SIX12 User's Manual*

T. Lane
R. K. Johnsson
C. B. Weinstock
Wm. A. Wulf
August 10, 1973

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

This document is an introduction to SIX12, an extensive debugging aid for Bliss programs on the PDP-10. It applies to the version of August 9, 1973; the program released in Nov. 1972 under the same name has only an ancestral similarity.

---

*This research was partially supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

Contents

Introduction 1

A Philosophy for Using SIX12 4

SIX12 Expressions 6

Basic Bliss 10

Peeking and Poking 12

Breakpointing, Tracing 14

Getting In and Out of SIX12 17

Concerning DDT 19

Console Input Monitoring 20

Monitoring Variables 21

Macros, symbol definition 22

Line printer and disk I/O 24

Display 26

Miscellaneous operators 29

Using SIX12 30

Defining your own operators 32

A Compressed Version 35

Introduction:

SIX12 is a specialized debugger adapted to the Bliss environment. It is not intended as a substitute for DDT; the user is expected to load both with his program and use whichever is more convenient to the need of the moment. Under this assumption there has been no attempt to duplicate functions already well-performed by DDT, such as

1) breakpointing and tracing at the instruction level,

2) symbolic typeout and typein in any mode required (at last count DDT could display or accept values in over a dozen different modes).

SIX12 is oriented to the Bliss programming environment in two ways:

1) It conforms to Bliss philosophy and notation. A debugging command is syntactically an expression, which is read in, evaluated, and the value (if any) printed. However, some modification has been made to the Bliss syntax in the interests of flexibility. In particular, an expression is not always required to yield a fullword value. It may yield no value, a word, or a vector value (several words). Also the syntax accepted for operators has been expanded to include nullary operators, which have no operands, and postfix operators which follow their operand. For example,

A AND 7	uses the infix operator "AND"
.A	uses the prefix operator "."
GO	uses the nullary operator "GO"
137/	uses the postfix operator "/"
BREAK R1,ERROR,PRINT	uses the prefix operator "BREAK", which accepts a vector operand. A higher-priority infix operator "," is used to form the vector from elementary operands.

(As demonstrated in the last example, with this syntax we can get by without any keyword forms. To SIX12 everything is an operator or operand. The user may find it more convenient, of course, to visualize commands formatted in this way as a keyword plus list of arguments.)

2) SIX12 is routine-oriented. By this we mean that breakpointing, tracing, and similar functions dealing with flow-of-control, all occur at the abstract routine level rather than at the individual machine instruction level (as

in DDT). That is, the smallest unit of code that SIX12 can see is a routine (or FUNCTION). Breakpoints must always be set at either the entry or exit of a routine, and tracing of execution is always in terms of routine calls and returns. DDT is used when code must be dealt with below the routine level.

Advantages of SIX12 over DDT:

1) Source language debugging - SIX12 accepts Bliss-like notation, and does tracing and breaking in a form easily relatable to the original source program.

2) Facilities that DDT simply does not have, for instance

   a) the ability to monitor data (not instruction) locations and report when they are modified,

   b) the ability to interrupt the program on console input. If any character is typed while the program is executing (i.e., when not in TTY input wait), SIX12 will intercept control and wait for completion of the debug command.

3) Extendability. SIX12 permits easy definition of new operators; in addition simple macros can be defined and used in expressions.

4) SIX12 can be used to debug in shared high-segments. DDT breakpoints cannot be used in shared code, but the SIX12 linkage has been designed so that two or more people can be debugging the same or different routines, and still others running without using the debugger, all in the same high-segment without interference.

Disadvantages:

1) No access to program at machine code level.

2) Limited variety of modes for symbolic typeout or typein of values.

3) Speed, or rather lack of it. SIX12 requires some computation at every routine call or return, while DDT requires time only at breakpoints (and less time at that).

4) Space required. SIX12 uses about 4.5K in the high segment

and 1K in the low segment (in the standard version), besides the 2K low-segment space for DDT, and the program-dependent low-seg space for the symbol table (which, however, is required even for DDT).

The first two of these are alleviated by the ease with which one can pass from SIX12 to DDT and back. Thus the full facilities of DDT are still available to the SIX12 user.

A Philosophy for Using SIX12

As you read the following description of SIX12 you will find that it contains, except for syntactic differences, most of the features that one would expect in any debugging tool - the ability to display and/or modify memory, the ability to suspend program execution at selected points, the ability to trace execution, etc. You may, if you wish, use SIX12 in a manner analogous to the way that other debuggers, e.g. DDT, have been used. This is not, however, the optimal way to use SIX12.

In principle a language-specific debugger can be far more powerful than one which works exclusively at the machine-language level. A FORTRAN-specific debugger, for example, can have specific knowledge about the format of data structures, the nature of subroutine calling conventions, the nature of register conventions, etc., for the code generated by the FORTRAN compiler. In some sense the "power" of a debugger is directly related to the number of assumptions it can make about the run-time representation of programs.

