross references for all user-defined operators (macro calls, OPDEFs, etc.), and another (if the proper switch is specified) showing the cross references for all op codes and pseudo-op codes (MOVE,XALL, etc.) A number sign (#) appears on the definition line of all symbols. The input to CREF is a modified assembly listing file created during a MACRO-10 assembly or FORTRAN IV compilation when the /C switch is specified in the command string.

Detailed information on CREF is contained in the DECsystem10 Assembly Language Handbook in the Utilities Section.

WARNING

The following changes have been made to the TENEX version of CREF:

1. After CREF has processed a source file, it is deleted from the user's directory.

2. The output listing from CREF is generated for wide line printer paper (132 characters per line). Unless wide paper is in the line printer, output to a disk file and request that the operator list the file on wide paper. (Narrow paper is standard for the TENEX line printer. Wide paper must be explicitly requested.)

3. If an output file name is specified without an output device, the default output device will be DSK. If neither an output device nor an output file name is specified, the default output device will be LPT.

FORMAT

Two input formats are acceptable to CREF. The first is produced by early versions of MACRO (prior to version 30), the PALX assembler for the PDP-8, and early versions of FORTRAN (prior to version 06). The second input format for CREF is produced by current versions of MACRO (version 30 and later), version 06 and later of FORTRAN, FORTRAN-10, and the Stanford FAIL assembler.

EARLY INPUT FORMAT
#
The codes listed below are produced by early versions of MACRO and FORTRAN as input to CREF. They are ignored by CREF if the control characters produced by the current versions of MACRO and FORTRAN are present.
#

ASCII Code	Meaning
33	Indicates that the code type is an op code, a pseudo-op code, or an op code defined by the user by OPDEF.
34	Indicates that the code type is a macro name.
35	Indicates the end-of-line.
36	Indicates a normal symbol; i.e., a symbol defined with an equal sign (=) or a colon (:).
37	Indicates a program break (between FORTRAN subroutines).

This input format to CREF should not be used when new programs are being developed.
#
CURRENT INPUT FORMAT
#
The control characters described below are placed on the listing produced by current versions of MACRO ,FORTRAN, and FORTRAN-10 as input to CREF.
#
Normally, each line of the listing contains CREF input data followed by the line of the listing. The CREF input data on each line is preceded by RUBOUT B and terminated by RUBOUT C. Each symbol or instruction type in the listing is defined in the CREF input by a control character (described below). The number of characters in each symbol or instruction is also defined by a control character. The set of control characters for defining symbols and instructions is identical to the set of control characters for defining the number of characters in the symbol or instruction. The position of a control character in the CREF input data determines the use of the control character. For example, in the input CREF data B^C^CENDC, the B indicates the beginning of the data, the first ^C defines the instruction END as a pseudo-op code, the second ^C defines the number of characters in the instruction END as 3, and the C terminates the

CREF data.

Optionally, CREF information may be terminated by either RUBOUT A or RUBOUT D. If RUBOUT D is used, more than one block of CREF information may be placed on a single line.

The control characters and their meanings are described below.

Beginning and Ending Control Characters

The control characters that begin and end the CREF input data are:

RUBOUT B (prints as B)	Signals the beginning of the CREF data on each line
RUBOUT C (prints as C)	Terminates the CREF data on each line and inserts the line number to the listing
RUBOUT A (prints as A)	Terminates the CREF data on each line and inserts the line number and a horizontal tab to the listing.
RUBOUT D (prints as D)	Terminates a block of CREF data in a line without inserting any additional information in the listing
RUBOUT E (prints as E)	Indicates program break (between FORTRAN subroutines)

Symbol-Definition Control Characters

The control characters that define symbols, instruction types, and macros are:

Character	ASCII Code	Meaning
CONTROL-A (^A)	ØØ1	Precedes each symbol that is defined with an equal sign (=) or a colon (:) each time the symbol appears in the listing; e.g., FOO:

CONTROL-B (^B)	002	Immediately follows the defining occurrence of the symbol defined by equal sign or colon.
CONTROL-C (^C)	003	Precedes the use of an op code (either hardware-defined or defined by OPDEF) or a pseudo-op code.
CONTROL-D (^D)	004	Precedes the defining occurrence of an op code defined by OPDEF.
CONTROL-E (^E)	005	Precedes a macro call.
CONTROL-F (^F)	006	Precedes the definition of a macro.
CONTROL-G (^G)	007	Precedes the definition of a line number
	015	Signals the beginning of a FAIL symbol block.
	016	Signals the end of a FAIL symbol block.

Although CREF recognizes and accepts all of the above control characters, current versions of MACRO do not produce all of these characters. As shown above, CREF recognizes a symbol defined by OPDEF as an op code because it is preceded by ^D when it is defined, and by ^C when it is used. MACRO treats a symbol defined by OPDEF as a macro and thus precedes it by ^F when it is defined, and by ^E when it is used. CREF also recognizes these symbols as macros because of the control characters produced by MACRO. The fact that symbols defined by OPDEF are treated as macros has no effect on the cross- reference listing from CREF because OPDEF's and macros are grouped into the same table.

Character-Count-Definition Control Characters

The octal value of the control characters described below is used by CREF to determine the number of characters in a symbol or instruction. The same set of control characters define the symbol as well as the number of characters in the symbol. The position of the control character in the input data determines the use. The character-count control character immediately precedes the symbol with no intervening spaces or characters (e.g., ^CEND). The control characters and their meanings are as

follows:
#
#
Character	ASCII Code	Meaning
CONTROL-A (^A)	001	The symbol contains 1 character
CONTROL-B (^B)	002	The symbol contains 2 characters
CONTROL-C (^C)	003	The symbol contains 3 characters
CONTROL-D (^D)	004	The symbol contains 4 characters
CONTROL-E (^E)	005	The symbol contains 5 characters
CONTROL-F (^F)	006	The symbol contains 6 characters

