Symbolic Debugger

User Guide

Intelligent Interfaces Ltd

Issue 2 March 1999

All information is given by Intelligent Interfaces in good faith. However, it is acknowledged that there may be errors or omissions in this User Guide. Intelligent Interfaces welcomes comments and suggestions relating to this User Guide.

All correspondence should be addressed to:-

Technical Queries
Intelligent Interfaces Ltd
P O Box 80
Eastleigh
Hampshire
SO53 2YX

Tel: 023 8026 1514
Fax: 087 0052 1281
E-mail: support@intint.demon.co.uk
URL: http://www.intint.demon.co.uk

This User Guide is intended only to assist the reader in the use of the Symbolic Debugger and, therefore, Intelligent Interfaces shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or an error or omission in, this User Guide.

CONTENTS

Introduction

This User Guide describes the Symbolic Debugger (ASD) which is an interactive aid to debugging programs written in FORTRAN 77 on Acorn RISC OS-based computers. Note that the terms source-level and symbolic are here treated as interchangeable. ASD was originally intended for debugging programs written in C, ISO-Pascal and FORTRAN 77. Therefore, this User Guide contains a number of references to C and ISO-Pascal. These have been retained as an aid to the understanding of the general principles involved in symbolic debugging. ASD should now only be used for debugging programs written in FORTRAN 77.

In this chapter the section, *About Debuggers*, introduces the concept of debuggers in general, and source-level debuggers in particular. *Using ASD* describes the way in which ASD is called from the command line. The last section, *Specifying Source-Level Objects*, describes the way in which various source-level items, such as variable names, line numbers and labels are specified in ASD commands.

The chapter *Program Examination Commands* describes the ASD commands which are concerned with examination of the program being debugged. You can print the value of program variables (or more precisely, arbitrary expressions involving variables and constants), and examine the state of execution of the program and the arguments of an active procedure call. You can also alter the value of a variable.

The chapter *Execution Control Commands* deals with the ASD commands which control the running of the program being debugged. Facilities available include the ability to start (or re-start after a break point) program execution, single-step a statement at a time, set and clear break points, and to initiate the 'watching' of a variable. Furthermore, tracing of procedure calls is available. If any of the terms used above are new to you, then the section *About Debuggers* should clarify matters.

The chapter *Miscellaneous Commands* describes the ASD commands which do not fall into either of the two previous categories. These commands include those that issue operating system commands, display help information, and quit from the debugger.

The chapter *An Example ASD Session* gives an example of how ASD might be used to track down a typical 'side-effect' bug.

The chapter *Command Summary* lists all the ASD commands alphabetically, with a brief description and page reference for the appropriate section in this user guide.

Conventions

In this User Guide, anything which is to be typed at the keyboard, or reflects something displayed on the screen, is shown in a typewriter font, eg:

```
*ASD -help
```

In descriptions of command syntax, *`italics`* are used to denote classes of item, rather than literal strings to be typed, eg:

`let` *`variable=expression`*

Items within square brackets [] are optional:

`con[text] [`*`context`*`]`

This means that the command `context` may be abbreviated to `con` and may optionally be followed by a *`context`* (described later). Finally, if a choice is available, the possibilities are separated by `|`, as in:

```
lang[uage] [f77|c|pascal]
```

This means that any one of the three strings may be used (or none of them, as they are optional).

About Debuggers

This section is aimed mainly at readers who have not used a program debugger of any sort before. However, others may find it useful reading, as it introduces some of the terminology used in the rest of this user guide.

Anyone who has written a program more than about ten lines long has had recourse to debugging techniques: the tracking down and removal of errors. The form this takes depends on many things, not least the language in which the program is written. A common technique for tracing bugs in systems which have no explicit debugging support is the planting of 'trace' information in the program itself. Such additions to the program can be useful, but are tedious to use in compiled languages because every time you want to change the debugging statements, the program has to be re-compiled. There is also the possibility that the debugging statements themselves have undesirable side-effects which contribute to the ill-health of the program.

Planting tracing information in assembly language programs is even trickier. In general, the assembly language programmer does not have access to the rich expression evaluation and print formatting routines of high-level languages. For example, displaying the contents of all of the ARM's registers in hex and decimal is a non-trivial code fragment in ARM assembler.

To help assembly language programmers debug their programs, a class of utility known as the debugger or monitor has evolved. Such programs allow the user to examine and alter memory locations and machine registers, set break points, single step through a program and 'watch' particular memory locations for changes.

Because much of the terminology used in ASD derives from such debuggers, the next section describes typical facilities that they provide. These are then compared with the equivalent ASD commands.

Starting execution

An important first step to debugging a program is of course to start it executing. Machine code monitors usually provide a command of the form `GO` *`addr`* which starts execution from a particular instruction. The ASD equivalent is just `go`. Machine code addresses do not appear in ASD commands.

Examining memory

As mentioned above, a machine-code-level debugger allows you to examine the memory of the machine, and possibly alter its contents. A typical command would be `MEMORY` *`address range`* to display the contents of a range of memory locations in hex, and `SHOWREGS` to display the contents of the machine's registers. The ASD equivalent is `print`. This displays the value of an arbitrarily complex expression. Instead of using memory addresses, you can use the names of variables, exactly as in the original source program. Note that there is no ASD equivalent to showing the state of the registers.