In the case of assembly language programs, and to a lesser extent Bliss programs, there are relatively few such assumptions which a debugger can safely make. In order to write systems programs both assembly language and Bliss allow the user much more freedom to specify the program structure than do languages such as FORTRAN. The response made to this dilemma by DDT, for example, is to make no assumptions - and then do the best it can from there.

SIX12, being Bliss-specific, can make a few more assumptions that DDT does -- concerning the nature of calling conventions, for example. Beyond that, SIX12 is based on quite another philosophy -- namely, the user knows.

The thrust of SIX12 is to provide a reasonable, interactive interface between the user at a terminal and his program in the machine. It is also intended that the user to able to exercise portions of his program interactively, most likely in a bottom-up fashion, during the initial testing phases of the program.

In order to realize these intentions SIX12 provides two main functions and several supporting ones: the main functions are those of symbol table searching, terminal i/o, etc. These will be explained below - for the present we are concerned with the parsing and evaluation of terminal input. The parser is fixed but quite flexible and probably should not be changed by the user. The evaluation of the input, however, is defined by normal Bliss routines. Initially a set of routines is supplied which defines a Bliss-like evaluation. However, the user may easily augment and/or

replace these routines to provide program-specific debugging facilities.

Thus, the recommended mode of use of SIX12 is for the user to supply a set of program-specific debugging routines. These routines might, for example, provide displays of elaborate data structures, check the consistency of a data structure, provide test environments, etc. The user then uses SIX12 as an interface to these routines and to his program itself.

SIX12 expressions:

SIX12 contains a fairly intelligent syntax analyzer/evaluator which can evaluate quite complex expressions. For instance, all of the following are legal Bliss expressions; they are also legal in SIX12 and produce the same result.

```
.ALPHA
SUM ← .SUM + 3
NOT .FLAGS<1,1>
MYROUTINE (7, .B, XYZ())
```

The syntax analyzer recognizes two classes of objects: operators and operands. The meaning of operators is not built into the analyzer, but is defined by a table of evaluation routines. This makes it simple to add new operators or redefine old ones; this can even be done at runtime if the necessary routines were compiled and loaded separately from SIX12. (The methods for defining your own operators are discussed later; we assume everywhere in this paper that you have not tampered with any standard operators.) The meaning of operands, however, is built into the analyzer. The possible types of operand are

<number> <string> <symbol>

A number is a sequence of digits, possibly preceded by a number sign . Its value is the equivalent binary integer. The number is assumed to be written in the default radix or base, which can be set or examined by one of the standard operators (BASE). The escape character # is provided to ease the use of two radices: a number preceded by # is taken to be written in octal radix regardless of the default radix, unless the default radix is octal, in which case the number is taken as decimal. For example,

if the default radix is	this input	has this value (octal)
10 (decimal)	34	000042
10 (")	34	000034
2 (binary)	10110	000026
2 (")	34	000034
8 (octal)	34	000034
8 (")	34	000042

Thus when the default base is decimal, this is the same as the Bliss source convention. When SIX12 is initialized, the default base is octal (as with DDT).

Strings are input as either

'string' or "string"

corresponding to left- and right-adjusted ASCII strings respectively. In either type of string the other string delimiter can appear freely, and double occurrences of the string delimiting character are used to denote it once. This is precisely the same as the Bliss convention. However, note the following differences:

? is not implemented as an escape character;
Carriage return cannot appear in a string; it terminates the string just as if the matching delimiter had been encountered (CR or LF always terminate debug input expressions);
RADIX50, ASCIZ and SIXBIT stringtype converters have not been implemented. They could be easily included by defining appropriate operators.

Examples:

```
'HI!' "Q"       'a CR or LF terminates me anyway
```

A long-string, as in the last example, generates a vector value.

Symbols are the most complex type of operand. A symbol is looked up in the Loader-generated symbol table, and its value is the value entered for it in the table. Thus a name is an address, as in Bliss. (We get around the fact that a name should be a byte pointer by some monkeying with the definition of . (contents) and ← (store). The gory details come later on.) DDT has a complex search convention to deal with the problem of multiply defined symbols (which exists because the table only contains six characters of a name). SIX12 uses this much simpler rule: identical symbols are implicitly numbered in the order in which they were loaded. If you want other than the first one, the name is suffixed by %n, where n is the ordinal you want. Examples:

```
HELP            gives value of (first) symbol 'HELP  '
LOTSOFME%5      gives value of fifth symbol   'LOTSOF'
```

There are symbol table searchers included in the standard set of operators to help you decide which ordinal you want, or remember what name you need in the first place. See Display.

Note: since names (letter-digit strings) can also refer to operators (e.g., AND OR) the % convention is also used to distinguish symbols from operators. An unadorned name is first searched for in the table of operators, then in the symbol table, taking the first match in the

process. However, name%n is only searched for in the symbol table. Thus 'OR%1' always gets you the symbol OR but 'OR' is taken for the operator OR.

Further note: The only source of program symbols that SIX12 has is the Loader symbol table (also used by DDT). Therefore, locals, formals, structures, and so on are unknown to SIX12. Only globals, owns, routines (or functions) and module names will be present in the table.

