It should also be noted that whereas the syntax of variable and function name lists uses a comma as a separator, label lists are separated by blanks. The reason for this is that the synatx for labels includes commas, but a blank is a valid label terminator. Also, all quoted strings in declarations are delimited by single quotes. A single quote may be entered inside such a string (for example, in a label) by using the '' convention mentioned in Section 2.1.3.

2.3 Control

In FASBOL there is an expanded repertoire of control cards for controlling listing, cross-referencing, and failure protection. In the following list, the first of a pair controlling a switch is the initial mode.

LIST, UNLIST	turns program listing on,off.
NOCODE, CODE	turns object listing off, on (the generation of object code can be inhibited by not specifying an object output file).
EJECT	causes a page eject (form feed).
SPACE n	spaces n lines (or 1 line if n is absent).
NOCROSS, CROSREF	turns symbol cross-referencing off, on. This can be done for a whole program or selectively for parts of it.
FAIL, NOFAIL* (1)	turns off a compiler feature that traps unexpected statement failures. When the feature is on (NOFAIL), any statement within its scope that does not have a conditional GOTO, and which fails, will cause an error exit. An unconditional GOTO is equivalent to none at all, and will be trapped if the statement fails.

2.4 Predefined Primitives

In the following sections, only those primitives which differ from SNOBOL4 or are new in FASBOL will be discussed. Appendix 2 has a complete list of primitives available in FASBOL.

---

(1)
*Credit for this idea goes to the authors of SPITBOL [3], who also inspired the inclusion of DUPL, LPAD, RPAD, and REVERS.

2.4.1 Pattern Primitives

Three new primitives in the SPAN/BREAK class have been added; these are structural, like the other pattern primitives, and cannot be redefined.

NSPAN(class) is like SPAN, but may match the null string.

BREAKQ(class) is like BREAK, but does not look for break characters inside of substrings delimited by single (') or double (") quotes.

BREAKX(class) is like BREAK, but has alternatives that extend the match up to each succeeding break character. Operates like

BREAK(class) ARBNO(LEN(1) BREAK(class)).

2.4.2 Expression Primitives

A number of SNOBOL4 primitives work somewhat differently in FASBOL, and new primitives have been added for I/O, string manipulation, and communication with the run-time system.

COLLECT(n) forces a garbage collection, returns the total number of words collected, and fails if no block of size n or larger is available.

CONVERT(table,'ARRAY') and CONVERT(array,'TABLE') are implemented differently, by removing the above facilities from CONVERT and putting them in ARRAY and TABLE. See below.

ARRAY(table) converts a TABLE datatype to an ARRAY as described in [2], pp.122. An empty TABLE causes ARRAY to fail.

TABLE(array) converts certain types of ARRAY datatypes to a TABLE as described in [2], pg. 122. The TABLE datatype is different from all others in that, once it has been created, it exists independently from its use in the program. Thus, to reclaim the storage, it must be explicitly deleted by TABLE(table). Once a table has been deleted, further references to it are illegal.

APPLY(fun,args) will accept either more or fewer arguments than required by the function; it will reject extra ones or fill in missing arguments with null values.

OPEN(device,chan) opens an I/O device on a software channel, assigns buffers and returns the channel number.

If chan < Ø or > 15, illegal I/O unit error.
If chan = Ø, an unused channel is assigned; if channel table is full (> 15 channels), error.
If chan is already in use, illegal I/O unit error.
If device is not a string of the form:
devnam [([outbuf] [,[inbuf]])]
it is a bad prototype or illegal arg error.
If devnam is not recognized, or is not a file structure and is already assigned to a channel, illegal I/O unit.
If ([outbuf] [,[inbuf]]) is missing, (2,2) is assumed. If either outbuf and/or inbuf is missing, Ø is assumed for the missing value.
If the device allows only input (or output), the other buffer parameter is ignored.

Examples:

```
OPEN('DTA3(,4)',5)
OUTPUT('DUMP',OPEN('MTAØ'),1ØØØ)
```

RELEASE(chan) releases the software channel and all associations to it, returns all buffers to free storage, and returns a null value.

If chan < Ø or > 15, illegal I/O unit error.
If chan = Ø, release all channels in use.
If channel not in use, ignore and return.

LOOKUP(file,chan) opens file for input (reading) on software channel, returns channel. Fails if file is not found.

ENTER(file,chan) opens file for output (writing) on software channel, returns channel. Illegal I/O error if file is not found.

If chan < Ø or > 15, illegal I/O unit error.
If chan = Ø, a preliminary OPEN('DSK') is performed, the new channel returned.
If file is not of the form
filnam [. ext] [[proj , prog]]
, a bad prototype error.
If channel is not open for operation, illegal I/O unit error.
If input (LOOKUP) or output (ENTER) side of channel already selects a file, the old file is closed.

CLOSE(chan,inhib,outhib) closes the input and/or output side of the software channel, returns null.

If outhib is non-null, the output side is not closed.
If inhib is non-null, the input side is not closed.
If chan < 1 or > 15, illegal I/O unit error.
If channel is not in use, ignore and return.

INPUT(var,chan,len)
OUTPUT(var,chan,len) create an input (output) association between the variable var and software channel chan, with line/character mode and association length specified by len, and return null.

If len > 0, line mode.
If len = 0, line mode with default length (72).
If len < 0 or not integer, character mode.
If chan > 15, illegal I/O unit.
If chan > 0 use channel table to determine I/O device.
If chan = 0 use TTY I/O.
If chan < 0 or not INTEGER, disconnect association but do not DETACH it.
If the input (output) side of the channel has not been opened, illegal I/O unit error.
If var is not a string, illegal arg.
If an association for var already exists, it is changed.
If variable is dedicated, illegal arg.

Examples:

```
INPUT('SOURCE',LOOKUP('SRCELT.SNO'),80)

OUTPUT('TYPEOUT.CHARS',0,-1)
```

The initial I/O configuration is equivalent to:

```
INPUT('INPUT')

INPUT('INPUTC',0,-1)

OUTPUT('OUTPUT')

OUTPUT('OUTPUTC',0,-1)
```

During execution, all system messages are output via the variables OUTPUT and OUTPUTC (which should always both be associated to the same channel). In order to switch system output to the printer, for example,

```
LPT = OPEN('LPT')

OUTPUT('OUTPUT',LPT,132)

OUTPUT('OUTPUTC',LPT,-1)
```

Channel Ø is never assigned, but when used in INPUT and OUTPUT associations implies the user TTY and TTCALL operation. On input, line mode reads up to (but not including) the next carriage return, line feed (CR,LF) sequence, and then these are discarded. Character mode reads only one character (including CR or LF). Line mode discards any characters beyond the association length. An EOF causes failure in either mode, but cannot occur on some devices (such as the user TTY).

On output, line mode writes out the string value with a CR, LF appended, whereas character mode does not append the CR, LF. In line mode, if the string length is greater than the association length, extra CR, LF characters are inserted every association length substring.

DETACH(var) disconnects input and output associations for the variable and detaches it from I/O processing, returns null. If the variable had no association, ignore and return.

SUBSTR(string,len,pos) returns the substring of string starting at pos of length len, and fails if len < Ø, pos < Ø, or pos + len > SIZE(string). The position convention is the same as that for patterns, and the operation is similar (but faster and less space-consuming) to:

```
string   TAB(pos)  LEN(len) . SUBSTR
```

INSERT(substring,string,len,pos) returns the new string formed by substituting substring for the one specified by the last 3 arguments into string, failing under the same conditions as:

```
string   TAB(pos) . PART1  LEN(len)  REM . PART2

INSERT = PART1 substring PART2
```

LPAD(string,len,padchr) returns the string formed by padding string on the left with padchr characters to a length of len. If string is already too long, it is returned unchanged; if padchr has more than one character, only the first is used. If the third argument is null, blanks are used.

