cpr

COLLABORATORS			
	TITLE : cpr		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

cpr

1.1 CPR Help

```
CPR Commands and Builtin-Functions
 Activate      ALias         ARgs          BClear
 BDisable      BEnable       BList         Break
 Call          CAtch         DEActivate    DEFine
 DETach        Display       DUmp          DZero
 ECho          ENV           EXecute       EXPAND
 FINish        FRegister     Go            Help
 HUnks         Jump          ListSym       List
 LOg           MEMCMP        MEMCPY        MEMMOVE
 MEMSET        Modules       OPt           Proceed
 PS            Quit          Register      RESTart
 RETurn        RFlag         SEArch        SEt
 SHow          SLEep         SOurce        STArt
 STRCAT        STRCMP        STRCPY        STRLEN
 SYMBol        SYMload       TAsks         Trace
 TS            UNAlias       Unassemble    Watch
 WBreak        WClear        WDisable      WEnable
 WHAtis        WHEre         WIndow        WList
 WMSG

 Parameter Types      Common Problems
```

1.2 parameter types

```
 Array-Slice
 Address
 Expression
 Itemsize
 Location
 Number
 Range
 Register
 String
 Subrange
 Task-Address
```

```
Task-Name
Type
Variable

Builtin Function Parameters
```

1.3 ADDRESS

An address parameter is any expression that denotes an address. The expression can be any of the following:

- a hexadecimal constant
- a C pointer variable
- a register
- the result of prefixing & to a C identifier of the correct type (not a bitfield or register identifier)
- the result of arithmetic calculations on other addresses
- any C expression that evaluates to a pointer type.

The following list shows expression that denote addresses. In these expressions, p is a pointer, and i and x are integers.

```
&i
(&i + 8)
&array[3]
&mystruct - x
p
0x00C85400
a0
(a0 + 0x22)
__this
myclass::memberfunc
```

In some cases, a register name may be ambiguous. For example, you may have defined a variable sp in your program. In the following command, CodeProbe cannot tell whether you want to dump data beginning at the variable sp or at the address contained in the register SP:

```
db sp
```

In these cases, the debugger assumes that you are referring to to the variable sp. However, you can prefix the register name with a dollar sign ($) or use the register command to examine the value of the SP register even if you have declared a variable named sp.

In some commands, it is necessary to specify the type of an object to be modified or displayed. If the object is referred to using an address constant or register, the value is assumed to be a character pointer (char *).

1.4 ARRAY-SLICE

An array-slice parameter is one of the following:

- a contiguous section of a unidimensional array of scalar variables
- an ordered selection of elements from a multidimensional array
- an ordered selection of members from an array of complex types.

You can use an array-slice anywhere an array is valid. You can specify an array-slice in either of the following formats:

```
variable [n..m]
variable [*]
```

NOTE: The inner brackets do not denote optional parameters and must be entered as shown if you want to specify index numbers or the asterisk (*).

The advantage of using an array-slice parameter over a range parameter (described later in this chapter) is that the array-slice can be used to select isolated subcomponents, such as a particular structure member from an array of structures. The array-slice parameter can also be nested for multidimensional arrays or structure members that are arrays themselves.

If you specify a subrange (as described later in this chapter) as the index of an array-slice, those elements within the subrange are selected. The asterisk (*) index specifies that all elements of the array are to be contained in the array-slice. The following are examples of the array-slice parameter:

```
a[3..7]
matrix[i..j][1..3]
b[*]
a[3..6]->b
```

1.5 EXPRESSION

Any C expression is accepted as an expression parameter, except those that use one or more of the following operators:

```
++ or --
,
?:
=
&=
|=
>>=
<<=
+=
-=
*=
```

```
    /=
    %=
    ^=
```

C++ expressions are not supported, so you cannot use overloaded operators in an expression in a debugger command.

The following are valid expressions:

```
    a[i*8]
    p->d[5]+f(3)
```

1.6 ITEMSIZE

ITEMSIZE is an optional parameter that controls the length of each item dumped by the DUMP command. ITEMSIZE can have a value of 1, 2, 4, or 8. ITEMSIZE is concatenated onto the end of the format parameter (for example, b4 for a binary format of length 4). If you do not specify ITEMSIZE, CodeProbe uses a default. As shown in the following table, the allowable size and default values depend on the format specified.

ITEMSIZE has the following default values and allowable sizes:

Format	Default	Allowed Sizes
ascii	1	1
binary	1	1, 2, 4
decimal	4	1, 2, 4
FFPfloat	4	4
float	8	4, 8
hex	4	1, 2, 4
IEEEfloat	8	4, 8
octal	4	1, 2, 4
unsigned	4	1, 2, 4

1.7 LOCATION

A location parameter specifies a place in the code at which a breakpoint is to be set, code is to be unassembled, or a similar action is to be performed. When debugging in source mode, a location parameter is a line number in a source file or a function entry point. When debugging in assembly mode, you may want to place breakpoints at specific addresses. To do this, use a hexadecimal address as the location parameter.