Setting break points

It is useful to be able to stop the program at a certain point, so that the state of its variables can be checked against their expected values. This is known as setting a break point. When the program reaches the instruction at which the break point has been set, execution is suspended and control returns to the debugger. In machine code terms, break points are set at particular memory addresses, eg:

```
BREAKSET 1ABFC.
```

Under ASD, you set break points in terms of source level addresses. That is, you specify the source-file line number (and possibly the statement within that line) where you want the program to stop, eg:

```
break ray:234
```

You can also set a break point at the entry and exit points of a procedure. A useful extension of break points is being able to set a count of the number of times the statement has to be executed before the program is suspended.

Single stepping and tracing

Once a break point has been encountered, you may wish to step slowly through the program, examining changes which occur to variables after each step. At the machine code level, single stepping involves executing one instruction at a time. Under ASD, single stepping works at the level of one statement at a time, as this is the smallest indivisible source-level construct. You may specify whether a procedure (or function) call counts as one statement or whether each of the statements within that procedure should be stepped individually.

Single stepping can be quite time consuming, especially if you only want to get an idea of the general flow of control within the program. An alternative is to use procedure tracing. When this is enabled, every time the program enters or leaves a procedure, a message is displayed to that effect.

Setting watch points

A common cause of incorrect operation in programs is the corruption of a variable. The reason for this corruption can be very hard to track down, especially if the program contains many global variables which are accessible from a large number of procedures. One way of minimising the chances of the problem occurring is to adopt a strictly modular approach to programming and reduce the number of global variables used to pass information between procedures. This is not always possible, for efficiency or language reasons. ASD helps to track down undesired assignments to variables by allowing you to place a 'watch' on them. When a variable is being watched, a check is made for any changes in the variable's value. When a change occurs, the program execution is suspended as if a break point had been encountered.

This concludes the description of common debugger concepts. Of course, only a few of the commands provided by ASD have been mentioned so far. Detailed descriptions of all of them can be found in the following chapters.

Using ASD

This section describes how the debugger is called, and how programs to be debugged under it must be compiled. ASD relies for its operation on the presence of information about the source code of the program being debugged. This information is not automatically included in the output of the compiler. (This is mainly for reasons of efficiency: programs which contain debug information are larger and take longer to compile than ones which do not include it.)

The generation of debug information is enabled by specifying the appropriate option or 'switch' on the command line.

FORTRAN-77 Compiler Debugging Options

The compiler for FORTRAN 77 is split into two components: the front end and the code generator. It is the latter part which controls generation of the debugging information, and therefore which takes the options enabling it. The option is `-debug` and may be followed by a word indicating the level of information required:

`none`	No information. This is the default
`min`	Subroutine and function names information only
`vars`	Subroutine and function names, plus variable name information
`lines`	Subroutine and function names, plus line number information
`all`	Subroutine, function, variable and line information. max is a synonym for all.

Additionally, the code generator also requires the name of the source file so that this may be planted in the debugging information. The switch is `-source` followed by the filename, eg `-source raytrace` (The source file name is included so that line numbers specified by the user may be qualified with the filename, where a simple line number might be ambiguous.)

Once the program has been successfully compiled using the desired debugging options, it must be linked to produce a final object program. `f77link` knows about debugging information and will produce a file which is suitable for use by ASD.

Invoking The Debugger

Once you have a successfully linked the program, the debugger may be used to control its execution . To call ASD, ensure that the program of that name is somewhere in your run search path. Then issue the command:

`*asd` *`filename`*

from the command line, where *`filename`* names the program which is the object of your interest. For example:

```
*asd raytrace
```

ASD responds to the `-help` option. This is the only option that it recognises; it responds by printing the version number, sub-command syntax, and some other useful notes.

On starting, ASD prints a few lines of the following form:

```
ARM Symbolic Debugger, version 1.00
Object program file raytrace
ASD:
```

If the file specified in the command line could not be found, an error message to that effect is displayed, and ASD returns to the command line.

To see a list of the commands that ASD provides, type help. You may recognise some of these and be tempted to use them. However, you are recommended to at least read the next section and the command summary in *Appendix B* before you do.

Specifying Source-Level Objects

Once ASD is running, the object program can be executed, single stepped, have its variables examined and so on. All of these facilities are described in the following chapters. However, before you can use these commands, you have to know how to specify certain source-level entities. For example, variable names, line numbers and program labels all have a syntax which must be used correctly if you are to reference the desired object.