We have now fully discussed operands, and turn to the operators which work upon them. Operators are denoted either by names (e.g., GEQ, BREAK) or by special characters (e.g., $ + ↑). In the latter category, ' " % space tab CR LF may not name operators.

The syntax analyzer distinguishes four syntactic types of operators, namely

Nullary, having no operands:		<operator>	
Prefix, preceding its operand:		<operator>	<operand>
Postfix, following its operand:	<operand>	<operator>	
Infix, having both operands:	<operand>	<operator>	<operand>

The same symbol may represent more than one operator in different parses. For example

```
+ E
E + F
```

show "+" in prefix and infix parses; these two instances will actually result in the invocation of two different routines to evaluate "+". In theory the same symbol could be given all four parses, invoking 4 different evaluation routines; in practice this is a poor idea. An operator with more than 1 parse can introduce ambiguity: the classic case is with "/", which in the standard definition has postfix (A/ prints contents of A, as DDT would) and infix (A/B does division) parses. Now, does

```
A/-N
```

invoke postfix "/" followed by infix "-", or prefix "-" followed by infix "/"? With the definitions given above the second is clearly the correct choice, but the analyzer can hardly be expected to know that. In point of fact this will be evaluated in the first way, due solely

to an arbitrary design decision. The moral of this story is that the analyzer cannot be expected to always do the right thing. It works fine for pure Bliss expressions; a key thing is to be wary of using expressions which yield a null value (as x/ does) in larger expressions. The analyzer assumes that operators will yield a value, and gets confused when they don't (since its assumptions about the parsing of subsequent operators must be junked). If an expression with side-effects blows up, always check to see how much of it had already been evaluated. Two good rules to follow are 1) parentheses can fix lots of things, and 2) avoid semicolons - you can afford to do it on two lines.

With these words of warning we pass to a description of the standard set of operators. This is the meat of what you can do with SIX12.

Basic Bliss:

We do not pretend to have implemented a Bliss interpreter. However, a fair subset of the simple expressions (not control expressions) is available, and more could be implemented if you need it. The following operators are defined exactly as in Bliss:

+ -	for example -E1 or E2+E3
* /	as A*B (beware of E/ , which is not an error)
↑	as A↑B (shift)
AND NOT OR	logicals
GEQ thru LSS	the relationals

Parentheses ( ) also behave as in Bliss: as grouping indicators (arbitraryexpression) or as routine callers R(elist) . For instance

```
5*(.ALPHA-1)
MYROUTN(41, .PARM)
```

The value of the second expression is the value returned by calling the routine MYROUTN with two actuals.

Angle brackets < > have the same meaning, of creating a byte pointer. However, index and indirect fields are not accepted; there must be exactly two values within the brackets, i.e.

```
addr<pos,size>
```

is the only allowable syntax for them.

Dot . performs the same function, 'contents', and left arrow ← the same function, 'store', as in Bliss. For instance

```
.A1
.FLAG<17,1>
SE ← .SE * 200
```

You should be aware, however, of this difference. SIX12 evaluates names as addresses, not byte pointers, so that the left half of a name's value is normally zero. The motive for this is that we can then display the addresses corresponding to program names without trashing things up with a nonzero left-half. (Also, we can't assume that every value in the symbol table should be converted to a fullword pointer.) In order that both

```
.A  and  .A<3,3>        or      A←0  and  A<0,18>←0
```

will work properly, dot and left arrow have been modified. If the left half of a pointer word is zero, it is treated as a fullword pointer, but words with nonzero left halves are taken as true byte pointers. (The same applies to the MONITOR operation, which may be given either word addresses or byte pointers). The only way this will be noticeable to the user is that sometimes . will act like @, and ← will do a store where nothing should happen. We mean that .41, which will yield zero every time in Bliss (being equivalent to .41<0,0>) will yield the contents of word 41 in SIX12 (corresponding to @41 in Bliss). Similarly for 41 ← E. Note: A ← B should be written A ← B<0,36> .

Atsign @ does exactly the same thing as Bliss (the above doesn't apply).

Brackets [ ] perform a structure access according to the standard VECTOR structure. Hence E1[E2] is equivalent to (E1+E2)<0,36>. (Perhaps someday we will get structure information from the compiler...)

Peeking and Poking:

One basic requirement on a debugger is the ability to examine and modify program locations.

The only standard operator for changing memory locations is left arrow (←), which should need no more explanation. We should point out, however, that SIX12 never parses from right to left. Therefore,

```
A ← B ← C ← 0
```

will not work in SIX12. In order to discourage accidental use of this construct, left arrow is defined to have no value... in the example above, the address of B would be stored into A, then a syntax error would occur since the second ← would have no left operand.

One Bliss-compatible method for examination of program locations is provided in the dot (.) and atsign (@) operators. As we mentioned in passing previously, SIX12 prints out the result of every evaluated expression. Thus one need only type .ALPHA to see the contents of ALPHA; for example, a terminal interaction could look like this:

```
&.FLAGS,.PNTR
    677  677
   5737  FFAREA+5
```

(Note: & is SIX12's prompt character).