You can specify a location parameter as follows:

$
   specifies the current location in the current executable, module, and function.

hex-address

```
   can be any valid absolute address specified as a hexadecimal
   integer.

[[executable-name:][\module\] function] [line]
   can identify any location in any executable.  The values for
   executable-name, module, and function can be specified explicitly
   as follows:

      executable-name:
   is the name of the executable image (the program, library,
   or device) containing the location.  It is usually
   specified in lowercase. In a program with only one
   executable image, you do not need to specify an
   executable-name:.  If you specify an executable image,
   follow the name with a colon (:).  The default executable
   image is the current executable.

      \module\
   is the name of the C source file compiled to yield the
   specified function.  If you specify a module name, follow
   the name with a backslash (\).

      function
   is the name of a function in the application.  For C++, the
   function can be a member function, operator, constructor,
   or destructor, but you need to precede the function name
   with the name of the class for which the function is
   defined, as follows:  p myclass::function
   myclass::operatorx myclass::myclass myclass::~myclass


      line
   can be any of the following:

   integer
      a line number, relative to the start of the C source
      file containing the function.

   $
      the current location.

   e[ntry]
      the entry to the function (prolog).

   r[eturn]
       the return from the function (epilog).

You must enter a space before the line parameter.  All other spaces
and tabs are ignored.  If you do not specify the name of an
executable, module, and/or function, the debugger uses the names
from the current environment.  (For more information, see the
description of the env command in Chapter 9.) If you specify a
function name but do not specify a line, the debugger uses the the
first execution line of the specified function.

The following are examples of locations:
```

```
main
test.c\main
myprog:test.c\main
example.device:\serial.c\cmd_handler 70
myclass::myfunc
myclass::operator+
mycxxprog:myclass::myfunc 12
```

1.8 NUMBER

A number parameter is a number in decimal, octal, or hexadecimal notation. If you begin the number with a 0 digit, the number is interpreted as an octal number. If you begin the number with 0x or 0X, the number is interpreted as a hexadecimal number, and if you begin the number with 0n or 0N, the number is interpreted as a decimal number.

By default, the number is interpreted as a decimal number. You can use the opt radix command to change the default.

The following examples show acceptable values for number parameters:

```
12345
0455
0x380
0X1849
0n12345
```

1.9 RANGE

A range parameter is a contiguous area of memory, and you can specify a range in one of two ways:

```
start-address. . end-address

start-address l|L length
```

In the second form, the length is a number (as described earlier) that indicates the number of elements, and the range is from the start address through address + number -1. When the debugger sees a solitary l or L in a command, it is considered part of a range expression.

For example, the following pairs of ranges are equivalent:

```
0x123456 .. 0x123461
0x123456 L 11

&p[0] .. &p[5]
&p[0] l 6
```

```
   p[0] .. p[5]
   p[0] l 6
```

The following examples use the names of address registers:

```
   a1 .. a2
   a7 l 20
```

1.10 REGISTER

A register parameter refers to the name of any of the following 680x0 registers:

A0 - A7
are address registers. Register A7 functions as the stack pointer and can be specified as either A7 or SP.

D0 - D7
are general-purpose registers.

PC
is the program counter.

SP
is the stack pointer.

SR
is the status register. It contains the CCR register as well as the current processor status.

CCR
is the condition code register. It resides in the lower byte of the status register, SR.

If a math coprocessor is present, the following registers also can be specified as the register parameter:

FP0 - FP7
are the floating-point registers.

FCR
is the floating-point control register.

FSR
is the floating-point status register. It contains the current floating-point condition codes.

In some cases, a register name may be ambiguous. See the description of the address parameter, earlier in this chapter, for more information.

1.11 STRING

A string parameter can be any standard C or C++ string in double quotes ("):

```
"string"
```

You can continue a string onto the next line by ending the line with a backslash (\). The debugger supports ANSI string concatenation: two strings adjacent to each other will be combined into one longer string.

In strings, the debugger supports the C escape sequences shown in Table 1.5.

Escape Sequences in Character Strings and Character Contstants

Escape Sequence	Meaning
\n	newline (0x0a)
\t	horizontal tab (0x09)
\b	backspace (0x08)
\r	carriage return (0x0d)
\f	form feed (0x0c)
\v	vertical tab (0x0b)
\NNN	octal constant, where NNN are 3 octal digits
\xNN	hex constant, where NN are 2 hexadecimal digits

A backslash followed by any character other than those shown previously is interpreted as a plain character. For example, \ is a string consisting of a single backslash, and a\"b is three characters long: an a, a double quote, and a b.

The Dialog window does not recognize any of these escape sequences when they are displayed. You cannot use the \t escape sequence to tab in the Dialog window. The escape sequences are useful in line mode and when setting a character string.