No symbol or instruction can contain more than six characters.
#
#
Example of the Current Input Format
#
The example below shows a small MACRO program and the listing
produced by MACRO to be input to CREF.
#
```
#         ;CREF SPECIAL CHARACTER DEMONSTRATION
#  .MAIN   MACRO 44.0 09:30 10-DEC-70 PAGE 1
#
#         ;CREF SPECIAL CHARACTER DEMONSTRATION
#         M=6                         ;1 CHAR SYMBOL DEFINITION
#         MOVEI M                     ;5 CHARACTER OPCODE
#                                     ;1 CHAR SYMBOL USE
# FOO:    SIXBIT /123/                ;3 CHAR SYMBOL DEFINITION
#                                     ;6 CHAR PSEUDO INSTRUCTION
#         MOVEI 6+FOO                 ;MORE OF THE ABOVE
# OPDEF TTYCAL [51B11]                ;OPCODE DEFINITION
# DEFINE TEST (X) <TLNE X>            ;MACRO DEFINITION
#         TTYCAL                      ;OPCODE USE
#         TEST M                      ;MACRO CALL & SYMBOL USE
#         END                         ;PSEUDO INSTRUCTION OCCURRENCE
#
#  .MAIN   MACRO 44.0 09:30 10-DEC-70 PAGE 1
# TEST  .MAC
# BC                                      ;CREF SPECIAL CHARACTER DEMO
# B^A^AM^BC                               000006 M=6
```

```
#                                        **  ;1 CHAR SYMBOL DEFINITION
# B^C^EMOVEI^A^AMC          000000' 201000  000006 MOVEI M
#                                        **  ;5 CHARACTER OPCODE
# BC                                       ;1 CHAR SYMBOL USE
# B^A^CFOO^B^C^FSIXBITC 000001' 212223  000000  FOO;  SIXBIT /123/
#                                        **  ;3 CHAR SYMBOL DEFINITION
# BC                                       ;6 CHAR PSEUDO INSTRUCTION
# B^C^EMOVEI^A^CFOOC  000002'  201000  000007'   MOVEI 6+FOO
#                                          ;MORE OF THE ABOVE
# B^C^EOPDEF^F^FTTYCALC                   OPDEF TTYCAL [51B11]
#                                        **  ;OPCODE DEFINITION
# B^C^FDEFINE^F^DTESTC                    DEFINE TEST (X) <TLNE X)
#                                        **  ;MACRO DEFINITION
# B^E^FTTYCALC      000003'  005100  000000 TTYCAL
#                                        **  ;OPCODE USE
# B^E^STESTC                              TEST M
#                                        **  ;MACRO CALL _SYMBOL USE
# B^C^DTLNE^A^AMC     000004'   603000 0000006 TLNE M
# B^C^CENDC                               END
#                                        **  ;PSEUDO INSTRUCTION OCCURRENCE
# NO ERRORS DETECTED
# PROGRAM BREAK IS 000005
# 2K CORE USED
#  .MAIN   MACRO  44.0 09:30  10-DEC-70 PAGE 2
# TEST   .MAC     SYMBOL TABLE
# FOO                              000001'
# M               000006
# TTYCAL 005100   000000
```

DDT

DDT is the debugger for most of the TENEX/DEC language processors. It is described in detail in the DECsystem1Ø Assembly Language Handbook. The differences between TENEX DDT and DEC DDT are specified here.

1. The "undefined symbol" assembler has several improvements. "LOADER" leaves a table of undefined symbols pointed to by the contents of a ".JBUSY" (117). DDT now uses this same table (rather than its own separate table) for assembling undefined symbols. The result is that DDT may be used to define and automatically patch locations for symbols that LOADER said were "undefined externals". This will work correctly for "linked" references or for "additive requests" in either the right or left half of a location.

2. Using symbol# on a symbol that is already defined will give the ubiquitous ? message.

3. DDT will assemble undefined symbols only when:

   1. There exists a reasonable .JBUSY pointer.

   2. A register is open and being modified, ("symbol#=" is not valid.)

   3. Only the arithmetic operations plus or minus may be used on the symbol. Multiply or divide or parentheses may not be used.

4. The problem of user defined tags being confused with machine operation codes finally has a reasonable solution. An input symbol is considered to be a machine operation code (and searched for first in DDT's OP code table, then in the user's symbol table) if:

   1. It is the first symbol or number input as part of an expression, and

   2. It is terminated by a space.

   The input symbol is searched for first in the user's symbol table (then in DDT's OP code table) if the above conditions are not both true.

EXAMPLES: Suppose MOVE is defined to be 6 (6<MOVE:), then:

```
MOVE=6
MOVE =200000,,0
MOVE MOVE=200000,,6
MOVE MOVE =200000,,6
MOVE+MOVE=14
```

Note that "space" and "+" are not equivalent. As a general rule, use spaces and pluses the same way that they are used in MACRO. Also, to force a symbol to be interpreted as a machine operation code, type it first and terminate it with a space.

5. The form "symbol?" will list all the program names where "symbol" is defined. The program name will be followed by "G" if "symbol" is a global symbol.

6. "FOO 1,,FOO 1" will now give the same results as "FOO+1,,FOO+1". Note again that space and plus are not normally equivalent, but have been made to be so in this special case. People insist on the "FOO 1,,FOO 1" form even though the DDT manual doesn't allow it. Remember, MACRO won't allow it either.

7. The PC word flags are now saved and restored correctly for all cases in the breakpoint logic.

8. For the instructions JRST, JFCL, and XCT, the accumulator field is always typed out in numeric mode, not symbolic.

In addition to the above, the "GO" command ($G) has been improved. In general, commands with two altmodes such as FOO$$G clear the interrupt system before going to FOO. With a single altmode, the interrupt system is not affected.

If there is a small number between the altmode(s) and the G, and no argument supplied, the command will start the program at the specified entry vector location. Thus, $$0G is the equivalent of the EXEC command @START, while $$1G is the same as @REENTER, $3G starts the program at the third entry vector location.

If the fork has a DEC 10/50 style entry vector ("length" = 254000), only 0 or 1 is legal between the altmode(s) and the G. In such cases, the contents of location 120 is used for $0G and contents of location 124 for $1G for consistency with DEC conventions.

There are two special cases: $$G is an abbreviation for $$0G and will start the program at its normal start address. $G means the same as $I which is where the user's flags,,PC are stored while

in DDT.

When DDT is entered, it will attempt to automatically set the program name (i.e., MAIN.$:) by finding which program contains the start address.

The following commands were added earlier in the history of TENEX:

$$Q has the value of the last quantity typed with halves swapped.

$V has the value of the left half of the last quantity typed.

$$V has the value of the left half of the last quantity typed with sign extended.

thus

```
FOO/ -4,,3  $Q_ -4,,3 $$Q_ 3,,-4        $$Q_ -4,,3

FOO/ -4,,3  $$V_-4

FOO/ -4,,3  $V=       777774
```

For completeness, the following is a list of differences between BBN DDT and DEC DDT.

DEC DDT $G with no argument is done with $$G in TENEX DDT.

BBN DDT does not have paper tape commands: $J, ^R and $L.

BBN DDT uses the Stanford block structured symbol table code which permits debugging FAIL-assembled programs. The command FOO$& will make the symbols in block named FOO current. This is relevant only for FAIL-assembled programs.

CONTROL-A in DDT causes its argument to be taken as RADIX-50 SQUOZE code.

DOUBLE-QUOTE (") sets up to accept a string in the same way that the ASCII pseudo-instruction does in MACRO. That is, "/STORED TXT/" will exactly fill two PDP-10 words with ten 7-bit characters.

$$".FOO MUNG. This is the sixbit text storage command. The point (.) is a delimiter, as was the slash in the above example. This example will fill one and one-half words (9

characters).

"Q$= will type the octal value of ASCII Q.

"/Q/= will type a 36 bit number containing ASCII Q in the left seven bits.

Note that if a register is not open, DDT will not permit inputting more than one word's worth of characters.