RPAD(string,len,padchr) is like LPAD, but pads to the right.

REVERS(string) returns the string formed by reversing the order of the characters of its argument.

EXTIME(progname) causes the runtime system to output current timing statistics for the program progname and returns null, or fails if the program is not being timed.

REAL(x) is like INTEGER for reals.

EJECT() causes a page eject (form feed) to be assigned to OUTPUTC.

DAYTIM() returns an 11-character string representing the time of day (since midnight), as

HH:MM:SS.HH

meaning hours, minutes, and seconds to the nearest hundreth.

2.4.3 FORTRAN Primitives

These are predefined EXTERNAL.FORTRAN functions that, except for FREEZE, merely perform some simple arithmetic task and have integer values.

FREEZE() can be called to freeze the state of the FASBOL execution for resumption at some future date. When FREEZE is called, it exits to the monitor; the job may be SAVED, and when run again, it will start off by returning from the call to FREEZE. This is particularly useful for some applications that perform a considerable amount of initialization and wish to be able to start after that point on a repeated basis. No I/O devices (other than the console TTY) may be open at the time of the FREEZE and the call should be made from function level 0 if any timing is active.

ILT(int,int) or ILT(real,real)
[also ILE, IEQ, INE, IGE, IGT] Like LT, etc. except more efficient for dedicated variables or expressions.

AND(int,int)
[also OR, XOR, RSHIFT, LSHIFT, REMDR] perform the specified arithmetic or logical operation on their integer arguments and return the value (logical AND; inclusive OR; exclusive OR; logical right and left shift of first by second argument; remainder of integer division of first by second argument).

NOT(int) returns one's complement of its argument.

2.4.4 Library functions

These are additional library functions that add new features without changing the list of predefined primitives in the compiler. They are accessible via the EXTERNAL.FUNCTION declaration.

MEMBER(table,key) can be used to directly replace any occurrence of a table reference (i.e. table<key>), except that if the key is not already in the table, the reference fails (i.e. FRETURNs) and a new table entry for the key is not created. A normal table reference always suceeds and always creates a new entry if one does not already exist. Notice that, like a normal table reference, MEMBER() may appear on either the left or right side of an assignment.

2.5 Keywords

Three new keywords, all unprotected, have been added in FASBOL.

&STNTRACE is initially Ø, but if assigned a nonzero value it causes a trace output for each statement, giving statement number, program name, and time. This slows down execution considerably, so it is best to turn it on as close to the suspected bug as possible. Programs compiled under the NO.STNO option will ignore the value of &STNTRACE, however, so another approach is to run with all but the suspect program under NO.STNO, with &STNTRACE on all the time.

&DENSITY is initially 75, and represents the desired density of free storage immediately following a garbage collection. For example, &DENSITY = 75 means that the free storage system will try to maintain at least a 1:4 ratio between available and total storage immediately following a garbage collection, and will expand total storage as far as necessary or possible in order to try to maintain this ratio. See Section 3.2.4.

&SLOWFRAG is initially Ø, but if assigned a nonzero value it serves to switch in a heuristic in the free storage mechanism that slows down the rate of fragmentation of blocks at the expense of some wasted storage. See Section 3.2.4.

CHAPTER 3

3. FASBOL II Programming

Using FASBOL involves two separate stages, as in FORTRAN: compilation and execution. The first requires the compiler, an absolute program named FASBOL.SAV (or FASBOL.DMP, depending on the

operating system). Execution of compiled (and then assembled) programs requires a library search, during loading, of the FASBOL library; this is a collection of relocatable programs named FASLIB.REL. The relative accesibility of these programs will depend on the installation. The compiler requires a minimum of 35K to run, and requires more core in proportion to the program being compiled. The core requirements for execution of user programs depends on the size of the compiled programs plus at most 5K (if every single facility in the library is used) for the library.

3.1 Using the compiler and runtime library

To compile a FASBOL program, type

```
.RUN FASBOL n
```

where the CORE argument (n) is optional. It is best to give the compiler an amount of core commensurate with the size of program being compiled: this will increase compilation speed by minimizing garbage collections, since the compiler will expand core on its own only when it absolutely has to.

The compiler will respond with

, to which the user is expected to respond with a set of file specifications of the form

```
*macfil,lstfil_srcfil
```

Each file specification is of the standard form, as would be given to MACRO, for instance. The MACRO output file, macfil, is given a default extension of MAC if not specified. Lstfil is the listing file, and both macfil and lstfil are optional. The source file, srcfil, is given a default extension of SNO if not specified. Only one source file is permitted.

Examples:

```
*DTA3:SAMPLE,LPT:_SAMPLE
*,TAPE,LST_MTAØ:
*NEW.NEW,_TEST.NEW
```

Once the compiler produces the MACRO output file, it must be assembled, using the Q flag to supress anxiety messages from MACRO:

```
.COMPILE macfil(Q)
```

The MACRO file can be deleted after assembly, as it will be of little interest to most users; it is mainly a shortcut for the compiler to avoid having to generate relocatable code. On the other

hand, those individuals who understand the workings of the run-time system may wish to hand-tailor these intermediate programs to suit their own needs; Caveat Emptor.

To prepare any collection of FASBOL and other programs for execution, the command list should be terminated with a library search of FASLIB, for example:

```
.LOAD fil1,fil2, . . . ,filn,FASLIB/LIB
```

It is important that FASLIB be searched only once, after all FASBOL programs have been loaded, since it is very carefully sequenced to provide dummy versions of elements that are somehow referenced, but not really needed. The automatic search of the FORTRAN library should take place after searching FASLIB, since FASLIB may require some FORTRAN routines.

While FASBOL may call or may be called by FORTRAN or user MACRO programs, the main program must be a FASBOL program. Furthermore, the FASBOL runtime system enables traps and uses user UUO's 1 through 10, so it is incompatible with the FORTRAN runtime system. What this means is that FORTRAN programs used within a FASBOL execution must not do any I/O or otherwise cause FORSE. to be loaded. FASBOL does provide an infinite stack (all FASBOL stacks are infinite, up to user core limits) in register 17, however, so a broad class of FORTRAN user programs and library routines are permissible.

Unless changed by the user's program, all system output during execution is sent to the user's console; upon either error or normal termination of execution, the appropriate messages and statistics will be printed out, and control returned to the monitor. The error numbers are described in Appendix 3.

3.2 Programming techniques

Because of the basic differences between interpretive and compiler systems, and the additional features available in FASBOL, some programming techniques besides those discussed in [2], Ch. 11, are described here. An interested user may wish to get a listing of the compiler itself to see examples of some of these techniques.

3.2.1 Dedicated expressions

Dedicated expressions in FASBOL are those that are known, because of some component, to have a numerical value of a predetermined type. At one extreme is the totally dedicated statement that involves nothing but declared dedicated variables, constants, and perhaps FORTRAN calls. For example, if I were declared INTEGER, the statement

```
I = 2 * I + 10
```

would be totally dedicated, and compile into

```
MOVE 1,I
IMULI 1,2
ADDI 1,1Ø
MOVEM 1,I
```

Even if an expression is mixed, with both dedicated and descriptor-mode subexpressions, in-line arithmetic code is compiled for as much of the expression as is possible to commit to a specific type of value (i.e. INTERGER or REAL) at compile time. It is therefore to the user's advantage to declare as many variables as he perceives will be dedicated in use to be of that dedicated type. Not only will the program run faster, it may even use less core. In a situation where all entities are descriptor mode, even arithmetic operators have to check the type of, and possibly convert each argument.