1.12 SUBRANGE

A subrange parameter is a series of contiguous integers, inclusive of its bounds. The integers may be either integer constants or variables from the program being debugged. You can specify a subrange parameter as follows:

```
[integer | variable] . . [integer | variable]
```

The following are examples of the subrange parameter:

```
3..7
i..j
```

1.13 TASK-ADDRESS

TASK-ADDRESS is the address of the executable for a task. TASK-ADDRESS can be determined from the output of the TASKS command. The following example is a partial listing of output from the "tasks all" command. Note that TASK-ADDRESS is listed on the far left.

```
> tasks all
Address Type Pri State  SigWait  StackPtr Debug Name
003D2498 13   0 Running 00000100 003D1D4E Cpr
002BED58 13   0 Waiting 80001000 002BFC66 iprefs 37.8 (31.5.91)
0035E9E0 13   0 Waiting C0003000 00371B66 Background CLI
00287790  1   5 Waiting F0000000 00287736 console.device
00295498 13  10 Waiting 40000100 00295E5A DF2
0029DA98 13  10 Waiting 40000100 0029E45A Work
```

1.14 TASK-NAME

TASK-NAME is the name of a task. TASK-NAME can be determined from the output of the TASKS command. The following example is a partial listing of output from the "tasks all" command. Note that TASK-NAME is listed on the far right.

```
> tasks all
Address Type Pri State  SigWait  StackPtr Debug Name
003D2498 13   0 Running 00000100 003D1D4E Cpr
002BED58 13   0 Waiting 80001000 002BFC66 iprefs 37.8 (31.5.91)
0035E9E0 13   0 Waiting C0003000 00371B66 Background CLI
00287790  1   5 Waiting F0000000 00287736 console.device
00295498 13  10 Waiting 40000100 00295E5A DF2
0029DA98 13  10 Waiting 40000100 0029E45A Work
```

1.15 TYPE

A type parameter can be any of the data types provided by the C language, including:

```
char
unsigned char
short
unsigned short
int
unsigned int
long
unsigned long
unsigned
float
double
struct name
union name
enum name
class name
```

A type also can be any of these C types followed by some number of asterisks indicating that the type is a pointer to the base object.

In addition, the type parameter can take any identifier defined by means of a typedef statement if the debugging information for the module supplies typedef information.

CodeProbe considers an int to be four bytes, regardless of whether you compiled your program with the shortint option. Use short to refer to objects declared int in code compiled with shortint.

1.16 VARIABLE

A variable parameter denotes a data object. The variable can be either a simple identifier or an expression referring to an array element, structure member, or object pointed to by a pointer. You specify variables as follows:

```
[[executable-name:][\module\]function\]variable
```

See the location parameter earlier in this chapter for a description of the executable-name, module, and function.

The following are valid examples of variable parameters:

```
i
max
array[3]
array
io:readfile\length
mystruct->name[2]
mystruct
*cptr
main.c\opnf\count
opnf\i
mylib.library:LIBmyfunc\myvar
myobject.mypublicmember
myobjectptr->mypublicmember
mycxxprog:myclass::myfunc\mylocalvar
```

In C programs, if you have more than one variable with the same name, CodeProbe uses the variable that is in scope.

In C++ programs, a local variable in a member function and a data member of the object that invoked the function can have the same name. For example, if the variable name is myvar, you can display the local variable with:

```
display myvar
```

To display the data member of the class that invokes the function, use the object's this pointer:

```
display __this->myvar
```

```
CodeProbe cannot display variables that have the same name as a
register but begin with a dollar sign such as $d5.
```

1.17 builtin function parameters

```
The parameters to CodeProbe's built-in functions are a little more
flexible than the parameters to the C versions:

   formal type       actual parameter types allowed
   -----------       ------------------------------
     void *      constant, register, any pointer scalar,
         array, function, address, string constant

     char *      constant, register, character pointer,
         unsigned character pointer, character array,
         string constant

      int      constant, register, any integral scalar,
         bitfield, enumerated constant

Floating-point constants and registers are not allowed, and string
constants can only be specified as the source operand.
```

1.18 Activate

```
Activate -- Activates a task under debugger control

SYNOPSIS
  a[ctivate] [ task-name  |  task-address ]

DESCRIPTION
The activate command activates a task that was deactivated by the
deactivate command. The task is activated when the next go, proceed, or
trace command is executed.

The  task-name  is the name of the task, and  task-address  is the
address of the task as specified by the tasks command.  You can specify only
one of these parameters.

EXAMPLES
  activate "Child"
      activates the task named Child under debugger control.

  activate 0x7D9F884
      activates the task with a task block starting at address
      0x7D9F884.

SEE ALSO
 catch ,  deactivate ,  detach ,  tasks
```