A DDT-type notation has also been implemented. The operator "/", used in a postfix fashion, prints out the contents of the fullword whose address is its argument:

```
&STACKCNT/
STACKC/   566005322  566,,SPACE+203
```

The infix operator "!" does the same thing for a consecutive set of words; A!N prints N words beginning at A. For example:

```
&BUFF!3
BUFF/          57  57
BUFF+1/       122  122
BUFF+2/         0  0
```

Note: In all cases, values or contents are first printed numerically

---

* Would you rather see the value of 'XYVAR' as 4322 or 4400004322? The situation is even worse if you're working in decimal, as then the halfwords are not separable by eye.

(in the default radix), then in symbolic halfword format (like DDT $R $H; offsets are in the default radix).

Breakpointing, Tracing:

The other basic requirement for a debugger is the ability to trace execution of a program and stop it ('break') where necessary. As we said earlier, SIX12 does this on a routine level. The basic terms are of setting (and later clearing) actions on routines. Such actions may be set conditionally. Conditions are fully general because they are given as SIX12 expressions. When required, the text given is evaluated; it must yield 1 in the low-order bit of its value for the action to be taken. (If the expression yields a vector value, only the first word is considered; a test which yields a null value always fails.) For instance, simple conditions* might be

```
                .FLAGS<17,1>
or              .CCOUNT  GTR  0
```

The standard syntax for setting unconditional actions is

```
        actionname  listofroutines
```

e.g.,

```
        BREAK  R2,PRINT,ERR3
```

The syntax for setting conditional actions is

```
        IF  $text of condition test$  actionname  listofroutines
```

e.g.,

```
        IF $.VALUE<10,1>$ TRACE TESTIT
```

where $ (=altmode!!!) delimits the text which is saved for evaluation.**
The syntax for clearing actions is

```
        Dactionname  listofroutines
```

i.e. same mnemonic with D prefixed, as in

---

*Note: it should be apparent that setting a conditional action on a frequently-called routine is a poor idea. For simple conditions such as the examples, the test requires several milliseconds (on the KA10).

**Two notes: 1) IF is a noise word and can be dropped. 2) Commas in a list of routines can be replaced by spaces. NEVER drop commas surrounding anything but a simple operand (number,symbol). Thus, in

```
     TRACE  T34, .PNTR, EXIT
```

the commas are necessary, but they aren't for

```
     TRACE  T34 EXIT METOO
```

The same applies to commas anywhere else in SIX12 (e.g., routine calls).

DBREAK ZURICH

This clears either conditional or unconditional action.

Conditional and unconditional actions do not coexist. There cannot be both conditional and unconditional instances of a given action on a given routine, nor can there be more than one condition governing a given action on a given routine. Thus, if a conditional or unconditional break is set on a routine, any previously set break of any type on the same routine is cleared, but other actions, say trace, on that routine are unaffected.

The possible actions are:

BREAK list	Stop execution at routine entry, with the message <=> AT: routine (←call-loc) parameters
ABREAK list	Stop execution at routine exit, with the message <=> AFTER: routine VALUE: value
TRACE list	Print a message when each routine is entered or left, without breaking. The messages look like this: --> routine (←call-loc) parameters <-- routine VALUE: value
TRACE AFTER list	Initiate trace mode when the routine is entered, so that all routine calls and returns are traced until the routine is exited. The original routine call and return are not traced. (Yes, Virginia, it still works if the routine is recursive!)
TRACE FROM list	Equivalent to TRACE plus TRACE AFTER; thus the routine and its subroutines are traced.
OPAQUE list	This lends a degree of abstraction to tracing. Tracing is turned off when an OPAQUE routine is entered, and remains off until the matching exit. OPAQUE 'outranks' TRACE; thus, even if routines with TRACEs set on them are called within the scope of an OPAQUE, they are not traced.
OPAQUE AFTER list	This does OPAQUE except that the routine itself is traced. Since we know that no trace printout will be required between entry and exit, paper is conserved by printing tracing notices for entry and exit on one line: --> routine (←caller) parms VALUE: value (assuming of course that tracing was on when the routine was entered).

These last five control the trace facility during program execution. The user may turn tracing on or off by means of the SETTRACE, CLRTRACE and GOTRACE operators, overriding OPAQUEs or TRACEs (see Getting In and Out of SIX12, next section). The TRACE and OPAQUE operators merely set or reset a switch controlling the printing of trace output. The user can manually set or clear this switch before resuming program execution.

Conditional actions set on routine exits may need to test the value which the routine is returning. This value is available as the contents of the global SIXVREG. In general, the user should never attempt to access any registers directly in SIX12 expressions. However, SIXVREG can be treated the same as the VREG (e.g., it can be modified, and the new value will be in the VREG when program execution resumes).