In this connection it should also be noted that the predefined FORTRAN primitives ILT, ILE, IEQ, INE, IGE, IGT have been provided in order to do a much more efficient job than LT, ..., GT when the arguments are dedicated. For example, if R and S are REAL, the test

```
ILT(R,S)
```

takes up several fewer words and runs about 1ØØ times faster than the test

```
LT(R,S)
```

FASBOL permits mixed mode (INTEGER and REAL) arithmetic, the general rule being that the result of an operation is INTEGER only if both sides are integer; furthermore, an arithmetic operation involving dedicated and descriptor mode values always has a dedicated result. A value being combined with a stronger mode is first converted, and then the operation is performed in that mode; for example, if I is INTEGER but D has not been declared dedicated,

```
I + D
```

implies the value of D will first be converted to an integer, and then added to I. The only exception to this rule is the ** (exponentiation) operator which permits a REAL raised to an INTEGER power.

Finally it should be noted that whereas the range of values of dedicated variables is the same as in FORTRAN, descriptor mode integers have a range two powers of 2 less in magnitude, and descriptor mode reals have two fewer bits of precision in the mantissa. The reason for this is that the two bits are needed for the descriptor type.

3.2.2 Use of the Unary? and . operators

Unary ? (interrogation) is useful to indicate to the compiler that an expression is evaluated for its effect, rather than value. For example, a frequent occurence in SNOBOL programs is the concatenation of null-valued functions for their sequential effects and/or succeed/fail potential. If the compiler knows that an element has a null value, it does not generate code to include it in the concatenation. Therefore it is efficient to precede predicates and other null-valued elements in a concatenation with the ? operator. This technique is especially valuable when combining predicates and dedicated arithmetic, as in

```
I = ?IDENT(A,B) ?IGT(I,25) I + 1
```

, since concatenation is avoided entirely and the dedicated arithmetic is performed after the execution of the predicates without any need for conversion between dedicated and descriptor values.

Another frequent occurence in SNOBOL programs is the repetitive access of the same indirect variable, array element, or field of a programmer-defined datatype. Each of these accesses, whether to retrieve or store a value, involve some overhead which is repeated for each access. For example, in the statements

```
$X = $X + 1
```

, the variable represented by the string value of X is looked up in the symbol table twice, the first time to retrieve its value, the second time to store into it. The unary . (name) operator can be used to save the result of one lookup by creating a NAME datatype, and then the NAME can be used in an indirect reference wherever the original expression was used. Instead of the above statement a more efficient sequence would be

```
Z  = .$X
$Z = $Z + 1
```

, where Z contains the NAME of the variable pointed to by X. The same considerations apply to array references and field references as apply to indirection; it is efficient to save the NAME of the variable referenced if it will be used more than once in close succession. For example, the statements

```
A<25>       = F(A<25>)
NEXT(LIST) = NODE(VAL, NEXT(LIST))
```

would be more efficient if coded as

```
Z  = .A<25>
$Z = F($Z)
Z  = .NEXT(LIST)
```

```
$Z = NODE(VAL, $Z)
```

It should be noted that TABLE references, and the symbol lookup involved makes it even more efficient to save the NAME. The NAME of an array or TABLE element, or of a field of a programmer-defined datatype, is only valid as long as that array table or datatype exist; attempts to retrieve or store using the NAME afterwards will have unpredictable results. Also, the NAME of a variable evaluated before that variable acquires an I/O association, does not reflect that association.

3.2.3 Pattern matching

Frequently a programmer wishes to write a degenerate-type statement consisting of a concatenation of elements executed for their effect, as in

```
F(A) F(B) F(C) F(D)
```

This syntax, however, is parsed as a pattern match, and, though having the same effect as intended (providing the match is successful), is less efficient in both space and time. The original intent can best be achieved by enclosing the concatenation in parentheses, and in this case, using the ? operator

```
(?F(A) ?F(B) ?F(C) ?F(D))
```

,which will suppress string concatenation.

One particularly unique feature of FASBOL is that explicit pattern expressions, i.e., those involving the pattern operators and/or primitives, are compiled as re-entrant subroutines, rather than constructed at run-time into intermediate-language structures. The significance of this to the programmer, aside from the increase in execution speed, is the there is less of a need to pre-assign subpatterns that will appear in pattern matches later on; in fact, an unnecessary pre-assignment will be slightly less efficient because the pattern match will have to recurse one level deeper than otherwise during execution. The way to determine the need for pre-assignment is to note how much evaluation is actually required in a subpattern; if little or none is required, it can just as efficiently be included in the body of the match. Of course, if a subpattern is large and/or used in several matches, the programmer may wish to pre-assign it anyway for convenience sake. Pattern evaluation involves only the elements of the pattern, not the structure itself. Literals and other constant values do not require evaluation, so the pattern

```
TAB(7)  (SPAN('XYZ') ! BREAK(';')) $ SYM ';??'
```

requires no evaluation at all. A generally applicable rule for all FASBOL programming is that it is more efficient to use, wherever

possible, literals instead of variables with constant value. Simple variables appearing in a pattern require little evaluation (only a determination if they have a string or pattern value), and even character class primitives (i.e. SPAN, BREAK, etc.) require little evaluation, if their argument is non-literal, provided the argument is a variable with a constant value. Examples of pattern elements requiring more extensive evaluation are (non-pattern primitive) function calls, non-pattern expressions requiring considerable evaluation in their own right, and character-class primitives whose arguments are other than literals or simple variables. An example of the latter case would be

```
ANY('XYZ' OTHER)
```

; even if the value of OTHER remains constant, the concatenation produces a new string each time, which prevents ANY from immediately using the break table it has generated on the last execution of that call. It has been assumed in all this discussion of pattern evaluation that the value of the pattern element would not change value between the time of assignment of the subpattern and its use in a match. Should this not be the case, of course, the alternative of including the subpattern in the match does not exist.

Even when integer constants cannot be used, it is still helpful to use dedicated integer variables or expressions in patterns, if possible. Dedicated integer expressions are ideally suitable as arguments to the positional pattern primitives (i.e. POS, LEN, etc.), and integer variables are ideally suitable as objects of the cursor assignment operator (@). For example, suppose one wishes to take a string composed of sentences separated by semicolons (and terminated by a semicolon) and output the sentences on separate lines. A single pattern match to do this would be

(P is INTEGER):

```
        STRING   @P  SUCCEED  TAB(*P)  BREAK(';') $ OUTPUT  LEN(1)
+       @P  RPOS(Ø)
```

Note that the pattern requires no evaluation.

Since FASBOL does not employ the QUICKSCAN heuristic of assuming at least one character for unevaluated expressions, left-recursive patterns will loop indefinitely at execution time, as they would in SNOBOL4 under FULLSCAN. Usually a set of patterns involving left recursion can be re-written to eliminate it. To take a simple example, the pattern

```
P = *P  'Z' ! 'Y'
```

, which matches strings of the form 'Y', 'YZ', 'YZZ', 'YZZZ', etc., could be re-written as the pair of patterns

```
P1 = 'Z'  *P1 ! ''
P = 'Y'  P1
```

3.2.4 Timing and storage management

The TIMER option permits the programmer to monitor the operation of any (or all) separately compiled programs, and provide feedback on where the time is being spent. Initial programming of some problem can be done rapidly with not much attention being paid to optimization. It is usually the case that some small sections of a program account for a large percentage of the execution time; these are identified using the TIMER option. The programmer's time is then spent most efficiently optimizing the critical areas and ignoring the rest. Of course, after a series of optimizations, a new bottleneck will develop; the process can then be iterated until the law of diminishing returns takes hold. Finally, the TIMER declarations can be removed and the programs run in production mode.