1.19 ALias

```
ALias -- Define an alias for a debugger command

SYNOPSIS
 al[ias]  [name[definition]]

DESCRIPTION

If you do not specify any arguments, the alias command displays the
list of all currently defined aliases.  If you specify a name as the
only argument, alias displays the definition for that name.  Any
text following the name parameter is treated as a definition and is
used to define the alias.

For a complete description of aliases, see Chapter 2, "Customizing
the Debugging Environment."

EXAMPLES
    alias
  displays a list of all alias definitions.

    alias foo
  displays a definition of foo.

    alias next proceed
  defines next to be an alias for proceed.

    alias pr display foo, bar, $*
  defines pr to be an alias that executes the display command
  printing the value of foo, bar, and any variables entered
  after the pr.

    alias doit {go $1; d foo}
  defines doit to execute the go command using the first
  parameter, and displays the variable named foo.

    alias doit {go $1;\
    d foo}
  uses the backslash command to split a command across two
  lines.

    alias doit "go $1; d $2"
  uses positional parameters with multiple commands. The first
  parameter, $1, is passed to the go command, and the second
  parameter, $2, is passed to the display command.

    alias this display *__this
  defines this to display the object pointed to by the this
  pointer in C++ when CodeProbe is inside of a C++ member
  function.  The C++ this pointer points to the object that
  invoked the member function.


SEE ALSO
 DEFine ,  UNALias ,  UNDefine
```

1.20 ARgs

```
ARgs -- Displays the arguments to a function

SYNOPSIS
    ar[gs]  [function-name]

DESCRIPTION

The args command displays all of the argument names and values to
the current function or the function specified by the function-name
parameter. If you specify a function-name, the function must be in
the calling sequence, as displayed by the where command.

EXAMPLES
    args
  displays the arguments to the current function.

    args sort
  displays the arguments to the sort function.

    args chessBoard::movePiece
  displays the arguments to the C++ member function
  chessBoard::movePiece.


SEE ALSO
 Display ,  ENV ,  WHEre
```

1.21 BClear

```
bclear-Clears (deletes) one or more breakpoints

SYNOPSIS
   bc[lear]  integer  [ integer...]
   bc[lear]  *|l[ast]
   bc[lear]  integer..integer


DESCRIPTION

The bclear command clears (deletes) one or more breakpoints.  When a
breakpoint is cleared, it ceases to exist and can be reinstated only
by issuing the break command again.  To disable breakpoints
temporarily, use the bdisable command.

The integer parameter specifies the breakpoint number as displayed
by the blist command.  You can specify as many breakpoint numbers as
needed.  An asterisk clears all breakpoints, and last clears the
most recently set breakpoint.  You can specify a range of
breakpoints with integer..integer.

EXAMPLES
    bclear 2 5 6
```

```
  clears the breakpoints numbered 2, 5, and 6.

    bclear last
  clears the last breakpoint set.

    bclear *
  clears all breakpoints.

    bclear 4..7
  clears breakpoints 4, 5, 6, and 7

    bclear 3..4 last 9
  clears breakpoints 3 and 4, the last breakpoint that was
  set, and breakpoint 9.

SEE ALSO
 BDisable ,  BEnable ,  BList ,  Break ,  WClear
```

1.22 BDisable

```
bdisable - Disables (turns off) one or more breakpoints

SYNOPSIS

bd[isable]  integer  [integer . . .]
bd[isable]  *|l[ast]
bd[isable]  integer..integer

DESCRIPTION

The bdisable command disables (turns off) one or more breakpoints.
When a breakpoint is disabled, it is not recognized by CodeProbe,
but it remains on the list of current breakpoints. To re-enable a
disabled breakpoint, use the benable command.  To remove the
breakpoint from the list, use the bclear command.

The integer parameter specifies the breakpoint number as displayed
by the blist command.  You can specify as many breakpoint numbers as
needed.  An asterisk disables all breakpoints, and last disables the
most recently set breakpoint.  You can specify a range of
breakpoints with integer..integer.

EXAMPLES
    bdisable 2 5 6
  disables the breakpoints numbered 2, 5, and 6.

    bdisable last
  disables the last breakpoint set.

    bdisable *
  disables all breakpoints.

    bdisable 4..7
  disables breakpoints 4, 5, 6, and 7.
```

```
    bd 3..4 last 9
  disables breakpoints 3 and 4, the last breakpoint that was
  set, and breakpoint 9.

SEE ALSO
 BClear ,  BEnable ,  BList ,  Break ,  WDisable
```

1.23 BEnable

```
benable - Enables (turns on) one or more breakpoints

SYNOPSIS

be[nable]  integer  [integer ...]
be[nable]  *|l[ast]
be[nable]  integer..integer

DESCRIPTION

The benable command enables (turns on) one or more breakpoints that
have been disabled by the bdisable command.

The integer parameter specifies the breakpoint number as displayed
by the blist command.  You can specify as many breakpoint numbers as
needed.  An asterisk enables all breakpoints, and last enables the
most recently set breakpoint.  You can specify a range of
breakpoints with integer..integer.

EXAMPLES
    benable 2 5 6
  enables the breakpoints numbered 2, 5, and 6.

    benable last
  enables the last breakpoint set.

    benable *
  enables all breakpoints.

    benable 4 .. 7
  enables breakpoints 4, 5, 6, and 7.

    be 3..4 last 9
  enables breakpoints 3 and 4, the last breakpoint that was
  set, and breakpoint 9.

SEE ALSO
 BClear ,  BDisable ,  BList ,  Break ,  WEnable
```