Variable names and context

It is clearly important that a source-level debugger allows you to refer to the program's

variables by the names they have in the original source code. A variable is simply referenced by its name: if you want to print the value of variable count, you would use the command:

```
print count
```

When a program written in a block-structure high-level language is executing, there exists a current context. This refers to both the textual nesting of procedures, and the dynamic run-time nesting of procedure calls. Taking the first aspect first, and using Pascal as an example, consider the following definitions:

```
program raytrace(input,output);
   var
      count : integer;
   ...
   procedure pixel(x,y : integer);
      var
         i : integer;
      ...
      function reflect(x,y : integer; angle : real) : integer;
         ...
         function quicksin(angle : real) : real;
         begin
            {of quicksin ** BREAK POINT HERE ** }
         end;
      begin
         {of reflect }
      end;
   begin
      {of pixel }
   end;
begin
   {of raytrace }
end.
```

Assume the program stops (because of a break point, perhaps) in the body of the function `quicksin`. At this point, the variables visible to the program are `angle`, the parameter of the function, `x` and `y`, the parameters of `reflect`, `i`, the local variable of `pixel`, and the global variable `count`.

Variables that are defined in the current context can be accessed from the ASD command prompt simply by giving their names. This would include the real parameter of `quicksin` called `angle` in the current example, but no others.

You can refer to other variables by qualifying their names with the context (procedure name) in which they are defined. For example, the parameters of the function `reflect` would be referred to as `reflect:x`, `reflect:y`, and `reflect:angle` respectively. Notice that in the last example, you can refer to a variable which would not actually be visible to the executing program.

Another example would be a reference to the global integer variable `count`, defined in the main program. Here you use the module name as the qualifier. In Pascal, the name after the `program` keyword is taken as the module name, so the required identifier is `raytrace:count`. In analogous situations in C, you would use the source filename as the qualifier (see the examples below and in the *Command Summary*). In FORTRAN-77, as with Pascal, the `PROGRAM` name is used to prefix global variables.

To avoid certain ambiguous cases, where more than one procedure of a given name exists, you can 'nest' the qualifiers before the final variable name. For example, the local variable

`angle` in the function `reflect` could be referred to as `raytrace:pixel:reflect:angle`, even though the first two components are not strictly required to access the desired object unambiguously in the present example.

The next example, in C, illustrates the further features of specifying variables outside the current context. The C language does not support the textual nesting of functions, so variables are either defined at the top level (outside any function definitions), or one level down, in a function definition (though see later in this section for further discussion). However, the language obviously does support nested function calls, and like Pascal, allows recursive calls.

Consider this source code fragment:

```
/* File >c.expr */
int val;

int factor(char **ptr)
{val ;
 /* BREAK POINT HERE */
 ...
 return val;
}

int add(char **ptr)
{v1 , val;
 ...
 v1 = factor(ptr);
 return v1+val;
}

int logical(char **ptr)
{v1 , val;
 ...
 {v1 = add(ptr);
  return v1 & val;
 }
}

int expr(char **ptr)
{return logical(ptr);
}

void main(int argc, char *argv[])
{*p ;
 ...
 val = expr(&p);
 ...
}
```

In this example, there is a global variable `val`, and several local variables. Suppose the current context is in the body of `factor`, which was called by `add`, called by `logical`, called by `expr`, called by `main`. As far as the programmer is concerned, the only variables visible at this stage are local integer `val` and the parameter `ptr`. Because of C's lexical scoping rules, all the other variables which are extant are invisible from `factor`. The global `val` would have been visible, had it not been for the local of the same name hiding it.

Similarly, the only variables accessible directly to ASD are those defined in the current context of `factor`. However, the others can still be examined and altered by giving their defining

contexts. Examples are: `expr:ptr`, `main:argc` and `add:v1`.

Consider what the name `expr:val` would refer to. Seemingly there are two candidates: the global variable, which qualifies because `expr` is the name of the module (program) in which it is defined, and the local variable defined in the function which is also called `expr`. In fact, it is the second one which would be referenced, as ASD always accesses more local variables first.

To overcome the above ambiguity, there is a special qualifier called `$ROOT`. You can regard this as the 'parent' of all the modules in the program, so the element following it in an identifier name is a module. In this example, you would use `$ROOT:expr:val` to access the global called `val` (and `$ROOT:expr:expr:val` would have the same meaning as just `expr:val`).

The function `logical` shows another example of possible ambiguity, which can occur in C but not Pascal. In Pascal, you are not allowed to declare local variables within `begin...end` blocks; in C you are. There are two declarations of `v1` in `logical`, so the name `logical:v1` is not precise enough. The way around this is to qualify the name with the line number on which it is declared as well as (or instead of) the function name. These might be `logical:115:v1` and `logical:123:v1` in the present example.

ASD will give an error if a reference to a variable name is ambiguous.

Finally, there is the question of multiple activations of a particular procedure. Both C and Pascal allow recursive function and procedure calls. Consider the standard example:

```
int factorial(int n)
{(n <= 1)
  return 1;
 else
  return n*factorial(n-1);
}
```

Suppose this function is called with an initial argument of 5. The function would recurse four times, until it was called with `n==1`. At this stage, asking ASD to access variable n would yield the most recent version. By prefixing the variable name with a backslash (\) followed by an integer, you can refer to any of the other activations. For example, `\1:n` refers to the first (oldest) invocation of `factorial`, where `n==5`. `\2:n` refers to the next oldest, and `\5:n` would refer to the active one. Negative integers can be used to work backwards from the current activation. So in this example, `\-1:n` and `\4:n` would be the same (`n==2`).