The programmer has a large degree of control over storage management in FASBOL, which in turn means control over the space/time tradeoff that exists due to the dynamic storage allocation system (free storage). To begin with, requests from the free storage system prior to the first garbage collection (regeneration of dynamic storage) have very little overhead compared to ones subsequent to the first garbage collection. Unless there are good reasons for the contrary, the user should capitalize on this by starting his execution with approximately the amount of core he expects will eventually be required - past experience with the program is the best guide. Thus the number of garbage collections will be reduced to a minimum, and initial execution speeded up. In the absence of a core specification, the program will begin with the minimum required for loading, and will expand core as it becomes necessary, but undergoing more garbage collections.

The &DENSITY keyword is also useful in controlling the space/time tradeoff. &DENSITY may be set dynamically to any value between 1 and 100; immediately following a garbage collection, the dynamic storage allocation mechanism attempts to satisfy this value, interpreted as the percentage of total storage allocated that is in use at that time. Nothing is done unless the actual ratio is greater than the desired one, in which case core is expanded to satisfy the desired ratio, or until user core limits are reached. For example a user who sets &DENSITY to 99 is saying he wishes to keep his core size to a minimum, and is willing to pay a (rather large) premium in repeated garbage collections. On the other hand, a user who sets &DENSITY to 1 is asking for all the core he can get, in order that his program execute as rapidly as possible. It is also perfectly feasible to use a strategy where &DENSITY is set to different values at different times during execution. The initial value of &DENSITY is 75, which represents a general-purpose compromise.

If a user's application will occasionally require large contiguous blocks of storage, he may give himself 100% insurance by reserving dummy arrays of the appropriate size at the very beginning of his program. An alternative is to turn on the keyword &SLOWFRAG, which activates a heuristic which tends to slow down the fragmentation of large blocks at the expense of some wasted storage. While not 100% guaranteed, it will give the desired effect in most cases, minimizing the situation where a large block is called for, and though enough total storage is available, no contiguous area is large enough to satisfy the request.

Finally, the COLLECT primitive may be invoked at appropriate times, both to force a regeneration and also measure the amount of storage that is available.

3.2.5 FORTRAN interface

External FORTRAN subroutines, whether user-written or from the FORTRAN library, must be declared, including the number of arguments expected. A function call to the external subroutine may have any expression (except patterns) as arguments, but all but a few recognized expressions are assumed to evaluate to integers and will cause an error exit if not. Dedicated integer and real variables are passed directly to the subroutine, as they would in a call from a FORTRAN program. Also, dedicated integer and real expressions are evaluated, the value is saved in a temporary location, and this location passed to the FORTRAN routine. Finally, dedicated string variables and literals are passed to FORTRAN as a vector, the first word of which is pointed to by the calling sequence, and the FORTRAN routine may interpret it as one-dimensional array of ASCII characters packed five to a word. In addition to returning an integer or real value, the function may modify the value of any dedicated integer, real, or string variable that is passed to it. In the last case, not only may the characters be modified, but the character count may be changed by storing it in the right half of the word immediately preceding the first word of the string (array(0)). For example, suppose a FASBOL program contains the declarations

```
DECLARE('INTEGER','I')
DECLARE('REAL','R')
DECLARE('STRING','S(15)')
DECLARE('EXTERNAL.FORTRAN.FUNCTION','GETDAT(3)')
```

and the FORTRAN function GETDAT is defined as

```
FUNCTION GETDAT(INDEX,ISTR,IDAT)
COMMON IDATA(1000,4)
EQUIVALENCE (RDATA,IDATA)
DIMENSION ISTR(3),RDATA(1000,4)
ISTR(1)= IDATA(INDEX,1)
ISTR(2)= IDATA(INDEX,2)
ISTR(0)= (ISTR(0)/2**18)*2**18+10
```

```
      IDAT= IDATA(INDEX,3)
      GETDAT= RDATA(INDEX,4)
      RETURN
      END
```

, then a typical use of GETDAT within the FASBOL program might be

```
      R = GETDAT(2,S,I)
```

,which would have the effect of setting I to some integer value, R to some real value, and S to a 10-character string.

Entries that are expected from FORTRAN must be declared with the ENTRY.FORTRAN.FUNCTION declaration. This works like ENTRY.FUNCTION in that an automatic DEFINE is performed on the first call. Valid actual arguments in the FORTRAN call to FASBOL can be integers, reals, and Hollerith arrays (as described above, denoted by the codes 0,2, and 5 in the calling sequence. Upon entering FASBOL, the actual arguments are copied, and converted if necessary, into the formal arguments, which are dedicated or descriptor mode FASBOL variables. The right half of the word immediately preceding the first word of a Hollerith array is considered to be the character count, and may be modified by FASBOL if the string argument is modified. Upon return to FORTRAN (via RETURN), the function value is determined by dedicated or descriptor value in the variable corresponding to the function name, but must be integer or real. The formal argument values are copied back (and re-converted, if necessary) into the actual arguments in the FORTRAN calling sequence, thus providing a means of passing back additional values, besides the function value, to FORTRAN.

It should be noted that FORTRAN is not recursive, and therefore any recursive combination of FASBOL and FORTRAN calls will not work. Even when a series of calls is not recursive, care must be taken not to re-enter a FASBOL routine which has a FORTRAN call pending, because the FASBOL routine uses the same temporary storage locations for all FORTRAN calls, including the predefined primitives IGT, etc.

In writing FORTRAN programs to be used with FASBOL programs, care should be taken not to perform any I/O, or use any other FORTRAN facility that requires the FORTRAN runtime system (FORSE.) to be loaded.

APPENDIX 1

Syntax for FASBOL II

Explanation of syntax notation

1. All terminal symbols are underlined, the remainder of the syntax consisting of non-terminals and syntax punctuation.

2. The ::= operator indicates equivalence.

3. The | operator indicates a series of alternatives.

4. The blanks between consecutive elements indicate concatenation.

5. The \ operator indicates the specific ruling out of the immediately following element as a precondition for further concatenation.

6. The ... operator indicates the indefinite repetition of the immediately preceding element.

7. The < > brackets serve to group expressions into a single element.

8. The [ ] brackets indicate the optional occurence of the expression contained within brackets, and also serve to group the expression into a single element.