1.24 BList

```
blist - Lists all breakpoints Synopsis
```

```
SYNOPSIS
  bl[ist]

DESCRIPTION
The blist command displays one or more breakpoints.  The integer
parameter specifies a breakpoint number.  You specify one breakpoint
or a range of breakpoints.  If you specify last, then blist displays
the last breakpoint set.  If you specify an asterisk (*) or if you
do not specify any parameters, blist displays the list of all
breakpoints, as shown in the following example:

   1  0x1CA8 example:\example.c\main 38 (1 hit)
       after(3) when(i==4)
   2  0x1E14 example:\example.c\initarr 62 (1 hit)
   3* 0x1D70 example:\example.c\main 43
   4  0x1DA4 example:\example.c\main 45
       trace {echo Hi}

The numbers 1, 2, 3, and 4 are breakpoint numbers.  You can use
these numbers to identify a specific breakpoint in the bclear,
bdisable, and benable commands.

If the breakpoint number is followed by an asterisk (as in
breakpoint number 3), the breakpoint is disabled and will not be
triggered until you enable it with the benable command.  Following
the breakpoint number is the hexadecimal address at which the
breakpoint resides.  If the executable module contains sufficient
debug information, the hexadecimal address is followed by the
location that it represents.  For example, breakpoint 1 resides at
address 0x1CA8, which occurs at line 38 of the main program.  The
hit count is the number of times that the breakpoint has been
triggered.  If any options (such as after or trace) are associated
with the breakpoint, these commands are displayed on the line
following the breakpoint number and address.


SEE ALSO
 BClear ,  BDisable ,  BEnable ,  Break ,  WList
```

1.25 Break

```
Break -- Sets a breakpoint

SYNOPSIS
b[reak]   location  [after(integer)] [when( expression )]
   [tr[ace]][q[uiet]] [te[mp]] [{cmd-list}]

DESCRIPTION

The break command sets a breakpoint at a location specified by the
location parameter. You can use a dollar sign to specify the current
location.

If you specify a C++ overloaded function name as the location,
the debugger displays a requester that allows you to select
```

```
the function on which you want to set a breakpoint.  You can click
on the function name or use the arrow and Return keys to select
a function.

When you are setting a breakpoint on a C++ member function or
operator member function, you must specify the class, followed by
two colons, and then the member function or operator member function
name.

The break command supports the following options:

    after
  specifies the minimum number of times the specified line
  must be executed before the breakpoint is triggered.  The
  integer you specify with the after option is called the pass
  count.  Each time the breakpoint is hit, the pass count is
  tested.  If the pass count is greater than 1, the pass count
  is decremented by 1, and execution continues.  The
  breakpoint is triggered when the pass count equals 1.  For
  example, if you specify a pass count of 5, the breakpoint is
  triggered when the line is executed the 5th time. For break
  commands that do not contain an after clause, the pass count
  is set to 1.  The blist command shows the current pass count
  for each breakpoint.

    when
  specifies an expression that must evaluate to true (nonzero)
  before the the breakpoint is triggered.  You can specify the
  after and when options in any order.  If you specify both
  options, the pass count is tested and decremented only if
  the when condition is true.

    trace
  continues execution automatically after the breakpoint is
  triggered, unless you reach the breakpoint by
  single-stepping.  If you reach the breakpoint by
  single-stepping, then the the trace option has no effect.

    quiet
  suppresses the default message showing the location when the
  breakpoint is triggered.

    temp
  deletes the breakpoint after it has been triggered.

    {cmd-list}
  specifies commands that you want the debugger to execute
  each time the breakpoint is triggered. You can use the
  backslash (\) to continue the command list onto more than
  one line.

NOTE:  To avoid confusion, the break command does not use the
       default radix setting. You must specify 0x for an address;
       otherwise, you will be specifying a line number. For example,
       even if the radix is set to hex, the following command
       specifies a break command for line 20 and not address 0x20:
```

```
    b 20

A breakpoint is triggered when all of the following conditions are
true:

    >  execution has reached the address shown
    >  the when and after conditions, if specified, are true,
    >  the breakpoint is enabled.

CAUTION!!
     Incorrect use of breakpoints may crash your machine.

     A breakpoint is implemented by placing an illegal instruction
     at the desired location.  All tasks under the debugger's
     control have a trap handler that reports back to the debugger
     when the illegal instruction is executed.  If you must place a
     breakpoint in shared code, such as a resident library under
     test, be sure that any task that might open the library is
     under debugger control.  Never place breakpoints in libraries
     that you do not control.

EXAMPLES
   break $
  sets a breakpoint at the current line.

    b 14
  sets a breakpoint at line 14 of the current module.

    b sort
  sets a breakpoint at the first line of the sort function.

    break sort return
  sets a breakpoint at the return from the sort function.

    break \myfile.c\sort 14
  sets a breakpoint at line 14 of sort in the module named
  myfile.c.

    break mylib.library:myfunc
  sets a breakpoint at the first line of the myfunc function
  in the shared library mylib.library.

    b $ after(5)
  breaks the fifth time the current line is executed.

    break 0x804A
  sets a breakpoint at the absolute address 0x804A.

    break 14 when (i > 5) {di p L 16; b sort 26
    when (i == 8); bl; go}
  sets a breakpoint on line 14 that performs the commands
  given inside the curly braces if i > 5 evaluates true.

    b 8 trace quiet {d "i = ", i}
  prints a message of the form "i = value " every time line 8
  is executed.
```

```
    break 100 temp {d foo}
  stops at line 100, prints foo, and deletes this breakpoint.

    break myclass::myfunc when(i == 5)
  sets a breakpoint on the member function myfunc in class
  myclass when i > 5 evaluates to true.

    break myclass::operator+ and break myclass::+
  set a breakpoint on operator + in myclass.  Do not enter a
  space between the keyword operator and the operator itself.

    break myclass::myclass
  sets a breakpoint on a constructor for myclass.

    break myclass::~myclass
  sets a breakpoint on a destructor for myclass.


SEE ALSO
 BClear ,  BDisable ,  BEnable ,  BList ,  Go
```