Note that if the function or procedure name is required along with an activation count, the form is `factorial\-1:n`.

This section has been quite involved, but that only reflects the versatility that is required by ASD in order to allow access to the various types of variable instantiations possible in modern, block-structured languages. For the most part, you will find that unqualified variable references are all that is required, and convoluted strings such as
`$ROOT:raytrace:fred:jim\-1:count` are not needed.

Program locations

Some ASD commands require arguments which refer to locations in the program. The command `break` is an example. You can refer to a place in the program by procedure entry/ exit, line number, statement within a line (in the case of C and Pascal), or label.

You reference lines simply by giving the line number: `123` refers to the 123rd line of the program. You can qualify line numbers in the same way as variables, so `prog:87` is the 87th line in `prog` and `$ROOT:ray:3214` is the 3214th line in the module `ray`.

You can use the procedure name alone, eg `fastsin` to set a break point at the entry point of the procedure. Alternatively, the end of a procedure (just before it returns) may be trapped using `proc:$exit`. (For symmetry's sake, `proc:$entry` can be used for the entry point, but the procedure name alone may be used just as effectively.)

To refer to a statement within a line, the notation `line.stat` is used. For example, `100.3` refers to the third statement on line 100. Clearly statement *n*`.1` is the same as statement *n*`.`.

It is possible that a simple line number is ambiguous. This occurs when an include file is invoked from within a function. For example, suppose you have the file `h.ray` which looks like this:

```
0001 #define maxx 1000
0002 #define maxy 1000
0003
0004 typedef unsigned char flag, byte;
0005 ...
...
0099 /* End of h.ray */
```

(The line numbers are included for clarity; they would not appear in the file itself.) Suppose further that this file is included in a C source file:

```
0001 main()
0002 {0003 #include "h.ray"
0004 int x,y;
0005 real angle
0006 ...
```

The resulting source as seen by the compiler, with line numbers, is:

```
0001 main()
0002 {0003 #include "h.ray"
0001 #define maxx 1000
0002 #define maxy 1000
0003
0004 typedef unsigned char flag, byte;
0005 ...
...
0099 /* End of ray.h */
0004 int x,y;
0005 real angle
0006 ...
```

Line reference `4` might refer to the `typedef` or the `int` declaration. To overcome the ambiguity, it is possible to suffix the line number with the filename: `4(c.ray), 4(h.ray)`.

The final type of program location reference is the label. FORTRAN 77 statement numbers are numeric. To distinguish these from normal line numbers, use the prefix `$`.

Expressions

Several ASD commands require arbitrary expressions as arguments. The syntax for these expressions is based on that found in the C language. This means it will come naturally to C programmers, but FORTRAN users will need to make some adjustments.

ASD has a set of operators and several levels of operator precedence. These are summarised

below.

1	`()`	grouping, eg `a*(b+c)`
	`[]`	subscript, eg `isprime[n], matrix[1][2]`
	`.`	record selection, eg `rec.field, a.b.c`
	`->`	indirect selection, eg `rec->next is (*rec).next`
2	`!`	logical not, eg `!finished`
	`~`	bitwise not, eg `~mask`
	`-`	negation, eg `-a`
	`*`	indirection, eg `*ptr`
	`&`	address, eg `&var`
3	`*`	multiplication, eg `a*b`
	`/`	division, eg `c/d`
	`%`	remainder, eg a%b is `a-b*(a/b)`
4	`+`	addition, eg `a+1`
	`-`	subtraction, eg `b-d`
5	`>>`	right shift, eg, `k>>2`
	`<<`	left shift, eg `2<<n`
6	`<`	less than, eg `a<b`
	`>`	greater than, eg `n>10`
	`<=`	less than or equal to, eg `c<=d`
	`>=`	greater than or equal to, eg `k>=5`
7	`==`	equal to, eg `n==0`
	`!=`	not equal to, eg `count!=limit`
8	`&`	bitwise and, eg `i & mask`
9	`^`	bitwise xor, eg `a ^ b`
10	`|`	bitwise or, eg `m1 | 0x100`
11	`&&`	logical and, eg `a==1 && b!=0`
12	`||`	logical or, eg `a>lim || finished`

The lower the number, the higher the precedence of the operator. Note the syntax for subscripting and record selection. The object to which subscripting is applied must be a pointer or array name. The debugger will check both the number of subscripts and their bounds in languages which support such checking. A warning will be issued for out-of-bound array accesses. As in C, the name of an array may be used without subscripting to yield the address of the first element.

If the left hand operand of a right shift is a signed variable, then the shift will be an arithmetic one (ie the sign bit is preserved). If the operand is unsigned, the shift is a logical one, and zero is shifted into the most significant bit.

If incompatible types are used during expression evaluation, the debugger will print a warning message, but evaluation will continue.

Constants may be decimal integers, floating point, octal integers or hexadecimal integers. Respective examples are:

`123`,
`12.3e10`,
`0100` (64 decimal),
`0x1ff` (511 decimal).

Character constants are also allowed, eg A yields 65 (the ASCII code for A). Note that `1` is an integer, whereas `1.` is a floating point number.