9. The order of precedence for the operators, from highest to lowest, is: \ ... (blanks) | ::=.

Syntax

```
program::= [declaration | comment]... [execute.body] end.statement
declaration::=bl DECLARE( [bl] declaration.type [bl] 1 eos
comment::= * [char]... eol | - [bl] control.type [bl] eol
execute.body::= statement [statement]...
end.statement::= END [bl label] eos
bl:= <blank | tab> [bl] | eol <+ | .> [bl]
eos::= [bl] <; | eol>
eol::= carriage.return linefeed [formfeed]...
statement::= comment | [label.field] [statement.body] [goto.field] eos
label.field::= \<END bl> label
goto.field::= bl : [bl] <goto | S goto [bl] [F goto] | F goto [bl]
        [S goto]>
goto::= ( [bl] <identifier | $ string.primary> [bl] )
statement.body::= degenerate | assignment | match | replacement
degenerate::= bl string.primary
assignment::= bl variable equals [bl expression]
```

```
match::= bl string.primary bl pattern.expression
replacement::= bl variable bl pattern.expression equals [bl
        string.expression]
equals::= bl <= | _>
variable::= \pattern.identifier identifier | & unprotected.keyword
        | string.variable
expression::= string.expression | \<string.primary
        [bl string.primary]... > pattern.expression
pattern.expression::= conjunction [bl ! bl conjunction]...
conjunction::= pattern.term [bl pattern.term]...
pattern.term::= pattern.primary [<bl . bl | bl $ bl>
        pattern.variable]...
pattern.primary::= pattern.identifier | pattern.primitive | @
        pattern.variable | [*] string.primary | sum | ( [bl]
        pattern.expression [bl] )
pattern.variable::= [*] variable
string.expression::= sum [bl sum]...
sum::= term [<bl + bl | bl - bl> term]...
term::= factor [<bl * bl | bl / bl> factor]...
factor::= string.primary [bl <** | > bl string.primary]...
string.primary::= \pattern.identifier identifier | literal | &
      < unprotected.keyword | protected.keyword> | string.variable |
      <? | \ | - | +> string.primary | . variable | (
      [bl] string.expression [bl] )
literal::= integer.literal | real.literal | string.literal
string.variable::= $ string.primary | array.element |
        procedure.call
procedure.call::= \pattern.primitive < identifier ( [bl]
        [parameter.list] [bl] )>
array.element::= \pattern.identifier identifier << | [> [bl]
        [parameter.list] [bl] <> | ]>
parameter.list::= expression [pc expression]...
pc::= [bl] , [bl]
identifier::= letter [letter | digit | . | -]...
label::= \<blank | tab | . | + | - | *> char [\<blank
        | tab | ;> char]...
integer.literal::=  digit [digit]...
real.literal::= digit [digit]... . [digit]...
string.literal::= ' [\<' \'> <'' | cont.char>]... ' | " [\<" \">
        <"" | cont.char>]... "
cont.char::= char [eol <+ | .>]...
letter::= A | B | C | D | E | F | G | H | I | J |
        K | L | M | N | O | P | Q | R | S | T | U |
        V | W | X | Y | Z | a | b | c | d | e | f |
        g | h | i | j | k | l | m | n | o | p | q |
        r | s | t | u | v | w | x | y | z
digit::=  Ø | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
char::=  any printing.character
protected.keyword::= STECOUNT | LASTNO | STNO |
       FHCLEVEL | STCOUNT | ERRTYPE | RTHTYPE | ALPHABET
unprotected.keyword::= ABEND | ANCHOR | FULLSCAN |
       STNTRACE | MAXLNGTH | STLIMIT | ERRLIMIT |
```

```
        DENSITY | INPUT | OUTPUT | DUMP | SLOWFRAG
pattern.identifier::= FAIL | FENCE | ABORT | ARB |
        BAL | SUCCEED | REM
pattern.primitive::= <LEN | TAB | RTAB | POS | RPOS |
        SPAN | NSPAN | BREAK | BREAKX | ANY |
        NOTANY> ( [bl] <string.expression | * string.primary>
        [bl] ) | ARBNO( [bl] pattern.expression [bl] )
control.type::= LIST | UNLIST | NOCODE | CODE |
        EJECT | SPACE [bl integer.literal] | NOCROSS |
        CROSREF | FAIL | NOFAIL
declaration.type::=
        'OPTION' pc <'NO.STNO' | 'TIMER' | 'HASHSIZE=
                integer.literal '> |
        'SNOBOL.MAIN' pc ' identifier ' |
        'SNOBOL.SUBPROGRAM' pc ' identifier ' |
        'PURGE.VARIABLE' pc <ALL | ' identifier.list '> |
        'UNPURGE.VARIABLE' pc ' identifier.list ' |
        'PURGE.LABEL' pc <ALL | ' label.list '> |
        'UNPURGE.LABEL' pc ' label.list ' |
        'PURGE.FUNCTION' pc <ALL | ' identifier.list '> |
        'UNPURGE.FUNCTION' pc ' identifier.list ' |
        'STRING' pc ' string.specifier.list ' |
        'INTEGER' pc ' identifier.list ' |
        'REAL' pc ' identifier list ' |
        'RENAME' pc ' identifier ' pc ' identifier ' |
        'GLOBAL.VARIABLE' pc ' identifier.list ' |
        'GLOBAL.LABEL' pc ' label.list ' |
        'GLOBAL.FUNCTION' pc ' identifier.list ' |
        'EXTERNAL.VARIABLE' pc ' restricted.identifier.list ' |
        'ENTRY.VARIABLE' pc ' restricted.identifier.list ' |
        'EXTERNAL.LABEL' pc ' restricted.label.list ' |
        'ENTRY.LABEL' pc ' restricted.label.list ' |
        'EXTERNAL.FUNCTION' pc ' restricted.identifier.list ' |
        'ENTRY.FUNCTION' pc ' restricted.identifier ( [bl]
                [identifier.list [bl] ) [[bl] identifier.list] ' [pc
                ' label '] |
        'EXTERNAL.FORTRAN.FUNCTION' pc ' fortran.identifier.list ' |
        'ENTRY.FORTRAN.FUNCTION' pc ' restricted.identifier ( [bl]
                [identifier.list] [bl] )' [pc ' label ']
identifier.list::= identifier [pc identifier]...
label.list::= label [bl label]...
string.specifier.list::= string.specifier [pc string.specifier]...
string.specifier::= identifier ( integer.literal )
restricted.identifier.list::= restricted.identifier
        [pc restricted.identifier]...
restricted.label.list::= restricted.identifier
        [bl restricted.identifier]...
fortran.identifier.list::= fortran.identifier
        [pc fortran.identifier]...
fortran.identifier::= identifier [= <INTEGER | REAL>]
        ( integer.literal )
restricted.identifier::= letter [lnd [lnd [lnd [lnd [lnd ]]]]]
```

lnd::= letter | digit | .

APPENDIX 2

Predefined symbols

1. GLOBAL and EXTERNAL variables
INPUT INPUTC OUTPUT OUTPUTC

2. GLOBAL and EXTERNAL labels
END FRETURN NRETURN RETURN

3. EXTERNAL.FORTRAN functions (all integer valued) AND(2) FREEZE(Ø) IEQ(2) IGE(2) IGT(2) ILE(2) ILT(2) INE(2) LSHIFT(2) NOT(1) OR(2) REMDR(2) RSHIFT(2) XOR(2)

4. Primitive pattern variables
ABORT ARB BAL FAIL FENCE REM SUCCEED

5. Primitive pattern functions
ANY ARBNO BREAK BREAKQ BREAKX LEN NOTANY NSPAN POS RPOS RTAB SPAN TAB

6. Predefined primitive functions
APPLY ARRAY CLOSE COLLECT CONVERT COPY DATA DATATYPE DATE DAYTIM DEFINE DETACH DIFFER DUPL EJECT ENTER EQ EXTIME GE GT IDENT INPUT INSERT INTEGER ITEM LE LGT LOOKUP LPAD LT NE OPEN OPSYN OUTPUT PROTOTYPE REAL RELEASE REPLACE REVERS RPAD SIZE SUBSTR TABLE TIME TRIM

7. Predefined library functions
MEMBER

APPENDIX 3

Runtime Errors

Conditionally Fatal

1. Illegal Data Type
2. Error in Arithmetic Operation
3. Erroneous Array or Table Reference
4. Null String in Illegal Context
5. Undefined Function or Operation
6. Erroneous Prototype
7. Dedicated String Overflow
8. Variable Not Present Where Required
9. Real to String Conversion Overflow
10. Illegal Argument to Primitive Function

11. Reading Error
12. Illegal I/O Unit
13. Limit on Defined Datatypes or Tables Exceeded
14. Negative Number in Illegal Context
15. String Overflow

Unconditionally Fatal

17. Error in FASBOL System
18. Return from Zero Level
19. Failure During Goto Evaluation
20. Insufficient Storage to Continue
21. Illegal Memory Reference
22. Limit on Statement Execution Exceeded
23. Object Exceeds Size Limit
24. Undefined or Erroneous Goto
25. [unused]
26. [unused]
27. Writing Error
28. Execution of Statement with Compilation Error
29. Failure Under NOFAIL
30. Divide Check
31. Arithmetic Overflow

FILEX

The FILEX program is a general file transfer program intended to convert between various core image formats, and to read and write various directory formats. It is primarily useful for DECtapes.