Getting In and Out of SIX12:

By getting into SIX12 we mean stopping execution of the user program and causing SIX12 to begin reading and executing user commands. Getting out is the reverse process of resuming user execution.

Getting in:

a) One method of entering SIX12 during execution is through a (previously set) breakpoint; see last section.

b) Another is through a break caused by terminal input monitoring, or the MONITOR (of data locations) operation. See Console Input Monitoring and Monitoring Variables.

c) You can enter SIX12 before program execution begins (but after stack initialization) by entering DDT and setting STARTFLG in SIX1.. to 1. For instance,

```
.DEBUG  program files,SYS:SIX12
LOADING
LOADER m+nK CORE
DDT EXECUTION

SIX1..$:        STARTFLG!       1 <CR>
$G
&
```

Here $ = altmode, and & is SIX12's prompt for an input. SIX1..$: can be dropped if the name STARTFLG is not used in your program.

d) You can explicitly call SIX12 from your program. Call the external name "SIX12" with one parameter, e.g.,

```
EXTERNAL SIX12;
.
.
.
SIX12(123);     ! HELP !!
```

SIX12 prints the parameter value and stop location:

```
PAUSE n AT location
&
```

e) You can get into SIX12 from DDT by PUSHJ SIXDDT$X. See Concerning DDT.

Getting out of SIX12 is accomplished by executing one of 3 operators:

GO — resumes user program without any special action.

GOTRACE — turns on tracing before starting. This cancels the effect of any active OPAQUE.

RETURN exp — The action of this depends on how you got into SIX12. For entry methods (c) and (e) above, it is the same as GO (but exp is the value of the CCL flag in method (c), if your main module was compiled with CCL). For method (d), exp is the value returned for the call to SIX12 (calls terminated by GO or GOTRACE return -1). For methods (a) and (b) above, if you stopped at a routine exit (ABREAK), execution resumes normally but exp is the value returned to the caller of the routine being left. If you stopped at a routine entry (BREAK), execution of that routine is suppressed completely, and control returns to its caller with the value exp. Thus RETURN is useful for hand-simulating unwritten or malfunctioning code.

Concerning DDT:

We have not tried to duplicate the many useful facilities already available in DDT. Instead, we have implemented easy transfers between SIX12 and DDT.

You can get into DDT from SIX12 by issuing the command

```
&DDT
```

To return to SIX12, type $P to DDT.

If you are in DDT but you didn't get there from SIX12, you can enter SIX12 by typing

```
PUSHJ SIXDDT$X
```

Subsequently issuing GO to SIX12 returns you to DDT. (Clearly, you must not do this if the stack has been destroyed.)

Note: If you have changed the default register declarations in such a way that the SREG is not register 0, you must issue

```
PUSHJ SREG,SIXDDT$X
```

instead. You must modify the routine SIXDDT in SIX12 if you change the default registers.

Console Input Monitoring:

If anything is typed when the program is not in a TTY input wait, SIX12 will shortly fake a breakpoint at some routine entry or exit, and wait for completion of the command. (The input is taken as the beginning of a debug expression. Many people cause interrupts by typing carriage return, so that they can get a prompt character before doing anything.) This monitoring is the normal state for SIX12. It can be disabled, permitting type-ahead, by the operator

&DISABLE

but will be automatically re-enabled whenever a break occurs (for some other reason, of course). If you issue DISABLE and subsequently regret it (e.g., get caught in an endless loop), you can re-enable monitoring by entering DDT and setting ENABFLG in SIX1.. to 1; the normal procedure is

```
↑C
↑C
.DDT

SIX1..$:        ENABFLG!        1 <CR>
$P
<CR>
&
```

Here $ = altmode; & is SIX12's prompt for an input. SIX1..$: can be omitted if your program does not use the name ENABFLG.

Monitoring Variables:

SIX12 can keep track of the contents of specified program locations, and report when they change. The contents of each location being monitored is compared against its last reported contents at every routine call and return. When any changes are found, SIX12 reports them and stops program execution (the same as a breakpoint). The monitoring message is

```
        *** BEFORE routine-name                 at an entry
or      *** DURING routine-name                 at an exit,
```

followed by a list of changes found, in the format

```
    location   OLD: oldvalue   NEW: newvalue
```

The syntax for requesting monitoring is

```
    MONITOR  listoflocations
```

where each location may be a word address or a byte pointer, as in

```
    &MONITOR  ACCUM, BUFHDR<0,18>, FLAGS<30,1>, FLAGS<21,1>, 41
```

(The input base was decimal here.)
The syntax for stopping monitoring is

```
    DMONITOR listoflocations
```

For example, the request shown could lead to a message as follows:

```
*** During GETCHR
CHAR  Old: 122        New: 56
FLAGS<30,1>   Old: 1  New: 0
&
```

where & indicates that SIX12 is waiting for a command. When the user issues GO, execution will proceed from the exit of GETCHR.

Note: Values are always printed in the default radix. When a monitoring request is not for a fullword, the position and size fields are printed in decimal.