The BBN hardware instructions have been added to the OPCODE table.

In floating point type out mode ($F) unnormalized numbers are printed as ordinary decimal numbers with a decimal point.

DELVER

DELVER is a program for assisting in the management of file versions. It provides the ability to delete excess* versions of files according to the most frequently needed algorithms. DELVER will delete excess versions of files as specified by a standard TENEX file group designator. For each group of files, two options are provided. The oldest (lowest version number) version may be optionally deleted and the version numbered one less than the most recent (highest numbered) version may be optionally deleted. This provides the ability to save a version which may be closest to the most recent in case that version gets lost, and the ability to save a version which is the most likely base for a series of changes for source comparisons etc. If any version number is greater than one of most recent, second most recent, or oldest, (time-wise) it must satisfy all tests before it will be deleted.

To use DELVER, type DELVER<cr> to the TENEX EXEC. Then answer the two questions about whether or not to delete the second most recent and oldest versions with either Y or N depending on which options are desired. DELVER will then ask for the file group designator over which deletion is to occur. Default group designator is *.*;*, i.e. everything in the connected directory. Each file deleted is printed on the TTY. If DELVER deletes any files you do not wish to have deleted, the EXEC command UNDELETE may be used to correct the error. Note that if DELVER is being overzealous, the best way to stop it is to type a single control-C (ETX). This will allow printout of all files actually deleted to appear on the terminal. Using multiple control-C (ETX) will not stop the program any faster, but will clear the output buffer of the terminal and thus lose any record you might have of what has actually happened.

---

*(i.e. all but the most recent, second most recent, and oldest, with the optional exceptions described below)

DO

DO is a new subsystem for passing a parameterized text string from a specified file to the EXEC for execution. Parameters are indicated in the text file by a % character, followed either by

1. a digit, or

   string not containing the character, followed by the second instance of the character.

For each parameter, the user is asked for a text word (arbitrary text delimited by space, tab, or cr.) to substitute for each instance of the parameter in the input text string. He is prompted for this word by the digit, if a parameter of type 1, or by the text string, if a parameter of type 2. Each uniquely named parameter is prompted for once, even if it is used several times in the input text string. See the attached example.

The DO program simply dumps the expanded text string into the input buffer, then HALTF's. It behaves just as if the user had typed the expanded text string ahead to the EXEC. Therefore, Control-C can be used to clear the input buffer and return to the EXEC. Before stuffing the input buffer, the program waits, checks for other input characters from user type-ahead, and, if so, rings bells, waits again, buffers the characters, and appends them to the expanded text string. Thus, the user may type ahead, even while DO is running, and is warned to stop while the program is stuffing the input buffer.

NOTES:

1. A % character which is followed by another % character expands to a single % character (this is an escape convention for % characters in the expanded text string).

2. Beware: the TTY input buffer is currently limited in size; if the expanded text string is too long (>119 characters), the extra characters will get lost, just as if you had typed too far ahead. A change to TENEX to allocate more TTY input buffer space in such situations is planned.

```
# ;  <XBCPL>DO.EXAMPLE;3    WED 10-APR-74 3:49 PM          PAGE 1
#
# @teco.SAV;12908
#
# *;Y$
#
# INPUT FILE: EXAMPLE.;1 [confirm]
# 47 CHARS
#
# *zt$
# LOADER
# sys:bcplib
# dsk:%#Program#$ssa$$$%#Program#.sav
#
#
# *;u$
#
# OUTPUT FILE: EXAMPLE.;2 [New version]
#
# *;h$
# @do
#
# Input file: exAMPLE.;2 [Old version]
# Program: t
# @LOADER
# *sys:bcplib
# *dsk:t$
#
# FIRST 4K CORE, 461 WORDS FREE
# LOADER USED 7+4K CORE
#
# EXIT.
# ^C
# @ssaVE (PAGES FROM) 0 (TO) 777 (ON) t.sav [New version]
```

DTACPY

DTACPY copies full DECtapes to full DECtapes. There is an option to write a copy of DTBOOT onto the front of a DECtape without having to copy it from another tape. Also it can relocate the bootstrap for any size of core.

Functions by reading a REL FILE of DTBOOT, produced by assembling DTBOOT with REL==1, and keeping it in core. This file is then used and relocated when needed.

DECtape.

EXAMPLE:

```
COPY TAPE OR NEW BOOTSTRAP (C OR N)?


(which does the old COPY functions if you say "C" and does the
new BOOTSTRAP function if you say "N". If the BOOTSTRAP has
not been put into DTBOOT, it will ask for it. The version
currently on SUBSYS has the BOOTSTRAP in it.


COPY DTA1: (TO) DTA2: [CONFIRM],_
COPY BOOTSTRAP? Y
COPY REST OF TAPE ? Y
VERIFY? Y
^C
@ST


COPY DTA1: (TO) DTA2: [CONFIRM],_
COPY BOOTSTRAP? Y
COPY REST OF TAPE? N
VERIFY? N
^C
@ST


COPY DTA1: (TO) DTA2: [CONFIRM],_
^C
@
```

DFTP User's Guide

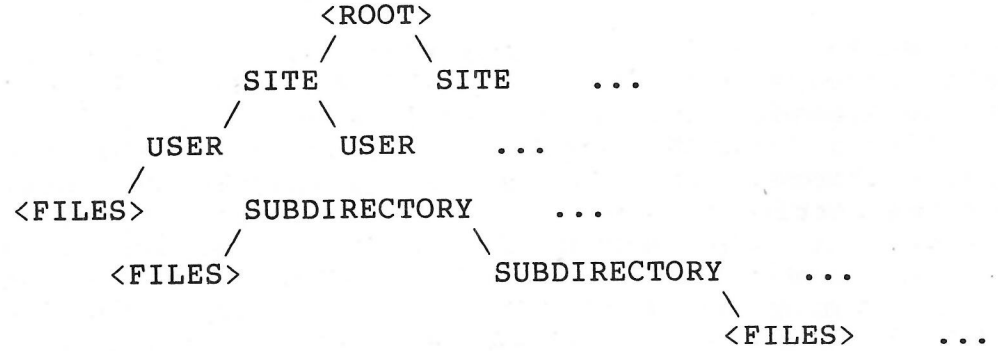
Overview

The Datacomputer is a shared large-scale data base utility offering data storage and data management services to other computers on the Arpanet. The system is intended to be used as a centralized facility for archiving data, for sharing data among various network hosts, and for providing inexpensive on-line storage for sites needing to supplement their local capability. The Datacomputer is implemented on dedicated hardware, and comprises a separate computing system specialized for data management. Logically, the system can be viewed as a closed box shared by multiple external processors and accessed in a standard notation called Datalanguage.

The Datacomputer File Transfer Program (DFTP) is a user-invoked program that stores and retrieves local files on the Datacomputer. DFTP translates simple user commands into Datalanguage, sends the Datalanguage and data to the Datacomputer, processes the messages and data returned from the Datacomputer, and notifies the user of the results. DFTP also manages local file input/output and secondary network connections to and from the Datacomputer.