Program Termination Commands

This chapter lists and describes those commands which examine the state of the program being debugged. In the syntax descriptions of the commands, various items such as `context` are mentioned. These are explained below.

`context` describes an activation state of the program. Possible elements of a context were described in the section *Specifying source-level objects.* of the chapter *Introduction* . Formally, it looks like:

```
[[$ROOT:]module:][proc:]...proc[\[-]count]
```

In other words, an optional module prefix, optionally prefixed by `$ROOT` to avoid ambiguity with procedures of the same name, followed by a list of procedure names, the last of which which may have optional invocation level following the backslash. Examples are:

`pixel`	procedure or module called pixel
`raytrace:pixel`	procedure pixel defined in raytrace
`raytrace:pixel\-1`	previous invocation of pixel
`$ROOT:raytrace:pixel:reflect`	procedure reflect defined in raytrace

If the program is currently in a 'stopped' state, eg after a break point or watch point has been activated, there is an `execution state context`. This refers to the context of the procedure being executed when it was suspended.

`expression` is an arbitrary expression using constants, variables and the operators described in the previous chapter.

`variable` is a reference to one of the program's variables. If a simple name is used, the variable is looked up within the current context. This may be over-ridden by prefixing the variable name with a `context` as described above.

`count` is an unsigned decimal integer.

`format` is a C `printf` function format descriptor, or the word `hex` or `ascii`. It is beyond the scope of this user guide to describe all of `printf`'s format strings, but the most common ones are:

Type	Format	Description
int	%d	signed decimal integer (default for integers)
	%u	unsigned integer
	%x	hexadecimal with lower case letters (same as `hex`)
	%X	hexadecimal with upper case letters
	%o	octal
char	%c	character (same as `ascii`)
float	%e	exponent notation, eg 9.999999e+00
	%E	exponent notation, eg 9.999999E+00
	%f	fixed point notation, eg 9.999999
	%g	general floating point notation, eg 1.1, 1.23e+06
	%G	general floating point notation, eg 1.1, 1.23E+06

Note that in the `print` command, the first group above (`int` and `char`) should only be used if the expression being printed yields an integer, and the second group should only be used for floating point results.

PRINT Command

This command can be used to examine the contents of the debugged program's variables. You can also use it to display the result of arbitrary calculations involving variables and constants. The syntax is:

`p[rint][/`*`format`*`] `*`expression`*

The *`format`* string was described in the previous section. If it is omitted, then for integer expressions, the default set by the `format` command is used. This is `%d` by default. For floating point values, the default format string is `%g`.

FORTRAN 77 complex numbers are printed in the form (*`real`*,*`imag`*).

Note that the `/` marking the start of the `format` should follow the command name, with no intervening spaces. Examples are:

```
print \-1:a+1
print isprime[3]
print/hex isprime
print/%10s promptstr
print/%f angle*180/3.14159
```

FORMAT Command

This command is used to set the default format string used by the `print` command for integer results. It is set to `%d` when ASD starts up. That may not be suitable (eg you may want to treat integers as unsigned quantities, or print integers in hex) so `format` allows you to change it. The syntax of the command is:

`form[at] [`*format*`]`

The *`format`* string is exactly as described previously. Examples are:

```
format hex
format %u
```

Note that there is nothing to stop you from using one of the floating point formats in this command. It would not be very wise, though, as integers would then not be printed correctly at all.

LET Command

The `let` command enables the value of a variable to be changed. It has the syntax:

`let `*`variable`*`=`*`expression`*

The *`variable`* and the *`expression`* should be compatible types, though the debugger will perform conversions between integer and floating if necessary (floats are rounded towards zero). It will also allow assignments to FORTRAN-77 complex variables. Only the real parts are affected by arithmetic on these types.

Note that although the value of an array *`element`* can be changed, using a command such as:

```
let isprime[2]=1
```

the address of the array itself cannot be changed, as array names are treated as constants.

Example:

```
let a=a+1
```

SYMBOLS Command

This command lists the symbols (variables) defined in the given context, or the current context if it is omitted. Its syntax is:

```
sym[bols] [context]
```

Each variable's name is displayed, along with its type information. An example of the output produced might be:

```
ANGLE  Float, local
X      Signed integer, local
Y      Signed integer, local
I      Signed integer, local
```

The format is *name type*, *storage class*. Other types you might see are:

```
Signed half-word (short)
Signed byte (character)
Unsigned integer
Unsigned half-word (short)
Unsigned byte (character)
Float
Double
Pointer to...
Array of...
```

Other storage classes are:

```
register
automatic (local)
static
external
```

To see the global variables, you would quote the module name as the context. For example, to see the external and static variables defined outside of any function definitions in a C program, you might use:

```
sym testp
```

where `testp` is the name of the source file.

Note: See the comment about potential problems with register variables in the description of the `watch` command.