Macros, symbol definition:

As a more or less free spin-off from conditional actions, we have implemented simple text substitution macros (no arguments at present). The format for defining a macro is

```
MACRO name=$macro text$
```

where $ (= altmode!!) delimits the macro text on both sides. The macro is invoked merely by writing its name, as in

```
&MACRO CALLR=$R4P(.A,  37)$
&RESULT ← CALLR
&CALLR
        5004  BUFFER+345
```

Macros can be deleted by the operator

```
FORGET  list of names
```

e.g.,

```
&FORGET CALLR
```

Space for the text is not reclaimed. See Disk I/O for a recovery method if you run out of text space. The PRINT MACRO operation can be used to examine the text of a macro; see the section on Display. Note: Macro names always have precedence over both operators and symbols; but a name followed by %n is never taken to be a macro.

New entries can be made in the symbol table; these names will also be available to DDT. The format is

```
BIND  name = expression
```

The name is defined as a global with value that of the result of evaluating the expression. For example,

```
&BIND  POINT= @@PNTR
&POINT
        1234567        1.,234567
&DDT

POINT=1.,234567              $P
&
```

The BIND operation should be used rather than MACRO to define a name with a constant value, as table lookup is much faster than macro

substitution. BIND is a good way to create debugging temporaries with user-specified names. For example,

```
&BIND MYTEMP = .JOBFF<0,18>; JOBFF ← .JOBFF + 1
```

or

```
&BIND DBGCHR= $2
```

(as explained later, $2 names the third of a set of temporary locations set aside in SIX12 for debugging use.)

Line printer and disk I/O:

Normally, all output from SIX12 is directed to the user's terminal. Under certain circumstances (such as when tracing, or dumping a large area of memory), it may be preferable to save the output on disk, or send it to the line printer. SIX12 contains a facility for doing this, which is controlled by the following operators:

LPTOPEN opens a file named SIX12.LPT on logical device LPT: . It does not initiate output to the file. (By assigning the logical name LPT: to some other device in advance, the user can cause the output to go anywhere.)

LPTON sets the output switch for output to the file. All subsequent printout from SIX12 (except error messages) will be directed to the file, not printed on the terminal.

LPTOFF resets the output switch for output to the terminal. It does not close the file, so that more file output may be done later in the same file.

LPTDUP sets the output switch for output to both terminal and file simultaneously. The worth of this option is doubtful, but it is included for completeness.

LPTCLOSE closes the file opened by LPTOPEN. All file output between one pair of LPTOPEN and LPTCLOSE forms a single file, no matter how many LPTONs and LPTOFFs have intervened.

Notes: An automatic LPTOFF is executed at every break. Thus output will normally go to the terminal during user interaction. LPT I/O uses logical channel 17. Do not use it when your program is using channel 17.

An option is provided for saving the state of SIX12 on a disk file, and restoring it at a later session without having to do considerable type-in, or save the whole core image. The operator

SAVE 'filespec'

saves all presently defined macros, requests for monitoring, and routine actions (including conditions) in a disk file specified by filespec, which is input as a string and must be enclosed by single quotes. The filespec may not contain a device specification.

LOAD 'filespec'

deletes any existing macros, monitoring requests, or routine actions, then loads the information in the SAVE-written file named by filespec.

Notes:

1) SAVE and LOAD use logical channel 16. You can use them in a program using that channel just as long as you do not issue them when your program has something open on 16. 2) The monitor and routine-actions tables contain absolute memory addresses. Thus SAVE/LOAD should not be used to preserve monitors or routine actions across a program reload. 3) Since all previously existing macros are deleted by LOAD, text space is compacted. The correct way to recover from a 'No space for macro text' error is to delete any unneeded macros, then issue

```
&SAVE 'TEMP'
&LOAD 'TEMP'
```

Display:

SIX12 has facilities for printing some information in a more meaningful format than could be obtained from dot or slash. In particular, special operators are available for displaying the run-time stack, the symbol table, and SIX12's internal tables.

These operators display the run-time stack in terms of routine calls.

CALLS displays the complete stack of routine calls. Each call is displayed in the format

```
routine (←calling-loc)  parameters
```

The first line (i.e., the current routine) is prefixed with B: if execution stopped at the routine's entry, or A: if at its exit, as in

```
A:IMHERE        (←CALLER+17) 1: 5       2: 0
CALLER  (←MAIN.F+12) 1: 45
```

which indicates that we are at the exit of 'IMHERE'.

LCALLS displays the call stack plus the locals area for each routine (including saved registers and 'displays' for functions) -- this may not be very useful to a user not familiar with the Bliss runtime environment. Locals are displayed after the call to the routine which owns them.

CALL n displays the last (topmost? innermost?) n calls on the stack. If n is omitted (i.e., CALL is used as a nullary operator), only the last call (to the present routine) is printed.

LCALL n works like CALL but also displays locals.

Two operators are included for searching the symbol table.