A writeup on FILEX can be found in the DECSYSTEM10 Users Handbook, second edition, page 557.

Following are the only three command strings which will work for 11 format tape:

1) to put files A.A,B.B,C.C on 11 format tape:
```
*DTA1:(ZQV)_DSK:A.A,B.B,C.C
```

2) to read the whole 11 DTA onto DISK:
```
*DSK:_DTA1:*.*(QV)
```

3) to list a directory from an 11 format tape:
```
*_DTA1:(VL)
```

WARNING:

Since FILEX was written for the DECsystem10 Monitor, not all forms of the command string will work under TENEX.

FIOCNV

Program to convert standard TENEX text files (ASCII) to punched paper tape for either flexowriter (FIO-DEC code) or Dura typewriter. FIO stands for Flexowriter Input Output.

@FIOCNV

Asks all necessary questions including name of source file and format of output. Prints message for any character not translatable into object code set.

@FIOCNV

```
TO OR FROM PAPER TAPE? (T-F) T
INPUT FILE: WIN.MOVES;1%

FLEXO OR DURA? (F OR D) D
CAN'T CONVERT ^^ (36)

INPUT FILE:
```

FLIST

FORTRAN LISTING

FLIST copies FORTRAN-generated ASCII disk files to the line printer or another disk file. The first character (column 1) of each ASCII record is not printed, but is interpreted as a carriage control character according to the DECsystem-10 FORTRAN Standards. Characters other than those listed have the same effect as a space in column 1.

Character		Effect
	space	skip to next line with a FORM FEED after every 60 lines
0	zero	skip a line
1	one	form feed-go to top of next page
+	plus	suppress skipping - will overprint line
*	asterisk	skip to next line with no FORM FEEDS
-	minus	skip 2 lines
2	two	skip to next 1/2 of page
3	three	skip to next 1/3 of page
/	slash	skip to next 1/6 of page
.	period	skip to next 1/20 of page
,	comma	skip to next 1/30 page

FLIST also has the option of replacing all line feed characters with a DC3 character, which has the properties of a line feed character but inhibits the automatic FORM FEED after every 60 lines.

FLIST asks for the name of the file to be listed as shown below. TENEX file name recogniton is in effect. A confirming carriage return must follow the file name. FLIST then asks if the output is to go to the lineprinter (LPT:) or a file (filename). A confirming carriage return must follow either the LPT: or the file name. FLIST then asks if the DC3 option is to be activated.

EXAMPLE:

```
@FLIST%

INPUT FILE: TEST.DAT% [CONFIRM]%

OUTPUT FILE (LPT: OR FILENAME):TST.LST [CONFIRM]%

CONVERT LINEFEED TO DC3 (Y OR N)?Y
INPUT FILE:  TEST2.DAT;1 [CONFIRM]%


OUTPUT FILE: LPT: [OK]%

CONVERT LINEFEED TO DC3 (Y OR N)?N
INPUT FILE: ^C
@
```

NOTE: Output disk files which have used the DC3 option should be listed on the lineprinter with the COPY command instead of the LIST command.

FLOW

FLOW is an automatic flowcharting program which produces a flowchart from a FORTRAN source file.

This subsystem was originally distributed by Digital Equipment Corporation as DECUS 10-38 and has been modified by BBN.

FLOW requests an input file name. This should be answered with the name of a FORTRAN source file on the DSK. (A maximum of 5 characters is allowed.) It then requests the input file extension and an extension of up to 3 characters should be supplied. An output file name is requested and a file name of up to 5 characters should be supplied for the DSK output file. An output file extension is then requested and an extension of up to 3 characters should be supplied.

During processing, a binary scratch file FOR20.DAT will be created on the DSK for each main program and subroutine. FLOW will delete FOR20.DAT after it has been processed. The ASCII output file will consist of a program listing of the main program and each subroutine/function with each associated flow chart followed by a list of all statement numbers used by the associated main program or subroutine/function.

```
@FLOW

OUTPUT BINARY SCRATCH FILE ON DSK:FOR20.DAT
BEGIN EXECUTION



INPUT FILE NAME (5 CHARS)=FLOW%
INPUT FILE EXTENSION (3 CHARS)=F4%
OUTPUT FILE NAME (5 CHARS)=FLOW%
OUTPUT FILE EXTENSION (3 CHARS)=LST%
FLOW-CHARTING PROGRAM MAIN
FLOW-CHARTING PROGRAM PRNT
FLOW-CHARTING PROGRAM GORT
FLOW-CHARTING PROGRAM ASSIGN
FLOW-CHARTING PROGRAM MOVE
FLOW-CHARTING PROGRAM REPT
FLOW-CHARTING PROGRAM NUM
FLOW-CHARTING PROGRAM NUMAL
```

```
FLOW-CHARTING PROGRAM BLOCK DATA
FLOW-CHARTING PROGRAM ERROR
FLOW-CHARTING PROGRAM I REFIN
FLOW-CHARTING PROGRAM SORT
FLOW-CHARTING PROGRAM XTRACT
FLOW-CHARTING PROGRAM TABFIX
FLOW-CHARTING PROGRAM TABPAK



     END OF JOB

CPU TIME: 4:0.96     ELAPSED TIME:  17:10.00
NO EXECUTION ERRORS DETECTED

EXIT.
^C
```

Note: Before doing FLIST of output file put WIDE paper in the Line Printer.

```
@FLIST%
INPUT FILE: FLOW.LST%[CONFIRM]%

OUTPUT FILE (LPT: OR FILENAME):LPT:%[OK]%

CONVERT LINEFEED TO DC3 (Y OR N)?Y
INPUT FILE: ^C%
@

EXECUTION TIME:          9.88 SEC.
TOTAL ELAPSED TIME:      2 MIN. 18.00 SEC.
NO EXECUTION ERRORS DETECTED
EXIT.
^C
```

FORTRAN System

```
# The DEC FORTRAN-IV Compiler, for further information see
# DECsystem10 FORTRAN-IV Manual.
#
# To obtain the FORTRAN compiler call F40.
#
# COMPILING AND RUNNING A FORTRAN PROGRAM
#
# It is assumed that the user has a FORTRAN program on a file in a
# card image format (one FORTRAN statement per line with the
# statement body starting in column 7 or greater as usual. Lines
# may be up to 80 characters terminating with a carriage return
# line feed). In the examples below we will call the FORTRAN
# program file X.
#
# The FORTRAN compiler, F40, takes only one command of the form
#
# object-file, listing-file _ input-file1,input-file2,...
#
# All file names may include a device, but not a version number.
# Furthermore, the name part is limited to 6 characters and the
# extension (if any) to 3 characters.
#
# The object-file is the relocatable binaries output by the
# compiler, and will assume a default extension of .REL if no
# extension is given.
#
# The listing file is for the compiler listing (error messages will
# also appear on the user's terminal as compilation proceeds), and
# will assume a default extension of .LST if no extension is given.
#
# As shown, there may be several input files separated by commas.
# The compiler will first look for an extension of .F4 (not .F40),
# and failing to find that, look for a file with no extension.
#
# EXAMPLES:
#
#    To get both relocatable binaries and listing:
#
#      @F40
#      *X,X_X
#
#         The first X will be X.REL, the second X.LST
#         The last X could have no extension or an extension of .F4.
#
#    To get just relocatable binaries with no listing:
#
#      @F40
#      *X_X     The first X will be X.REL
```