The Directory

The DFTP Datacomputer directory is a tree, with site nodes anchored to a common root node, user nodes subordinate to site nodes, optional subdirectories of arbitrary depth and breadth beneath user nodes, and user files stored in special leaf nodes (called '<FILES>' nodes). Pictorially,



The <FILES> nodes are the repositories of all data. The user files they contain are not known individually to the Datacomputer (unlike nodes), but are separate entities only to DFTP. There can be only one <FILES> node directly under any given user or subdirectory node. DFTP users do not reference a <FILES> node directly; a reference to a file under a specific user or subdirectory node is expanded into a reference to that file in the <FILES> node under the specified node. (If a node is not specified a default is supplied).

There are two basic types of commands -- those that reference only nodes and those that reference user files. Node level commands operate at the global level of sites, users, and subdirectories. File level commands operate at the local level (inside a <FILES> node) storing, retrieving, and modifying data within that node. The argument to a file level command can consist only of a file name, or of a file name preceded by a node argument (such as the node level commands take).

Referencing Nodes

The mechanism for referencing a node, called a node path, consists of a context and a node list. A context is an anchoring point for node name references (indicated by one, two, or three left-angle-brackets); if none is specified, DFTP supplies a default. A node list is a sequence of node names, starting from the anchor, defining the desired branch of the directory tree.

There are three contexts, TOP, ATTACH, and CONNECT.

1. The top context ('<<<') anchors the node path at the root node and is used primarily for referencing other site and user nodes.

2. The attach context ('<<') is a node path, set by the ATTACH command (and by DFTP automatically at the beginning of a session), and usually indicates a user node. It is used mainly as a reference point for name space division beneath the user node.

3. The connect context ('<') is a node path (initially the same as the attach context), set by the CONNECT command, and conventionally indicates a user node or subdirectory.

A node list consists of a sequence of node names (consecutive levels in the tree) separated by right angle brackets. A password may be necessary in acquiring access privileges at a particular node, in which case the node name is followed by a colon and the password. Sets of nodes can be referenced -- all nodes at a particular level are indicated by '*', and all inferior nodes are designated by '**' (which can occur only at the end of the node list).

For example,

```
<<<CCA>HACKER:>WALDO
```

Starting at the top context, the node path references the subdirectory WALDO under user HACKER at site CCA (with a password supplied to gain access to HACKER).

```
<<WALDO>**
```

Starting at the attach context, the node path references the subdirectory WALDO and all inferior nodes (note that WALDO is included -- the REMOVE command, for example, would delete the node WALDO as well as its inferiors).

Referencing Files

User file names have the same form as TENEX file names: a file designation, an optional extension, and an optional version number. The file designation is separated from the extension by a period, and the extension from the version number by a semicolon. File sets may be indicated by an asterisk in any or

all of the file designation, extension, and version number fields.

Version numbers allow unambiguous reference of files with the same file designation and extension. Each file has a version number assigned to it by DFTP (which is unrelated to its TENEX version number) -- version numbers cannot be set by the user. Later versions of a file with the same file designation and extension receive higher version numbers. A version number may be explicitly supplied in referencing an existing file, otherwise a default is provided.

All commands that accept as input a file name will also accept a file path, which consists of a node path followed by a file name, with the two parts separated by a right angle bracket (unless the node path is only a context, in which case the right angle bracket is omitted). If a node path is given, the file name is used in the <FILES> node under the node referenced. If a node path is not given, the file name is used in the <FILES> node under the connect context -- the default context for a file reference is the connect context.

For example:

```
MAIL.TXT
```

The file name references the file MAIL.TXT in the <FILES> node under the connect context.

```
<*.SAV;*
```

The file name references all versions of all files with the extension SAV in the <FILES> node under the connect context.

```
<<MACROS>COMMON>SYSMAC.MAC
```

Starting at the attached context (presumably a user node), the file path references the file SYSMAC.MAC in the <FILES> node under the COMMON subdirectory of the MACROS subdirectory.

Command Summary

DFTP command and argument input is similar to TENEX, with command recognition and TENEX editing controls. In particular,

<control-A> deletes a character,

<control-R> retypes the line,

<control-X> and <rubout> delete the line,

<escape> and <space> are separators, and

<carriage return>, <line feed> and <eol> are terminators.

The DFTP commands and their arguments are:

ATTACH <node path>

CONNECT <node path>

DELETE <file path>

DIRECTORY <file path>

TERSE

VERBOSE

EXPUNGE <node path>

GET <file path> [local synonym]
RETRIEVE is exactly equivalent to GET.

LIST <node path>

NO-DATALANGUAGE

PUT <file path> [remote synonym]
STORE is exactly equivalent to PUT.

QUIT

REMOVE <node path>

SHOW-DATALANGUAGE

TIME-TRANSFERS

UNDELETE <file path>

UNTIME-TRANSFERS

Items in angle brackets are required arguments; items in square brackets are optional ones.

The connect context is the default context for all commands except ATTACH and CONNECT, which have as their respective defaults the top context and the attach context.

Many commands have default arguments and trailers which are invoked by giving a space or escape as the argument or argument terminator. The default argument is '<<' for the CONNECT command, '<' for EXPUNGE, and '**' for LIST and REMOVE, which is also the default trailer. For DIRECTORY, GET, PUT, DELETE, and UNDELETE, the default argument and trailer (after a '>') is '*.*;*'.

Node Oriented Commands

The ATTACH command sets the attach context and initiates Datacomputer accounting functions.

The CONNECT command sets the connect context (and creates new subdirectories).

The EXPUNGE command removes files marked as deleted from the <FILES> node under the node given as the command argument. If the <FILES> node contains no files (deleted or undeleted) it is deleted from the Datacomputer directory.

The LIST command lists Datacomputer nodes (sites, users, subdirectories, and <FILES>) and information about them. The information displayed by the VERBOSE option comes directly from the Datacomputer.

The REMOVE command removes nodes from the Datacomputer directory; they must either have no inferior nodes, or be part of a node set specified using "**". In the latter case, data stored under the nodes will also be deleted.

File Oriented Commands

File deletion operates as with TENEX. The DELETE command marks files as deleted, but does not eliminate them. They can be listed via the VERBOSE option of the DIRECTORY command, and their deleted status can be changed by the UNDELETE command. The removal of deleted files is deferred until an EXPUNGE is performed on the <FILES> node containing them. The default version number is the lowest undeleted, unless a file set is indicated, in which case all versions in the set are deleted.

The DIRECTORY command lists files and information about them. The VERBOSE option lists deleted and undeleted files (with

deleted ones indicated by a D after the name), the date and time created (for TOPS-10 sites), the date and time last written (for TENEX sites), the date and time stored, and the size. For files stored from TENEX sites the size information is in the form <number of bytes>(<byte size>). For files stored from TOPS-10 sites the information is in the form <number of 36 bit words>(-<data mode)). The TERSE option lists undeleted files and their sizes (as with the VERBOSE option). The default version number is the highest undeleted, unless a file set is indicated, in which case information for all versions in the set is listed.