PRS listofsymbols (PRint Symbols)

For each symbol given, PRS prints every entry in the symbol table, in the format

```
name%ordinal    value   type    module
```

For instance,

```
&PRS  CTYPE,CX
CTYPE%1                 400360  Own     MAIN..
CTYPE%2*        5601    Own     INPU..
CX%1            500040  Global  DECL..
```

A * next to a name (following the ordinal) means that that entry will not be used for typeout by SIX12 or DDT (i.e., $K has been performed on it).

SEARCH 'partially-specified-symbol'

This allows searches using the "wild-card" convention that question-mark means any character, as in

```
&SEARCH 'P?C?'
PICK          500050  Own     TABL..
PAC             3001  Global  INPU..
&SEARCH '??????'
(prints every entry in symbol table)
```

The partially-specified symbol (only one per search) must be entered in single quotes. SEARCH does not print ordinals.

The PRINT operation displays the state of SIX12.

PRINT OPER name or PRINT OPER "char"

displays the definition (priority and routine name for each defined parse) of the specified operator, as in

PRINT OPER "↑" PRINT OPER AND

Note: Priorities are displayed in decimal.

PRINT MACRO name

prints the text of the macro named 'name'. For instance,

PRINT MACRO CALLR

PRINT ACTION actionname routine

prints the status of the specified action on the specified routine. The action must be given as one of

BREAK ABREAK OPAQ OPAQAFT TRACE or TRACEAFT
(remember that TRACE FROM = TRACE + TRACE AFTER).

The possible responses are

'Action not set'
'Unconditional'
or the text of the condition test.

For example,

```
&PRINT ACTION OPAQ XYZ
Action not set
&PRINT ACTION ABREAK PPP
.X LSS 3
```

Miscellaneous operators:

BASE n — sets the default base to be n, and prints the new base in decimal. Subsequent input numbers are assumed to be in this base, and output will appear in this base (except for items specified to appear in decimal). If n is omitted BASE only prints the current base. The initial default base is 8 (for octal).

WBASE n — sets the maximum displacement to be allowed when printing symbolic addresses in the form 'symbol+offset'. This works in exactly the same way as n$UW in CMU-DDT. (W is for Wulf for historical reasons.) WBASE functions in the same way as BASE except that the value is printed in the current default base. The WBASE is initialized to 1000 octal.

$n — (Dollar sign, not altmode.) This names the n-th of an array of locations in SIX12 which are reserved for use as debugging temporary storage. In the standard version n must be 0 - 19. As pointed out previously, BIND can be used to give meaningful names to them:

```
&BIND CHKSUM = $10
```

SETTRACE — Turns on the trace flag. When execution resumes tracing will begin immediately. This cancels the effects of any active OPAQUE. The GOTRACE operator described earlier is equivalent to

```
SETTRACE ; GO
```

CLRTRACE — Turns off the trace flag. This cancels the effects of an active TRACE AFTER or a previous SETTRACE.

NODEBUG — This performs

```
JOB41 ← 255000000000                (JFCL 0,0)
GO
```

Thus, the DEBUG UUO is rendered a no-op. This can be used to improve execution time if you are only running a program without intending to debug it.

Using SIX12:

The modules to be debugged must be compiled as follows:

1) Each module must be compiled with the DEBUG switch set. This can be accomplished by including "DEBUG" in the module head, or by specifying "/D" in the compiler command string.

2) The main module (the one including the STACK specification) must include "TIMER=EXTERNAL(SIX12)" in its module head. (At present, timing and debugging cannot be specified simultaneously.)

Once you have compiled all your files, load them with SIX12 and DDT; be sure that local symbols are loaded for your program files. The easy way to do this is with the monitor DEBUG command:

```
.DEBUG  your program files,SYS:SIX12
```

If you prefer to use LINK, you can issue

```
.R LINK
*/DEBUG  your program files,SYS:SIX12
*/GO
```

Usually, SIX12 should be loaded last so that its program symbols will be scanned last in a symbol table search. To reduce the chance of confusion further, you can load SIX12 without its local symbols, as in

```
.DEBUG  your program files,SYS:SIX12%W
```

(All global symbols in SIX12 begin with the letters S I X. The only non-global symbols that you are likely to need are STARTFLG and ENABFLG, which can be accessed as SIX1.0 and SIX1.0+1, respectively.)

The debugging linkages generated by the DEBUG switch use the 037 user UUO. If your program does not use user UUOs (opcodes 001- 037), you can skip the following. If you do use UUOs, you must

1) not use opcode 037,
2) arrange for your UUO handler to link to SIX12 properly.

(2) is merely a matter of getting the proper jump address for SIX12's UUO entry point. Because the entry point is not a global symbol, you must retrieve the jump address from location 41 before loading it with a branch to your own handler. (SIX12 loads 41 with a PUSHJ to itself immediately after stack initialization.) Your UUO initialization code might look something like this:

```
    GLOBAL UUOROUTS[ 40]; EXTERNAL UUOSWITCH, JOB41;
    .
    .
    UUOROUTS[ 037] ← .JOB41<0,18>;              ! **** DEBUG ONLY ****
! PUT  PUSHJ SREG,UUOSWITCH  INTO LOC. 41
    JOB41 ←  260↑27 OR SREG<0,0>↑23 OR UUOSWITCH<0,0>;
```

where the routine

```
    MACHOP      JRST =  254;
    GLOBAL ROUTINE UUOSWITCH = JRST(0,UUOROUTS[ .JOB41<27,9> ],0,1);
```

must be in a separate module which is compiled without FSAVE, TIMING, or DEBUG switches (/F, /T, or /D).