1.26 Call

```
Call -- Evaluates an expression and discards the result

SYNOPSIS
  c[all]  expression

DESCRIPTION
The call command evaluates an expression and discards the result.
This command is useful when you do not care about the value returned
by an expression and is equivalent to the following statement in C
language:

       expression ;

The most common use of the call command is to call a function in
your program, though function calls also may appear in arbitrary
expressions in other commands.  For example, you may want to write a
function to display the contents of a complex structure and call
that function at various points during the execution of your
program.

If you want to see the value returned by an expression, use the
display command.

The debugger looks at the types of the parameters specified and not
at the definition of the function or any prototype.  Thus, the
debugger does not flag an error if the wrong number of parameters
are specified, nor does it perform automatic casting of types or
allow passing of char, short, or float types.  For example, the
following command passes two parameters, an int and a double,
regardless of how the function is declared:

   call f(2, 1.5)
```

```
CodeProbe supports standard C syntax for function calls.  Either a
function name or an expression evaluating to a function type may be
specified before the opening parenthesis.  Thus, the call command
can be used only to call functions or evaluate expressions that
point to a function.  For example, if pf is a function pointer, and
func is a function, the following commands are allowed:

   call func()
   call (*pf)()

However, the following commands are not allowed:

   call pf()
   call 0x134()
   call register()

You cannot use the call command to execute a function after your
program has ended.

If a breakpoint is hit, a signal is caught, or the program
terminates in the middle of a function call, the debugger aborts the
expression and leaves you at that location.  A warning is printed if
any parameters are not cleaned off the stack.  This may cause a
return to an invalid location later if execution is allowed to
continue.  If this happens, you should enter the restart command.

EXAMPLES
    call status()
  calls the status function.

    call print("Test", 300)
  calls the print function passing in a string and an integer
  parameter.

    call (*fp)(&arr[5], j+10, 3.0)
  evaluates arbitrary expressions.

SEE ALSO
 Display ,  ENV ,  WHEre
```

1.27 CAtch

```
catch - Places a task under debugger control Synopsis

SYNOPSIS
  ca[tch] [ \

DESCRIPTION
The catch command places a task under the debugger's control
(catches a task).  You can use this command to gain control of tasks
that were not started under the debugger.  For example, you may find
that a process or task not started under the debugger is behaving in
an unpredictable or undesired manner.  The task may be caught in an
infinite loop or waiting on some message port for a message that
```

```
will never come.

The task-name and address are the task and address as displayed by
the tasks all command.

If the task that you want to catch is a process, this command uses
the process structure to find the code segments associated with the
executable module.  If you are running SegTracker and the task you
catch was launched after SegTracker, just enter symload, and
CodeProbe will automatically find the seglist.  If you are not
running SegTracker, you must locate the task's seglist yourself and
specify it on the symload command.

For more information on segment lists, see The AmigaDOS Manual, 3rd
Edition.

EXAMPLES
    catch "Child"
  catches the task named Child.

    ca 0x7D9F884
  catches the task with a task block starting address
  0x7D9F884.

SEE ALSO
 Activate ,  DEActivate ,  DETach ,  SYMload ,  TAsks
```

1.28 DEActivate

```
DEActivate -- Deactivate a task under debugger control

SYNOPSIS
  dea[ctivate] [ \

DESCRIPTION

Normally, the debugger starts all tasks under its control when you
enter a go or proceed command.  You can use the deactivate command
to prevent a task from running, unless it is the current task.  You
can reactivate the task with the activate command.

The task-name and address are the task name and address as displayed
by the tasks command.


EXAMPLES
    dea "Child"
  deactivates the task named Child that is running under
  control of the debugger.

    deactivate 0x7D9F884
  deactivates the task with a task block starting at address
  0x7D9F884.

SEE ALSO
```