THE LOADER
#
After the FORTRAN programs are compiled, the relocatable binaries output by the compiler must be loaded along with required routines from the FORTRAN library. Assuming the output from F4Ø is on X.REL, this is done by:
#
#
@LOADER
*/SX$ ($ represents the escape character - it causes
the load process to be completed with a search of
the FORTRAN library and a return to the EXEC)
#
If several relocatable binary files representing the separate compilations of the main program and subroutines of a system are to be loaded together (say X.REL, Y.REL, and Z.REL), then do:
#
@LOADER
*/SX,Y,Z$
#
The '/S' is not required, but is desirable if the program is to be debugged via DDT. It causes the Loader to leave the user's FORTRAN symbols (variables and statement numbers) in core for use by DDT. If no debugging is to be done, the /S may be left off, but its use is encouraged.
#
SAVING THE ABSOLUTE CORE IMAGE
#
The Loader has no "output file". Instead, everything is loaded into core and left there. Thus after the escape ($) completes the load and causes the exit to the EXEC, it is necessary to save the assembled core image (if repeated executions are desired - if they are not, the program may be immediately started, but then the load process must be repeated for another execution). This is done by:
#
@SAV$$$X$ ret (the $ is escape so there are four escapes
here altogether)
#
This will echo as
#
@SAVE (CORE FROM) 2Ø (TO) 777777 (ON) X.SAV;1 [NEW FILE]
#
and commands that the entire core image be saved on file X.SAV (which is here assumed to be a new file - a new version would be created if an earlier version of X.SAV already existed).
#
The program may then be started by
#
@START ret

and may be repeatedly executed by simply typing
#
@X ret

NOTES ON DEBUGGING FORTRAN Programs via DDT
#
These notes are not a complete explanation of DDT. it is assumed that the reader has at least glanced at the DDT manual and that he has a copy convenient for reference. it is also assumed that the reader knows how to compile, load, and run a FORTRAN program.
#
We use the following conventions below:
#
$ represents the altmode or escape character.
#
^x represents control-x, e.g., ^C is control-C.
#
lf represents line-feed.
#
ret represents carriage return.
#
No convention is established for distinguishing user type-ins from the system's response. Most DDT commands are short and the system will respond as soon as the command is typed (that is, no carriage return is necessary with DDT commands). Thus, the novice user should be able to get started by typing slowly and waiting for the system's response.
#
PREPARATION FOR DEBUGGING - THE LOADER
#
Although it is possible to get an assembly language listing from the FORTRAN compiler and a load map from the Loader, this is not usually necessary in debugging a FORTRAN program. It is however necessary to instruct the Loader to save the symbol table that is automatically output by the compiler, so that the user's symbols will be in core and available for reference after the loading process. This is done by using the /S switch during loading. Example: assume the user intends to debug program PROG. The Loader command would then be
#
@LOADER
*/SPROG$ (the $ is the altmode)
#
After the load is complete, the usual TENEX SAVE command is executed, and whenever the program is loaded into core to be run, the user's FORTRAN symbols will also be brought in for debugging.
#

USER AND OTHER SYMBOL NAMES

In addition to the user's symbols (that is, his variable and subprogram names), there are several types of symbols created by the compiler which the user will need to reference:

1. The user's FORTRAN statement numbers will all be suffixed by a 'P'. Example: if the following statement appears in the program,

```
110 A = B
```

   then the machine instruction to load B will be labelled with the symbol 110P.

2. The main program is always labelled MAIN. (the . is included as part of the name), and all FORTRAN programs start at location MAIN. .

3. When the compiler needs to generate a label for a jump instruction, it uses the form nM where n is an integer starting at 1 for the first label generated. Examples: 1M, 3M, 17M, etc. In particular, the first executable statement of any FORTRAN program (not containing arithmetic statement functions) will be labelled with '1M' (if not labelled by the user).

GETTING IDDT AND THE RUNNING FORTRAN PROGRAM TOGETHER

There are two approaches - IDDT and DDT. The best is to use IDDT. IDDT contains many useful features including the ability to interrupt a running program and then restart it (as opposed to break-pointing it as required by the old DDT). Documentation for IDDT is available in the TENEX Users' Guide.

To use IDDT, the user may do one of two things: he may first run his program and when it terminates invoke IDDT, or he may start IDDT, use it to load the program via the ;Y (for Yank) command, and then start it with $$G (two altmodes, G), possibly after setting breakpoints.

Example of the first method (assume program named PROG):

```
@RUN PROG
```

(assume program fails and the system reports errors on the TTY and returns to the EXEC; or alternatively, the program stopped for TTY input and the user interrupted it with ^C.)

```
#         @IDDT
#              The user is now in IDDT in the context of the program
#              just run, and he may proceed with the commands described
#              below.  If the program was interrupted with ^C, it may be
#              continued with $P (altmode, P for Proceed).
#
# Example of the second method:
#
#         @IDDT
#
#         ;Yank file: PROG.SAV$ ret
#
#
#         MAIN.$:      11Ø P$B      $$G
#
#              (If the program requests TTY input before reaching the
#              breakpoint, it may be typed in normally - if a user's
#              program is waiting for input there is no confusion over
#              whether the input goes to the program or to IDDT.)
#
#         $1B>>11Ø P
#
#              (IDDT signals the break as shown and waits for commands.
#              The user may now examine variables, insert or remove
#              breakpoints, and continue the program from the break or
#              from some other location, etc., as described below.)
#
# EXAMINING FORTRAN AND OTHER VARIABLES
#
# Indicating the Routine to be Debugged
#
#      We must first consider the establishment of the proper context
# when referring to variable (or any other kind of) names. Consider a
# program that consists of a main routine and a subroutine called SUB.
# The compiler will automatically give the name MAIN. to the main
# routine (including the period). The command for establishing a
# routines symbols as the current context is of the form 'name$:' (
# routine name, altmode, colon). Thus, if we wish to debug the main
# routine we first type
#
#         MAIN.$:
#
# To then debug the subroutine we type
#
#         SUB.$:
#
# Thus we may refer to the symbols of several different routines by
# first establishing the context as that routine with the colon command.
#
#      The colon command is not always absolutely necessary. If
# variable X exists in the main routine but not in routine SUB, and if
```

we have issued SUB$:, and we then type 'X/', the system will respond
with
#
X/[_MAIN.$:X] ...
#
This does not change the context, but shows us that there is no X in
the current context. Note that if several different X's exist outside
of the current routine, then the colon command may be necessary to
refer to the desired one.
#
Typing Out a Location and the Current Type-out Mode
#
In general, typing any location name followed by a slash (/) will
cause IDDT to type out the contents of the location in the current
type-out mode. This mode is initially symbolic. Thus, typing '110P/'
will cause IDDT to type out the instruction which starts statement
110. For variables, however, we would like the type-out to be
numeric. The mode may be set permanently to floating point by '$$F'
(two altmodes followed by 'F'), or temporarily by '$F'. (The floating
point mode will also type integers correctly.)
#
Examples:
#
110P/ MOVE 2,A ret
X/ MOVEI 11,@146315(3) $F; 1.2000 lf
X+1/ 1.3000
#
$$F X/ 1.2000 lf
X+1/ 1.3000 ret
110P/ 17196647924. $S; MOVE 2,A
#
In the first three lines, the user first opens location 110P which
types out symbolically. Then X is opened which also types out
symbolically, and then the altmode-F-; is used to show the
corresponding numeric interpretation (the '; ' - semicolon, space -
command causes IDDT to retype the last value typed, presumably in a
new mode). Finally a line feed causes the cell after X to be opened
(assume X is an array), and typed out in the temporary mode. This
temporary mode will continue until a carriage return is typed, after
which the mode will revert to the current permanent mode. Thus,
multiple line feeds will continue in the temporary mode, or additional
cells may be opened by name in the temporary mode:
#
$FX/ 1.2000 Y/ 16.
#
In the second example above, the user set the mode permanently to
floating point. then, when 110P was opened, its floating point
representation was converted back to symbolic with the '$S' command.
as with F, '$$S' would be used to set the type out mode permanently to
symbolic.