Defining your own operators:

As previously advertised, SIX12 is capable of easy extension. The method for this is normally to define new operators or revise standard ones to suit your needs. (Please review what we said about operators under 'SIX12 expressions', if it is not fresh in your mind.)

The syntax (print name, priority, possible parses) of an operator is defined by an entry in a syntax analyzer table. Its semantics are defined by a routine which the table entry points to. To evaluate the operator, the analyzer calls this routine using a standard linkage convention. The content of this section is a description of 1) the linkage convention and, 2) the proper method for making entries in the syntax table.

Linkage: Since operands and values can be vectors, it is not possible to transfer them by standard Bliss linkage. Instead, global variables are set to point to an operand and give its length. The variables are:

SIXLP contains a pointer to the first word of the left operand. (Undefined if no left operand.)
SIXLC contains the number of words in the left operand. (Zero if no left operand.)
SIXRP contain corresponding values for the right operand.
SIXRC

These variables are set at the routine call; the routine may destroy them if it wishes. (The contents of the operands may also be destroyed.) To return a value, the routine should set these two variables:

SIXVP must contain the address of the first word of the value. (The left half of SIXVP is ignored.)
SIXVC must contain the number of words in the value.

If no value is to be returned these can be left unmodified. (The criterion for finding a value is that SIXVC be positive; it is set to zero before calling the routine.) Note that operands and values are always vectors of fullwords.

An evaluating routine may need to determine what parse it has been called under. The parse in use can always be determined by examining SIXLC and SIXRC, but a more convenient way is provided. Evaluating routines are called with a single parameter (standard Bliss linkage), which has the value

```
0       for null parse (no operands)
1       for prefix parse (a right operand only)
2       for postfix parse (a left operand only)
3       for infix parse (both operands);
```

thus bit 0 denotes the presence of a right operand and bit 1 that of a left operand.

We suggest examining the source of SIX12 to see the best ways to code operators. The macro APPLY and the routines XBASE,LPAREN are recommended objects of study.

The table: For each operator symbol, the table of operators contains the symbol itself (print name), and information on each of the four possible parses for the symbol. This information consists of the priority of operation and the address of the evaluating routine for that parse. (If both are zero, the parse does not apply.) Priorities are in increasing sequence, that is an operator of priority 15 is evaluated before one of priority 14. If the user is making a permanent modification to SIX12, he should modify the table in the source program; this is explained by notes in the source. Otherwise, in order to avoid recompiling SIX12, the user can create his new routines separately, compile them (without /D, of course!), load them together with SIX12 and the program to be debugged, and modify the operator table at run-time. To facilitate this approach, the linkage variables explained above are all global names, and an operator is provided for modifying the table at run-time.

```
        DEFINE name,parse = priority,routine
or      DEFINE "char",parse = priority,routine
```

sets the parse information as requested. "Parse" can be one of

```
NULL    PREFIX    POSTFIX    INFIX
```

or 0 - 3. Note that only the specified parse is affected; the others remain set as before. (If a new operator symbol is being defined, the other 3 are initialized to zero, i.e. "parse not applicable".) For example,

```
&DEFINE SIXBIT,PREFIX = 100,CONVRT
```

where CONVRT names a routine to translate ASCII inputs to SIXBIT outputs, could be used to implement the SIXBIT stringtype. Once this has been entered, the user could issue

```
&FILNAM ← SIXBIT 'ABCDEF'
```

Again,

```
&DEFINE "?",0 = 10,DISPLAY
```

would make it possible to call the routine DISPLAY by typing a question mark. (If DISPLAY expected no arguments and returned no interesting value, this could be done even if DISPLAY had not been written explicitly as a SIX12 operator...) Thus

```
& ?
(output from DISPLAY)
&
```

A Compressed Version

The PRINT OPER operation can be used to verify the effects of DEFINE. Remember that priorities are printed in decimal by PRINT OPER.

SIX12S:

A smaller version, SIX12S, exists for those who find that standard SIX12 is too large and/or too slow. SIX12S has no macros, conditional actions, or SAVE/LOAD facility. This means much faster breakpoint checking, and much less code and table space inside SIX12. Otherwise it is identical to regular SIX12. The operators not defined in SIX12S are

```
MACRO           PRINT MACRO             FORGET
IF  $test$  action  rtn-list            (i.e., conditional actions)
SAVE            LOAD
GEQ through LSS         (you won't need them without conditionals).
```

To use this version (at CMU), access

SIX12S.BLI, SIX12S.REL on DSKB:[L130BL98]
(it will not be put on SYS: at CMU).