VARIABLE Command

This provides type and context information about a specified variable. Its syntax is:

`var[iable]` *variable*

Examples of its usage and the results displayed are:

```
ASD: var angle
```

```
ANGLE           Float, local
context: FASTSIN  pascal.raytrace
ASD: var reflect:angle
ANGLE           Float, local
context: REFLECT  pascal.raytrace
ASD: var count
COUNT           Signed integer, static
context: RAYTRACE  pascal.raytrace
```

ARGUMENTS Command

This command is used to show the arguments which were passed to the current procedure, or to another active procedure. Its syntax is:

`arg[ument]s [`*`context`*`]`

If the *`context`* is omitted, the current context is used (usually the procedure that was active when the program was suspended, unless it has been changed by a `context` command). Examples are:

```
args
args \-1
args main
```

For each argument, its name and current value are displayed.

CONTEXT Command

This is used to set the context in which variable lookups will occur. It also affects the default context used by commands such as `symbols`. When program execution is suspended, the search context is set to the active procedure. The syntax of the command is:

`con[text] [`*`context`*`]`

If the argument is omitted, the context will be reset to the active procedure. Examples are:

```
con
con factorial\1
con prog:expr
```

OUT Command

The next two commands, `out` and `in`, are shorthand ways of changing the current context by one level. `out`, the syntax of which is simply:

out

sets the context to that of the caller of the current context. For example, if the current context were `pixel:reflect:quicksin` then executing an `out` command would set it to `pixel:reflect`. An `out` command cannot be issued if the current context is the top level of execution.

IN Command

This command performs the opposite function to `out`. It sets the context to the procedure called from the current level. Continuing with the previous example, if an `in` is executed, the context will be set back to `pixel:reflect:fastsin`.

The syntax of the command is:

```
in
```

An `in` command may not be issued when the current context is that of the executing procedure.

WHERE Command

This command prints the current context in terms of a procedure name, line number in the file and filename. The syntax is simply:

```
where
```

An example display from the `where` command is:

```
REFLECT          45                  pascal.raytrace
```

BACKTRACE Command

This command is similar to where. It prints the same information, but does so for all of the currently active procedures (most recent first), or for a given number of levels. The syntax is:

```
b[ack]trace [count]
```

An example of the output from this command is:

```
REFLECT          45                  pascal.raytrace
PIXEL            124                 pascal.raytrace
RAYTRACE         548                 pascal.raytrace
```

Execution Control Commands

This chapter describes the ASD commands which control the execution of the object program. Facilities covered include the setting of break points and watch points, and single stepping.

GO Command

This command starts execution of the program. The first time `go` is executed, the program starts from its normal entry point (eg at the start of the `main` function in C). Subsequent `go`s resume execution from the point where execution was suspended. This will be at a break point or a watch point.

The syntax is:

```
go
```

Parameters can be set up that will be passed to the program using the `cmdline` command, described in the chapter *Miscellaneous Commands*.

STEP Command

This command steps execution through one or more statements. It can only be issued after the program has been started: you cannot use `step` to initiate program execution. The syntax is:

```
s[tep] [in] [count]
```

If the optional *`count`* is omitted, one statement is executed, otherwise *`count`* statements are executed. The `in` keyword, if present, denotes that single stepping continues into procedure calls. That is, each statement inside a called procedure counts. If `in` is absent, a procedure call counts as only one statement, and is executed without single stepping.

Examples are:

```
step 20
s in
s in 5
```

BREAK Command

This command is used to set a break point. Break points may be specified at procedure entry and exit, lines, statements within a line, or at program labels. The section *Specifying source-level objects* in the chapter *Introduction*, describes how program locations are specified. The syntax of `break` is:

```
br[ea]k [location [count]]
```

That is, the command may be followed by a *`location`*, and that may be followed by a *`count.`* If you issue `break` with no arguments, a list of the currently set break points is displayed. For example:

```
#1 at FASTSIN; count = 1/1
#2 at RAYTRACE:324; count = 1/10
#3 at RAYTRACE:$999; count = 1/1
```

The two numbers after `count =` are the number of times the break point has been encountered since the last program stop, and the break point count, respectively.

Break point numbers (#n) may be used in the `unbreak` command instead of the location descriptor.

The *`count`* that follows the break point location indicates how many times the statement there must be executed before the program is actually suspended. It defaults to 1, so if the *`count`* is omitted, execution will stop the first time the break point is encountered.

Examples are:

```
brk
break fastsin
brk raytrace:324 10
brk raytrace:$999
```

Note: If a break point is set at a procedure exit, using for example:

```
brk proc:$exit
```

then several break points may be set, one for each possible exit. (A C function, for example, may have multiple `return` statements.) Break points which are not required may then be deleted using unbreak with a break point number or they can all be deleted by using the same location as given in the break command.

UNBREAK Command

This command removes a break point location from the current list. It has the form:

`unbr[ea]k` *`location`*

where *`location`* may either be a source code location, or the break point number, as displayed by the `break` command. If the break point being removed is not the last one, the break point list is not renumbered, so once a break point number is assigned, it remains constant.

Examples are:

```
unbrk #1
unbrk raytrace:$999
```

WATCH Command

This command is used to set a watch point on a variable. When the variable is altered, program execution is suspended. The syntax of the command is:

`watch [`*`variable`*`]`

If the argument is omitted, a list of current watch points is listed, eg:

```
#1 at K
#2 at ISPRIME[4]
```

As with break points, the watch point number may be used in the `unwatch` command to remove a watch point.