Examining Related Locations

Once a word has been typed out, related words may be examined by:

lf — line feed examines the next word.

^ — up arrow (shift N) examines the previous word.

/ — examines the word specified by the address of the current word, but does not move the location pointer thus, lf and ^ are still relative to the first original word).

tab — (control-I) changes the location pointer to the word specified by address of the current word, and types out the contents of this new current word (especially useful for examining array parameters - see below).

Examining Arrays Passed as Parameters to Subprograms

Assume we have a main program with an array X, and that the array is passed as parameter Y to subprogram SUB. That is, we have

```
REAL X(100)
   .
   .
CALL SUB(X)
   .
   .
SUBROUTINE SUB(Y)
REAL Y(100)
   .
   .
```

The following shows how the user may examine the contents of X in floating point format while in the subroutine:

```
SUB$:   Y/   JUMP MAIN.X  tab
MAIN.X/   MOVEI 11,@146315(3)   $F; 1.2   lf
MAIN.X+1[   1.34   lf
MAIN.X+2[   6.21
  .
  .
  .
```

As may be seen, when the user opened Y within the subroutine, he got not the contents of array X, but the address of array X (for simple scalar variables passed as parameters, examining the parameter gets the contents of the actual parameter as it is used within the

subprogram). These array parameter addresses are always of the form 'JUMP ADDRESS', and in fact, seeing a parameter of that form is a reliable way of telling that a routine has been passed an array as opposed to a scalar variable or expression. To return to the example, the user then moved the location pointer to the array X by using the tab command (which moves the pointer to the address specified by the word last typed, in this case the MAIN.X). He then changed the type-out mode to floating point and continued to examine successive words of the array with successive line feeds.

Value of LOGICAL Variables

Logical variables set to the values .TRUE. and .FALSE. have the values -1 and 0 respectively in core.

BREAKPOINTS, PROCEEDING, AND OTHER CONTROL

We now present a nonsensical example of a FORTRAN program in both the FORTRAN form and as typed out by IDDT. The example will be used throughout the discussion of breakpoints and other control commands.

```
@COPY TST.;3 (TO) TTY: [OK]

      REAL A(10)
      READ(-1,60100) A,B
60100 FORMAT(11F10.0)
C
  110 IF(A(1).EQ.B) GO TO 120
      B=1.2
  120 CALL SUB(A,B)
      END
      SUBROUTINE SUB(X,Y)
      REAL X(10)
      END

@LOADER
*/STST$
LOADER 2K CORE
5+4K MAX 522 WORDS FREE
EXIT.
^C
@SAVE (CORE FROM) 20 (TO) 777777 (ON) TST.SAV [New version]
@IDDT

MAIN.$:   1M/   MOVEI 1,60100P          1f
1M+1/   16040,,-1                  1f
1M+2/   25100,,A                   1f
1M+3/   JUMP 12                    1f
1M+4/   20100,,B                   1f
1M+5/   21000,,0                   1f
60100P/   JRST 2M                  1f
```

```
#       60100P+1/   ROT 10,@143142(6)   $T; (11F1      1f
#       60100P+2/   0.0)                ret
#       1f
#       2M/   MOVE 2,B                  1f
#       2M+1/   CAMN 2,A                1f
#       2M+2/   JRST 120P               1f
#       2M+3/   MOVE 2,CONST.           1f
#       2M+4/   MOVEM 2,B               1f
#       120P/   JSA 16,SUB              1f
#       120P+1/   JUMP 2,A              1f
#       120P+2/   JUMP 2,B              1f
#       120P+3/   JSA 16,EXIT           1f
#       MAIN./   15000,,0               1f
#       MAIN.+1/   JRST 1M              1f
#       CONST./   MOVEI 11,@146314(3)   $F; 1.1999999     ret
#       1f
#
#
#       B/   0              1f
#       A/   0              1f
#       A+1/   0            1f
#       A+2/   0            1f
#       A+3/   0            1f
#         .
#         .
#         .
```

We first note several things about the IDDT output:

1. The "instructions" from 1M+1 to 1M+5 are system calls to perform the READ statement. In general, the user can expect to find similar code wherever I/O is done.

2. FORMAT statements and any other text in the user's program, when typed out in instruction mode will appear as strange looking instructions with unusual address as shown at 60100P+1. Note the jump instruction around the FORMAT (JRST 2M), and also note the use of the '$T; ' command to temporarily change the type-out mode to text at 60100P+1. This mode was continued for one word with a line feed, after which a carriage return returned us to the original mode.

3. The instruction labelled 2M is in fact the first instruction of the statement numbered 110, and if the user typed '110P', IDDT would have opened the same cell. This illustrates that there may be several labels which all refer to the same location (the user's plus additional labels generated by the compiler). This also occurs with variables at times, where the compiler may assign a name like '%TEMP.' to a variable which the user has called 'B', for example. If that had been

the case here, then the instruction at 2M+4 might have typed out as 'MOVEM 2,%TEMP.' instead of 'MOVEM 2,B'. '%TEMP.' and 'B' would then represent the same location, and typing out either would show the same contents.

4. Subroutine calls are of the form 'JSA AC,ADR' as shown at 120P The JSA subroutine jump instruction saves the AC (accumulator) at the ADDRess, and jumps to the ADR+1 to start the subroutine. thus, the first executable instruction of out subroutine is not at SUB but is at SUB+1. The first few instructions of the subroutine do initialization and are not usually of interest to the user. After the initialization there will be a jump to a label 1M which usually labels the first executable FORTRAN statement of the subroutine.

   Following the JSA instruction is the parameter list where each parameter's address is part of a JUMP instruction. These JUMP instructions are never executed because the subroutine will return to the instruction following the last JUMP. This shows why variable length calling sequences to subprograms may not be used with DEC FORTRAN since the return point is always based on the number of parameters in the SUBROUTINE or FUNCTION statement, not on the number in the call.

5. The first instruction of the main program is at label MAIN. This is a system call for initialization, followed by a jump instruction to the first executable FORTRAN statement (the JRST 1M instruction). This first statement will always have the label 1M for all programs (main, subroutine and function), unless the program starts with arithmetic statement functions. In that case, the first executable statement may be at 2M, or 3M, etc. It can be found either by typing out instructions starting from MAIN. (or from the subprogram name) looking for the JRST, or by typing out instructions starting at label 1M until the first executable statement is recognized.

6. Following the code is space for constants and variables. Note the use of the '$F;' command to examine the constant at CONST.

We now give some of the breakpoint and other control commands, and then describe with reference to the example how they might be used. A breakpoint is a way of stopping a program at a particular location. When the execution reaches the breakpointed location, IDDT regains control (before the instruction at the breakpoint is executed), and types something of the form

```
$nB>>adr
```

where 'n' is a number from 1 to 8 and 'adr' is the address of the breakpoint. The user may have up to 8 breakpoints at any one time.