The GET and PUT commands retrieve and store local disk files on the Datacomputer. Files of any type (text or binary image, for example) can be stored. If a synonym is not supplied, the Datacomputer file name is used as the local file name. If the first argument to either command is completed with an escape or a space, the synonym option is invoked and the commands then operate in the form

GET Datacomputer file [AS] local file, and
PUT local file [AS] Datacomputer file.

For the GET command the default version number is the highest undeleted, unless a file set is indicated, in which case all versions of the set are retrieved. The PUT command sets the version number of the file being stored to be one greater than the highest version of existing files with the same file designation and extension (note that a file set indicated in any file name field is treated as if all existing files had the same field -- storing *.* results in the stored files receiving version numbers one greater than the highest version number found in any existing file).

The UNDELETE command rescinds a file's deleted status. The default version number is the highest deleted, unless a file set is indicated, in which case all versions in the set are undeleted.

Miscellaneous Commands

ENABLE causes DFTP to recognize an expanded set of commands, including the privileged commands discussed below. Its action is marked by a change in the prompt character, from "*" to"!".

The SHOW-DATALANGUAGE and NO-DATALANGUAGE commands respectively allow and inhibit the output to the user's terminal of the messages sent to and received from the Datacomputer.

Data transfer rates are calculated and given to the user when the TIME-TRANSFERS command has been invoked. The calculations are avoided with the UNTIME-TRANSFERS command.

QUIT exits gracefully from DFTP, closing network connections.

Site Dependent Features

For the TOPS-10 version, the LOCAL-DIRECTORY command lists the user's local file directory.

For the TENEX version, the EXEC command provides the user with an inferior exec, which is flushed when the user returns to DFTP. Where a local file name is possible (in the GET and PUT commands) an initial space or escape invokes TENEX name recognition, indicated by a right angle bracket prompt. A control-O can be used to halt the output from the LIST and DIRECTORY commands.

Responses

There are three types of messages that DFTP gives the user. Comments surrounded by square brackets are primarily informational messages, and are never errors. Parentheses enclose non-fatal errors and informational Datacomputer messages, such as '(LEBAR2: ERROR: NO SUCH FILE)', resulting from an attempt to DELETE a nonexistent file, and '(SXPF9: STAGING DATA FOR FILE = DFTP.CCA.DFTP.<FILES>)', indicating that data is being moved from tertiary mass memory to secondary buffer memory.

These messages come directly from the Datacomputer, indicated by the name and colon at the head of the message. Fatal error messages are surrounded by question marks, and of course never occur.

Access Control

Access control in DFTP uses a subset of the full Datacomputer facilities. The full discussion of Datacomputer privilege facilities is in the current Datacomputer User manual; however, the following summary should suffice for most DFTP users. Access privileges are specified in "privilege blocks" attached to nodes in the directory. A node may have any number of privilege blocks attached to it; each specifies a particular set of access privileges, and a class of users to whom that set applies. DFTP provides two classes of access:

- CONTROL allows users to create and allocate nodes under the node at which it is granted, change privileges at that node and below, and read, write, and delete data stored below that node.

- READ allows users to attach to a node and read data stored under it, but not to perform any of the other functions granted by CONTROL.

All other users are prevented from any access to the data. Users may be may be identified in DFTP by their network identity (defined by the host and socket from which they access the Datacomputer), and by passwords. (Some systems enforce assignment of socket numbers according to the user's identity on that system, thus providing a convenient automatic identification to DFTP.)

When a user attempts to ATTACH to a node, or to read or write data stored under it, the Datacomputer checks to see if the user is in any of the user-sets identified in privilege blocks on that node, and if so, assigns the corresponding class of privileges. If no set matches the user, then no privileges are allowed. The scan is done in the order of creation of privilege blocks, and if a user matches more than one, the first one takes effect. Access controls are set by the CHANGE and CREATE commands, described in the following section.

Privileged Commands

Certain administrative functions are performed by a set of restricted commands, known as "privileged commands." These are not normally available to the user; they are recognized by DFTP only after execution of the ENABLE command.

- The ALLOCATE command is used to set the maximum number of megabits a user may consume (it can also be used to set subdirectory limits). Allocations are made and reported in Megabits (actually 1,Ø13,76Ø bits), which is 55 512-word pages, or 22Ø 128-word blocks.

- The CHANGE command resets the access control information for a given node. It first deletes all existing access control specifications (privilege blocks); then it builds new privilege blocks interactively with the user. The PROTECTION subcommand of the LIST command can be used to examine the privilege blocks of nodes.

- The CREATE command is used to create a node for a new user; after the node has been created, it falls into the same access control specification as the CHANGE command.

- The DISABLE command returns the user from Privileged Command mode to normal use; this is signalled by a return to an asterisk as a prompt character.

- The LINK command allows the user to send Datalanguage to the Datacomputer directly. In this mode, prompted by a left angle bracket, each line typed by the user is sent directly to the Datacomputer. It is terminated by submission of an empty line, whereupon DFTP returns to ENABLE mode, as signalled by an exclamation-point prompt.

DFTP Command Summary
Paths

```
<node path> ::=   <context>
                | <node list>
                | **
                | <context> **
                | <context> <node list>
                | <context> <node list> > **
                | <node list> > **
<context> ::=  <                    (connect context)
                | <                 (attach context)
                | <<<               (top context)
<node list> ::=  <node>
                | <node> > <node list>
<node> ::=       <name>
                | <name> : <password>
                | *

<file path> ::=  <file name>
                | <context> <file name>
                | <node list> > <file name>
                | <context> <node list> > <file name>
<file name> ::=  <file>
                | <file> .
                | <file> . <extension>
                | <file> ; <version>
                | <file> . ; <version>
                | <file> . <extension> ; <version>
<file> ::=       <name> | *
<extension> ::=  <name> | *
<version> ::=    <number> | *
```

Notes:

Underscored angle brackets should be included literally.

Any printing ASCII characters except <, >, ., :, *, ?, ', and " may be used in a <name>.

Any printing ASCII characters (plus space) except >, ., ', and " may be used in a <password>.

Commands

ATTACH <node path>

CONNECT <node path> (1)

DELETE <file path> (4) (5)

DIRECTORY <file path> (4) (6)
        TERSE
        VERBOSE

ENABLE

EXPUNGE <node path> (2)

GET (RETRIEVE) <file path> [local file name synonym] (4) (6)

LIST <node path> (3)
        TERSE
        VERBOSE

NO-DATALANGUAGE

PUT (STORE) <file path> [remote file path synonym] (4)

QUIT

REMOVE <node path> (3)

SHOW-DATALANGUAGE

TIME-TRANSFERS

UNDELETE <file path> (4) (7)

UNTIME-TRANSFERS

Notes:

Required input is indicated by angle brackets.
Optional input is indicated by square brackets.
The connect context is the default context for all commands except ATTACH and CONNECT, which have as their respective defaults the top context and the attach context.