Activate , CAtch , DETach , TAsks

1.29 DEFine

```
DEFine -- Defines a macro

SYNOPSIS

DESCRIPTION

The define command provides a general macro substitution facility
that is nearly identical to the mechanism provided by the C
preprocessor.  Macros defined in the debugger are expanded anywhere
within a line of text except when

    >  the macro occurs inside quotes or filenames
    >  the macro is prefixed with a backtick character (`).

The # sign is optional and allows the debugger to read C header
files (.h files) using the execute command.

The define command does not support the ANSI C # and ## operators.

EXAMPLES
    define
  displays all the macro definitions.

    define foo
  defines foo as a null definition.

    def foo bar
  defines foo to be bar.

    define func(a,b) (a+b)
  defines func with parameters.

SEE ALSO
 ALias ,  UNALias ,  UNDefine
```

1.30 DETach

```
detach - Frees a task from debugger control

SYNOPSIS
  det[ach] [ \

DESCRIPTION
The detach command frees a task from debugger control.

You may want to allow one or more tasks spawned by a task under the
debugger to run freely and not under the control of CodeProbe. For
example, a task designed to respond to Intuition events may cause
```

```
the system to lock up if it does not respond to menu events quickly.
You may not want such a task to be stopped when other tasks hit a
breakpoint.

The task-name and address are the task name and address as displayed
by the tasks command.

To use the detach command, first set a breakpoint at the entry point
to the task.  CodeProbe will hit this breakpoint when the task is
launched, which allows you to detach the task right away.  When the
breakpoint is reached, use the tasks command to identify the address
or name of the task (you may know this based on your code).  Then,
clear all the breakpoints in the task, including the breakpoint that
you set at the beginning of the task. The task will crash if it hits
a breakpoint after it has been detached.  After you clear all the
breakpoints, enter the detach command with the task name or address.
Also, do not detach a program if any of its child processes continue
to run in the same code segment.

The detached task is not removed automatically when you quit the
debugger, and it can continue running even after you exit CodeProbe.
Since CodeProbe frees all of the memory containing the code used by
the detached task, the program will be executing in freed memory,
and it may crash.  Make sure that the program runs to completion
before you quit the debugger.

If you later want to re-attach the task to the debugger, you can use
the catch command.

EXAMPLES
    detach "Child"
  detaches the task named Child from debugger control.

    det 0xC08540
  detaches the task with a task block starting at address
  0xC08540.

SEE ALSO
 Activate ,  CAtch ,  DEActivate ,  TAsks
```

1.31 Display

```
display - Displays the value of an expression Synopsis

SYNOPSIS
  d[isplay]   expression  ["format"][,  expression  ["format"]]...
  d[isplay]   string [,  string ]...

DESCRIPTION
The display command evaluates the value of the expression parameter
and displays the result. The expression parameter can be any C
expression containing constants, variables, and function calls from
the program being debugged. Any number of expressions and strings
can be specified, separated by commas. All expressions are displayed
on the same line.
```

If you do not specify a format, the display command chooses the default format based on the type of the expression. Values of type char are displayed as characters as well as in decimal and hexadecimal format. If the character is unprintable, it is displayed as an escape sequence, for example, \n or \xAC. Values of type int and long are displayed in decimal and hexadecimal format, and floats and doubles are displayed in floating-point format.

To override the default format, you can specify printf format containing conversion operators after each expression. No error checking is done on the format parameter, so strange results are possible. The format may contain up to two conversion operators for short and long types but only one for all other types. Refer to the description of the printf function in the SAS/C Development System Library Reference for more information.

All members of structures, unions, and arrays are displayed using the default formats. You can display partial arrays by specifying the first and last elements to display For example, the following command displays array elements 3, 4, and 5:

```
   display a[3..5]
```

To display all elements in an array, use an asterisk (*).

The string parameter can be any standard C string in double quotes as described in Chapter 1, "Getting Started."

To display a null-terminated string, use the dzero command. To dump an area of memory, use the dump command.

EXAMPLES

```
    display i
```
displays the variable i.

```
    d a "a=%10.5e"
```
displays a using a printf style format specifier.

```
    d p->d[5] + f(3)
```
displays the value of an expression containing a function call.

```
    display "X is", x, "J is", j
```
displays the value of several arguments including strings.

```
    d x "X is %d", j "J is %d"
```
displays the value of several arguments using printf style format specifiers.

```
    d a[3]
```
displays the value of an element of an array.

```
    display a[3..5]
```
displays the value of elements 3 through 5 in an array named a.

```
    d a[*]
```

```
  displays the value of every element of an array.

    d myobject
  displays the contents of the object myobject.

    d myobjectptr
  displays the address value of the object pointed to by the
  object pointer myobjectptr.

    d *myobjectptr
  displays the C++ object pointed to by the object pointer
  myobjectptr.

    d myobject.mypublicmember
  displays the contents of the public member, mypublicmember,
  of the objectmyobject.

    d myobjectptr->mypublicmember
  displays the contents of the public member of the object
  pointed to by the object pointer.

SEE ALSO
 Call ,  DUmp ,  DZero
```