Examples are:

```
watch k
watch isprime[4]
```

Notes: If there are any watch points set, execution becomes very slow. This is because the values of the watched variables are checked after every machine instruction. The best way to deal with this is to set a break point in the area of code under suspicion, and only set the watch point(s) when the program stops there.

UNWATCH Command

This command clears a watch point. It has the syntax:

`unwatch` *`variable`*

As mentioned above, the variable reference may either be an actual variable name, or a watch point number preceded by a # sign.

RETURN Command

This command can be used to return to the caller of the current procedure, passing back a result if required. It has the form:

`return [`*`expression`*`]`

For example, if execution is suspended in a Pascal procedure, return would pass control back to the caller, and execution would be continued from the next statement.

It is not possible to return compound data types (arrays and records) using this command.

PTRACE Command

This command enables and disables procedure tracing. When enabled, this causes the name of the current procedure to be printed every time it is entered or left. The syntax of the command is:

```
pt[race] [on||off]
```

If no argument, or `on`, is given tracing is enabled. If `off` is specified, tracing is disabled. Indentation is used to indicate procedure nesting. For example:

```
Entered main
 Entered init
 Left init
Left main
```

Miscellaneous Commands

This chapter describes those commands which do not come under the headings of the previous two chapters.

CMDLINE Command

This command is used to set the command line arguments read by the debugged program. Normally they would appended to the command name. For example

```
*diff file1 file2
```

When running a program under the debugger, the following would be used instead

```
*asd diff
ASD: cmdline file1 file2
...
ASD: go
```

The syntax of the command is:

`cmdline [`*`string`*`]`

where *`string`* is a sequence of zero or more characters.

HELP Command

This command displays a list of available commands, or help on a particular command. Its syntax is

`help [`*`command`*`]`

If the argument is omitted, a complete list of ASD commands is displayed (long forms only). If the argument is present, that command's syntax and a brief description is printed. For example, `help p` will display:

```
p[rint][/<format>] <expr>
Print value of expression, using given format or default.
```

Notice that the help information uses < > brackets for items which would be shown in italics in this user guide.

LANGUAGE Command

This command is used to tell the debugger what language rules it should obey. Currently, the only variation in the way the various languages are handled is whether the case of identifiers is relevant or not. C is case sensitive; Pascal and FORTRAN-77 are not. The syntax of the command is

`lang[uage] [`*`language`*`]`

where *`language`* is one of `f77`, `c` or `pascal`. The default (which is reverted to if the argument is omitted) is the language that the program's entry module is written in.

OBEY Command

This command executes a set of debugger commands from a file, as if they had been typed at

the keyboard. It has the form

`obey` *`filename`*

This command would be identical to the * command:

`*exec` *`filename`*

but for one difference. The `obey` command uses the newline character (ASCII 10) to mark the end of the line. `*exec` uses carriage return (ASCII 13). Thus if the command file was created using !Edit, which uses newline, then `obey` should be used. If it was created using `*BUILD`, which uses carriage return, then `*exec` should be used.

LOG Command

This command causes subsequent typed commands, and their output, to be sent to a file as well as the screen. The format of the command is:

`log [`*`filename`*`]`

To start logging, use the form with the *`filename`*, eg:

```
log logfile
```

The file will be opened, and a couple of introductory lines sent to it. Thereafter, all user input and command output (excluding `ASD:` prompts) is sent to the file.

To terminate logging, type `log` without an argument.

The `log` command is useful for capturing the output after a `ptrace` command, enabling the flow of control to be examined at leisure using an editor such as !Edit.

QUIT Command

This causes the debugging session to be terminated. It also closes any open `log` and `obey` files. The syntax is

`q[uit]`

* Command

Any command whose first non-space character is * will be sent to the operating system for execution. This gives access to the RISC OS command line interpreter (CLI). For example, function keys may be programmed with common command strings, and the filing system may be used. Examples are

```
*key1 p/hex ptr|m
*cat f77
```

An Example of Using ASD

The ASD can only be used with programs linked by the older linker (f77link). Refer to Appendix L - Notes on Using a Debugger in the Desktop Fortran77 User Guide.

The example program is very simple and the fault it contains is easy to spot but it serves as an introduction to the use of ASD for debugging FORTRAN programs. The program can be found in the FORTRAN.Examples directory on the distribution disc.