(1) The default argument is <<.
(2) The default argument is <.
(3) The default argument (and trailer) is **.
(4) The default argument (and trailer) is *.*;*.
(5) The default version is the lowest undeleted.
(6) The default version is the highest undeleted.

(7) The default version is the highest deleted.

Privileged Commands and Their Subarguments

```
ALLOCATE <node path>
    Megabits: [decimal integer]

CHANGE <node path>
    Add a new privilege? [Y(es)] or [N(o)]
        Allow write? [Y(es)] or [N(o)]
        Restrict via network? [Y(es)] or [N(o)]
            Restrict via local host? [Y(es)] or [N(o)]
                Host number (octal): [octal integer]
                  (if host not local)
            Restrict via user? [Y(es)] or [N(o)]
                Socket number (octal): [octal integer]
                  (if user restricted and host not local)
                User: [user name]
                  (if user restricted and host local)
                  (a directory name if TENEX)
                  (a programmer number if TOPS-10)
        Restrict via password? [Y(es) or N(o)]
            Password: [string]

CREATE <node path>
    (see CHANGE)

LIST
    PROTECTION

LINK
    (A null input line returns the user to command mode.)
```

Examples Using Privileged Commands

```
;Attach to a node.
; (gain control at site CCA by supplying the proper password)
*ATTACH <<<CCA:

*ENABLE

;Create a user and privileges.
; (the first privilege allows creation and deletion for
;   local user HACKER upon supplying the password "ETAOIN")
; (the second privilege allows creation and deletion for
; anyone from Harvard (host 11) upon supplying the password "SHRDLU")
```

```
; (the third privilege allows anyone read)
!CREATE HACKER
 [OK]
 Add a new privilege? Yes
  Allow control? Yes
  Restrict via network? Yes
   Restrict via local host? Yes
   Restrict via user? Yes
    User: HACKER
  Restrict via password? Yes
   Password: ETAOIN
  [OK]
 Add a new privilege? Yes
  Allow control? Yes
  Restrict via network? Yes
   Restrict via local host? No
    Site: 11
   Restrict via user? No
  Restrict via password? Yes
   Password: SHRDLU
  [OK]
 Add a new privilege? Yes
  Allow control? No
  Restrict via network? No
  Restrict via password? No
  [OK]
 Add a new privilege? No

;List the privileges.
; (in Datacomputer format (passwords are never listed))
!LIST HACKER
!!PROTECTION
    CCA
        HACKER
            ] (1),U=**,H=31,S=12582928,G=CLWRA
            ] (2),U=**,H=9,S=ANY,G=CLWRA
            ] (3),U=**,H=ANY,S=ANY,G=LR

;Replace the privileges.
; (the first privilege allows creation and deletion for
;   local user HACKER)
; (the second privilege allows anyone read
;   upon supplying the password "WALDO")
; (the "[OK]" indicates that the previous privileges have been delete)
!CHANGE HACKER
 [OK]
 Add a new privilege? Yes
  Allow control? Yes
  Restrict via network? Yes
   Restrict via local host? Yes
   Restrict via user? Yes
    User: HACKER
```

```
  Restrict via password? No
  [OK]
 Add a new privilege? Yes
  Allow control? No
  Restrict via network? No
  Restrict via password? Yes
   Password: WALDO
  [OK]
 Add a new privilege? No

;List the privileges.
!LIST HACKER
!!PROTECTION
    CCA
        HACKER
            ] (1),U=**,H=31,S=12582928,G=CLWRA
            ] (2),U=**,H=ANY,S=ANY,G=LR

;List all information.
; ("MX-U" indicates the maximum allocation in megabits)
!LIST HACKER
!!VERBOSE
    CCA
        HACKER
            ] MX-U=10.00 CHRG=0.00
            ] IN-N=0 IN-F=0
            ] CREA=761101052805

;Change the allocation.
; (decrease the allocation from 10 megabits to 2 megabits)
!ALLOCATE HACKER
 [Megabits:2]


;List all information.
!LIST HACKER
!!VERBOSE
    CCA
        HACKER
            ] MX-U=2.00 CHRG=0.00
            ] IN-N=0 IN-F=0
            ] CREA=761101052805
```

ECAP

ECAP is an Electronic Circuit Analysis Program.

This is an integrated system of programs which can be used for design and analysis of electronic circuits. The system of programs can produce DC, AC, and/or transient analyses of electrical networks from a description of the connections of the network (the circuit topology), a list of corresponding circuit element values, a selection of the type of analysis desired, a description of the circuit excitation, and a list of the output desired.

The user requires neither a knowledge of the internal construction of the system of programs nor computer programming techniques to use ECAP effectively.

This subsystem was originally distributed as DECUS No. 10-34. It is documented in The IBM 1620 Electronic Circuit Analysis User's Manual, #GH20-0170-2.

ECAP accepts input from the file DSK:INPUT.DAT and writes its output on DSK:OUTPUT.DAT.

It is started by typing ECAP to the EXEC.

```
#                              FILCOM
#
# FILCOM compares two files in either ASCII mode or binary
# depending upon switches of file name extensions. All standard
# binary extensions are recognized as binary by default.
#
#
# Switches are:
#
#
# /A    compare in ASCII mode
# /B    allow compare of Blank lines
# /C    ignore Comments and spacing
# /S    ignore Spacing
# /H    type this Help text
# /#L   Lower limit for partial compare
#       or number of Lines to be matched
#       (# represents an octal number)
# /#U   Upper limit for partial compare
# /Q    quick compare only, give error message if files differ
# /U    compare in ASCII Update mode
# /W    compare in Word mode but don't expand files
# /X    expand files before word mode compare
```

FASBOL II

FASBOL II is a SNOBOL Compiler for the PDP-10 written by Paul Joseph Santos, Jr.

ABSTRACT

The FASBOL II compiler system represents a new approach to the processing and execution of programs written in the SNOBOL4 language. In contrast to the existing interpretive and semi-interpretive sytems, the FASBOL compiler produces independent, assembly-language programs. These programs, when assembled, and using a small run-time library, execute much faster than under other SNOBOL4 systems.

While being almost totally compatible with SNOBOL4, Version 3, FASBOL offers the same advantages as other compiler systems, such as:

1. Up to two orders of magnitude decrease in execution times over interpretive processing for most problems.

2. Much smaller storage requirements at execution time than in-core systems, permitting either small partitions or large programs.

3. Capability of independent compilation of different program segments, simplifying program structure and debugging.

4. Capability of interfacing with FORTRAN and MACRO programs, providing any division of labor required by the nature of a problem.

5. Measurement and runtime parameter facilities to aid in optimizing execution time and/or storage utilization.

CHAPTER 1

1. Introduction

The first FASBOL [1] was a similar system designed and written for the UNIVAC 1108 under the EXEC II operating system, and operational as of October 1971. FASBOL II, the PDP-10 system , is an enhanced version which is in addition compatible with Version 3 of SNOBOL4. It is presumed that the reader is familiar with SNOBOL4, Version 3, as described by the second edition (1971) of the Prentice-Hall publication [2]. Using [2] as a base description of SNOBOL4, the following chapters explain any differences and additions present in FASBOL II, as well as describe how to use it to compile and run programs.

The FASBOL II compiler is itself written in FASBOL and is like FORTRAN and MACRO in that it accepts specifications for source, listing, and object files (the object is a MACRO program which must be assembled). The reason for writing the compiler in FASBOL was for speed of implementation, automatic checkout of the run-time library, and ease of modification. If after some use the compiler should prove to be unsatisfactory in terms of core utilization or execution speed, the MACRO stage can be hand-tailored, using the measurement techniques available in FASBOL, into a more efficient program. A further enhancement would be the direct production of relocatable code by a one-pass compiler written in either FASBOL or MACRO.

The FASBOL II run-time library is written in MACRO, since its efficiency is paramount, and is searched in library mode, after loading all FASBOL programs, in order to satisfy program references to predefined primitives and system routines. Sections of the FORTRAN library may also be loaded, provided they do not compete with FASBOL for UUO's and traps.

Internal documentation of the operation of the run-time system is available as a separate document.

Use of the male gender in third-person references in this manual in no way implies that FASBOL is not useful for female persons; the author is simply not aware of any easy way to write in neuter.

CHAPTER 2

2. Language Description

The syntax for FASBOL II is given in Appendix 1. In addition to the detailed changes mentioned below, this syntax differs from that given in [2] only in that it is more restrictive of compile-time syntax. For example, since FASBOL II does not permit redefinition of operators, the expression

```
( A . B ) + ( C . D )
```

is flagged as a compilation syntax error, whereas the interpreter (i.e. the system described in [2]) would accept it and then produce an "illegal type" error message during execution. Most SNOBOL4 programs should run "as is" under FASBOL II; Sections 2.1.1 and 2.1.2 describe exactly all features that may cause incompatibility, and the remaining sections deal with enhancements.

2.1 General Language features

The following three sections discuss, respectively, features of SNOBOL4 not implemented in FASBOL II, features of SNOBOL4

implemented differently in FASBOL II, and additional features available in FASBOL II and described more completely in sections 2.2, 2.3, 2.4, and 2.5.

2.1.1 SNOBOL4 features not implemented

The predefining primitives EVAL and CODE, the datatypes CODE and EXPRESSION, and direct gotos (as used with CODE) are not implemented. They imply a run-time compilation capability which is not available in the FASBOL library at this time.

The redefinition of operators via OPSYN, or the redefinition of predefined primitive pattern variables (e.g. ARB) or functions (e.g. SPAN) is not permitted in FASBOL, which considers all these items as a structural part of the language essential to generating efficient code. For this reason, the keywords &ARB, &BAL, &FAIL, &FENCE, &REM and &SUCCEED are not needed and therefore not implemented. Also, &CODE has no meaning for the PDP-10, and is not available either.

The SNOBOL4 tracing capability is not implemented in FASBOL at this time. However, the &STNTRACE keyword (see section 2.5) provides some tracing capability.

Although QUICKSCAN mode for pattern matching is implemented in FASBOL, two features of this mode, available in SNOBOL4, are not implemented. They are a)continual comparison of the number of characters remaining in the subject string against the number of characters required by non-string-valued patterns, and b) assumption that unevaluated expressions must match at least one character. This implies that some matches may last a little longer and perhaps have a few more side-effects (e.g. via $), and that left-recursive pattern definitions will loop indefinitely (see Section 3.2.3).

2.1.2 SNOBOL4 features implemented differently

The FASBOL I/O structure is time-sharing oriented and does not use FORTRAN I/O, so that it differs somewhat from the SNOBOL4 I/O. Both input and output can be either line or character mode. Line mode is similar to SNOBOL4 I/O, with input records being terminated just prior to a carriage return, line feed sequence, and with this sequence being added to output records. Trailing blanks seldom occur on input, and so the &TRIM keyword is not implemented (but the TRIM function is). Character mode gets one character (including a carriage return or linefeed) on input, and outputs a string without appending a carriage return, linefeed sequence to it. INPUT, INPUTC, OUTPUT and OUTPUTC have predefined associations (see 2.4.2, I/O primitives) corresponding to line and character mode teletype input and output, respectively. PUNCH does not have a predefined association. There are additional predefined primitives for device and file selection, etc., discussed in Section 2.4.2.

Changes in program syntax are as follows:

a) Compiler generated statement numbers are always on the left.

b) Source lines (not statements) are truncated after 132 characters.

c) The character codes and extended syntax are like the S/360 version, except the character ! (exclamation point) replaces | (vertical stroke) and \ (back slash) replaces \ (not sign).

d) Binary $ and . (immediate and conditional pattern assignment) have lower precedence than the binary arithmetic operators, but higher precedence than concatenation. Thus, the expression

```
X   A + B $ C
```

is taken to mean

```
X   ((A + B) $ C)
```

In SNOBOL4, $ and . have the highest precedence of all binary operators, and would give the meaning

```
X   (A + (B $ C))
```

to the above expression.

e) The number of arguments in a function call, formal arguments in a function definition, and fields in a datatype definition are limited to a maximum of 15.

f) The object of the unary * (unevaluated expression) operator cannot be an explicit pattern structure (e. g. use LEN(*n) instead of *LEN(n) and (*patl ! *pat2) instead of *(patl ! pat2).

Changes in program semantics and operation are as follows:

a) The binary . (name) operator always returns a value of type NAME (SNOBOL4 sometimes returns a STRING). Names of TABLE entries are permitted.

b) Some predefined primitive functions operate differently than in SNOBOL4 (see Section 2.4.2).

c) &MAXLNGTH is initially set to 262143 (in SNOBOL4, the value is 5000). This value is also the absolute upper limit on string size.

d) If &ABEND is nonzero at program termination, an abnormal (EXIT 1,) exit to the system is taken.

e) Primitive functions may be called with either too few or too many arguments, even via OPSYN or APPLY.

2.1.3 Additions to SNOBOL4

Declarations are provided in FASBOL for the enhancement of programs. No declarations are ever required, but if they are used they must all precede the first executable statement in a program. Declarations are described in Section 2.2.

Additional control cards and compilation features are available, discussed in Section 2.3, and additional predefined primitive functions and keywords are described respectively in Section 2.4 and 2.5.

Additions to program syntax are as follows:

a) Quoted strings may be continued onto a new line, with the continuation character removed from the literal.

b) Single and double quotes may be included in a literal that is bracketed by the same, by use of the construction '' to stand for ' and "" to stand for " inside of literals bracketed by ' and ", respectively.

c) Comment and control lines (i.e. starting with * or -) may start inside a line image (i.e. after a ;), and consume the remainder of the line image.

d) The run-time syntax for DEFINE and DATA prototypes has been loosened to conform with the rest of FASBOL syntax by permitting blanks and tabs after ((open parenthesis), around , (comma), and before) (close parenthesis).