The program is supposed to print out the date of Easter Sunday for the years 1970-1989. Note that the line numbers are not part of the program but are included to enable references to be made in the text below.

```
 1          PROGRAM EDATE
 2          INTEGER Y, DAY
 3          CHARACTER *(5) MONTH
 4
 5          DO Y = 1970,1989
 6            CALL EASTER DATE(Y, DAY, MONTH)
 7            PRINT '(I4, 1X, I2, 1X, A)', Y, DAY, MONTH
 8          END DO
 9          END
10
11          SUBROUTINE EASTER DATE(YEAR, DAY, MONTH)
12          INTEGER YEAR, DAY, G, C, X, Z, D, E, N
13          CHARACTER *(*) MONTH
14
15          G = MOD(YEAR, 19) + 1
16          C = YEAR / 100 + 1
17          X = (3 * C) /  4 - 12
18          Z = (8 * C + 5) / 25 -  5
19          D = (5 * YER) /  4 - X - 10
20          E = MOD(11 * G + 20 + Z - X, 30)
21
22          IF (E .EQ. 25 .AND. G .GT. 11 .OR. E .EQ. 24) THEN
23            E = E + 1
24          ENDIF
25
26          N = 44 - E
27          IF (N .LT. 21) N = N + 30
28          N = N + 7 - MOD(D+N, 7)
29
30          IF (N .GT. 31) THEN
31            DAY   = N - 31
32            MONTH = 'April'
33           ELSE
34            DAY   = N
35            MONTH = 'March'
36          ENDIF
37          END
```

If this program is run, the results at first sight seem correct - the date for 1988 (3 April) is right. However, the date shown for 1989 is incorrect - the true result is 26 March. The result for 1970 is also wrong - it should be 29 March.

An example session using the debugger follows. Commands to be typed are shown in bold

text and responses produced by the computer are in normal text.

Open the directory viewer on FORTRAN.Examples and double click on !SetDir. Press <F12> and to compile and link the program with debug information included type on the command line

```
*df77lk edate
Topexpress FORTRAN 77 front end version 1.19
Program    EDATE Compiled
Subroutine EASTERDATE Compiled

2 program units compiled.
Total workspace used 6016
ARM FORTRAN 77 code generator version 1.62
Main program (EDATE): code 196; data 60
EASTERDATE: code 432; data 40
Total code size: 628; data size: 100
```

To enter the debugger type

```
*cache off (StrongARM only)
*asd aif.edate
ARM symbolic debugger, version 1.00
Object program file aif.edate
```

The first step might be to check that the results are printed correctly. A break point is set on the PRINT statement in the main program (line 7)

```
ASD: brk edate:7
```

`edate` in the `brk` command is the main program name. If the program had not started with a PROGRAM statement, the default name `F77_MAIN` would have been used instead. Note that upper and lower letters in names are not distinguished when debugging FORTRAN programs, so the `brk` command could have been typed in any of the following forms

```
ASD: BRK EDATE:7 or
ASD: brk Edate:7 or
ASD: brk EdAtE:7 or
ASD: BRK eDaTe:7 etc
```

The program is run until the PRINT statement is reached. The value of day is then displayed.

```
ASD: go
Program stopped at breakpoint #1, location edate:7
ASD: print DAY
27
```

The value `27` shows that the fault lies in `EASTER DATE` - the correct result is `29`. The next step is to trace through the subroutine.

```
ASD: brk EASTERDATE
ASD: go
1970 27 March
Program stopped at breakpoint #2, location easterdate
ASD: print year
1971
```

The print shows that the argument (YEAR) is correct. A simple step might be to set a break

point after the initial calculations, on line 22

```
ASD: brk 22
ASD: go
Program stopped at breakpoint #3, location 22
ASD: print G
15
ASD: print C
20
ASD: print X
3
ASD: print Z
1
ASD: print D
-13
ASD: print E
3
```

The value of D (-13) looks a bit suspicious so the value of the expression is displayed

```
ASD: print (5*YEAR/4)/4-X-10
2450
```

2450 is not the same as -13. Inspection of the expression for D (line 19) shows that YEAR was mistyped as YER. D was therefore computed using an undefined value. To check whether this is the only problem, D can be set to the correct value (it is not used in an expression until line 28) and execution resumed

```
ASD: let D=2450
ASD: unbreak #1
ASD: go
1971 11 April
Program stopped at breakpoint #2, location easterdate
```

The break point on line 7 was cleared so that a result would be printed without stopping. The new result for 1971 is correct. If the program is altered and recompiled, the results for each year are now correct.

To end the debugging session type

```
ASD: quit
Quitting
*
```

and press <Return> to return to the window environment.

Command Summary

This appendix lists all of the ASD commands in alphabetical order with a brief description and page reference for the appropriate section in this user guide.

`arguments`	14	Display arguments of current procedure
`backtrace`	15	Display stack-frame history
`break`	16	Set a break point or display all break points
`cmdline`	19	Set the application's command line arguments
`context`	14	Set or reset the current context
`format`	12	Set default print format for integers
`go`	16	Start or resume execution of the program
`help`	19	Display general or specific help information
`in`	14	Set context to current context's caller
`language`	19	Set current language name
`let`	12	Assign value to a variable
`log`	20	Open a log file storing ASD commands and output
`obey`	19	Execute command lines stored in a text file
`out`	14	Set context to current context's caller
`print`	12	Display result of an arbitrary expression
`ptrace`	18	Enable or disable procedure tracing
`quit`	20	Leave the debugger, returning to the OS
`return`	18	Return from active procedure, with optional result
`step`	16	Single step by one of n statements
`symbols`	13	Display variables in current context
`unbreak`	17	Clear a break point
`unwatch`	18	Clear a watch point
`variable`	13	Display information about a variable
`watch`	17	Set a watch point or display all watch points
`where`	15	Display current context