2.2 Declarations

FASBOL declarations have two primary purposes. One purpose is to optimize a program in space and/or time. The second purpose is to allow inter-program linkage and communication. The general form of a declaration is a call on the pseudofunction DECLARE, with two or three arguments, the first of which is always a string literal identifying the type of declaration, and the remaining arguments specifying the parameters or program symbols upon which the declaration has effect. As has been noted, all declarations must precede the first executable statement; this also implies no declaration line may contain a label. A FASBOL program with declarations can be made otherwise compatible

with a SNOBOL4 interpreter by inserting the statement

```
DEFINE('DECLARE()', 'RETURN')
```

at the beginning of the program.

2.2.1 PURGE and UNPURGE

Normally, a FASBOL application will involve a main program and several independently compiled subroutines. During execution, the run-time system maintains a run-time symbol table for each separately compiled program, as well as a global symbol table. In the absence of declarations to the contrary, all explicitly mentioned variables, labels, and functions are put into the local symbol table for that program. Thus, program X and program Y may both have labels LAB to which they perform indirect gotos. The global symbol table contains such global symbols as OUTPUT and RETURN, and any new symbols that arise during execution of any of the programs. A symbol lookup in program X first searches the local symbol table for program X, then the global symbol table, and then, if still not found, creates a new entry in the global symbol table. Thus a local symbol table never grows beyond the size determined for it at compilation time.

The purpose of the PURGE.VARIABLE, PURGE.LABEL, and PURGE.FUNCTION declarations is to eliminate symbols from the local symbol table and thus conserve space. This can be safely done for labels provided that the label is never referenced indirectly ($ goto), or explicitly and/or implicitly in a DEFINE call. A similar criterion applies to safely eliminating variables, only the number of cases to watch for is greater; any situation that requires an association between the string representing the variable name, and the actual location assigned to that variable, is such a case. For example, the statement

```
INPUT('VARB',0,60)
```

implies that the variable VARB, if it is mentioned explicitly in the program and thus assigned a location, must be in the run-time symbol table. An explicit reference to VARB would be, for example,

```
TTYLIN  =  VARB
```

On the other hand, a variable that is never referenced explicitly need not be in the local symbol table, but the first symbol lookup for it will create an entry for it in the global symbol table. In the case of functions, the only symbols that can be safely purged are the ones

corresponding to predefined primitives, since all others are needed to be able to define the user functions, via DEFINE or otherwise.

When there appear to be more symbols of a given type to be purged than left in the symbol table, the second argument to the declaration can be the pseudovariable ALL; then, the UNPURGE.VARIABLE, UNPURGE.LABEL or UNPURGE.FUNCTION declarations can be used to place specific symbols into the symbol table.

2.2.2 GLOBAL. ENTRY. and EXTERNAL

These declarations permit interprogram communication on an indirect (i.e. symbol lookup) and/or direct (i.e. loader linking) basis. The GLOBAL.LABEL, GLOBAL.VARIABLE, and GLOBAL.FUNCTION declarations override PURGE/UNPURGE and cause the specified symbols to be placed in the global symbol table instead of the local one. Only one subprogram may globalize a particular symbol, since the implication is that the variable, label, or function belongs to that program. Any other program that does not have a similar symbol in its local symbol table will then be able to reference the global symbol.

While GLOBAL provides for interprogram communication via the symbol table, the ENTRY/EXTERNAL declarations provide for more direct interprogram communication by using the linking loader to connect external references. The ENTRY.VARIABLE, ENTRY.LABEL, and ENTRY.FUNCTION declarations make the specified local entities accessible to external programs. The second and third arguments to the ENTRY.FUNCTION declaration are like the arguments to DEFINE, and the function is automatically DEFINED the first time it is called, so no extra DEFINE is necessary. The ENTRY.FORTRAN.FUNCTION declaration is similar to ENTRY.FUNCTION except that the compiler assumes the entry will be called by a FORTRAN program. Any combination of FASBOL, FORTRAN, and MACRO programs is permitted, provided the main program is FASBOL, and certain restrictions on FORTRAN code (see Section 3.2.5) are observed.

The EXTERNAL.VARIABLE, EXTERNAL.LABEL, EXTERNAL.FUNCTION, and EXTERNAL.FORTRAN.FUNCTION declarations are the converse of the ENTRY

literal) to FORTRAN is to use a variable declared to are STRING, which is the only real use for that declaration; a fixed amount of storage is allocated for the variable based on the max character count given in parenthesis after each name in the parameter list (second argument). The only restrictions on these variables, referred to here as dedicated mode variables, is that they may not have I/O associations. All keywords (except for &RTNTYPE and &ALPHABET) are treated as dedicated integers.

2.2.4 Other declarations

The OPTION declaration serves to specify various compilation options. The HASHSIZE=n declaration, ignored in all but the main program, is used to cause a larger or smaller than normal hash bucket table to be allocated for use by the run-time symbol table. The number n should be a prime and represents the number of buckets in a linked hash table; the standard value is 127. This bucket table is at the center of all symbol lookups in the runtime system, including TABLE references, so that there is a distinct tradeoff between the sparsity of the table and the time required for a lookup. The NO.STNO option causes the compiler to eliminate the normal bookkeeping on &STNO, &STCOUNT, etc. that occurs each time a statement is entered, and is helpful to speed up slightly the execution of debugged programs. The TIMER option, which is incompatible with NO.STNO in the same program, adds to the normal bookkeeping a valuable statement timing feature (see Section 3.2.4). The timing statistics on each program being timed are printed out at the end of execution, and intermediate timing statistics can be printed out during execution by using the primitive EXTIME (see Section 2.4.2).

The SNOBOL.MAIN and SNOBOL.SUBPROGRAM declarations indicate whether the program is a main program or a subprogram, and give it a name (i.e. TITLE in MACRO). In the absence of either declaration,

```
DECLARE('SNOBOL.MAIN','.MAIN.')
```

is assumed.

The RENAME declaration is used primarily to rename predefined symbols (see Appendix 2) that would otherwise conflict with a given user's. For example, if a user wished to have a variable called ARB, or his own IDENT function while retaining the primitive also, he should rename them some other names. On the other hand, if a user wants to re-define IDENT, for example, no RENAME should be used, and IDENT will become redefined when his own DEFINE is executed.

Although usually the order in which declarations occur is not important, all ('PURGE.x',ALL) declarations should precede others which also refer to entities desired to be purged. For example, the sequence

```
DECLARE('ENTRY.VARIABLE','A,B,C')

DECLARE('PURGE.VARIABLE',ALL)
```

will cause the variables A, B and C to be missed and included in the symbol table, since the purge flag only has effect on new symbols.