1.32 DUmp

```
dump - Dumps memory contents

SYNOPSIS
  du[mp] [ variable  |  address  |  range ] [$]
   [format [ itemsize ] | text]

DESCRIPTION
The dump command displays an area of memory.  If you do not specify
any parameters, the dump command uses the same parameters that you
specified in the previous dump command (if any), but it displays
memory beginning where the previous dump command ended.

The variable, address, range, array-slice or $ parameter specifies
the memory to be displayed.  If you use the l or L form of the
range, the range length is scaled by the itemsize.  The $ specifies
the memory immediately following the last memory dumped when the
format or text parameters are used. The $ is ignored if a variable,
address, or range parameter has already been specified making it
useful for aliases where the dump address may be omitted.

The format parameter controls the format of the dump and
its value can be any of the following:

     a[scii]         FFP[float]     IEEE[float]
     b[inary]        f[loat]        o[ctal]
     d[ecimal]         h[ex]        u[nsigned]

The default format is hex.
```

If you specify a format parameter, you also can specify an itemsize parameter to control the length of each item dumped. The itemsize parameter can have a value of 1, 2, 4, or 8. The itemsize parameter is concatenated onto the end of the format parameter (for example, b4 for a binary format of length 4). If you do not specify itemsize, CodeProbe uses a default. As shown in the following table, the allowable size and default values depend on the format specified.

Format	Default	Allowed Sizes
ascii	1	1
binary	1	1, 2, 4
decimal	4	1, 2, 4
FFPfloat	4	4
float	8	4, 8
hex	4	1, 2, 4
IEEEfloat	8	4, 8
octal	4	1, 2, 4
unsigned	4	1, 2, 4

The text parameter is used to specify that an ASCII representation of the memory dump should be displayed.

Aliases For The dump Command

To provide compatibility with previous releases, CodeProbe supports the following aliases for the dump command:

da
dumps a range of memory as ASCII characters

db
dumps a range of memory in both hexadecimal and ASCII format

dc
dumps a range of memory in decimal format

dd
dumps a range of memory as 8-byte floating-point numbers in IEEE format

df
dumps a range of memory as 4-byte floating-point numbers in IEEE format

dffp
dumps a range of memory as 4-byte floating-point numbers in FFP format

di or dl
dumps a range of memory in integer format using decimal representations of the integers

dw
dumps a range of memory in short integer (16-bit) format using hexadecimal representations of the integers

```
    dp
  dumps a range of memory in 4-byte hexadecimal format

    ds
  dumps a range of memory in short integer (16-bit) format,
  using decimal representations of the integers.

EXAMPLES
    dump var
  dumps memory contents of variable var.

    du var L 13 hex1
  dumps memory contents for 13 bytes using 1-byte hexadecimal
  format.

    du var1..var2 ascii
  dumps memory contents between var1..var2 in ASCII format.

    dump 0xAF8100 L 8
  dumps memory contents for 8 longs (32 bytes) starting at
  absolute address 0xAF8100.

SEE ALSO
 Display ,  DZero
```

1.33 DZero

```
dzero - Displays memory as a null-terminated ASCII string

SYNOPSIS
  dz[ero]  variable  |  address  |  array-slice

DESCRIPTION
The dzero command displays the contents of memory at the specified
location as a null-terminated (\0) string.  The location can be any
of the following:

    nonpointer variable
  displays the memory contents starting at the address of the
  variable and continuing until the null character is found.

    pointer variable
  displays bytes starting at that address until a null
  character is encountered.

    absolute address
  displays the contents of that address until a null character
  is encountered.

    array-slice
  displays all of the strings pointed to by the elements in
  the slice of the array of char * elements.

The dzero command wraps long strings on successive lines.  You can
```

```
use the opt strlen command to limit the number of characters printed
in long strings.  You can use the opt badchars command to control
printing of strings containing unprintable characters.

EXAMPLES
    dzero string
  displays characters until the null character is reached.

    dz ptr
  displays data pointed to by ptr as a string.

    dz ptr[3 .. 7]
  displays each element in the array slice ptr[3 .. 7] until
  the null character is reached.

SEE ALSO
 Display ,  DUmp
```

1.34 ECho

```
echo - Displays a string

SYNOPSIS
  ec[ho]  [text]

DESCRIPTION

The echo command writes the specified text in the Dialog window.  If
the text contains semicolons, the semicolons must be preceded with a
backslash (\;).  The echo command does not expand defines or
aliases.

This command is useful for documenting the actions being taken in a
debugger command file, the cprinit file, or AREXX macros.

EXAMPLES
    echo Starting program\; loading defines.
  displays the message following the echo command in the
  Display window.

SEE ALSO
 Display ,  EXecute
```

1.35 ENV

```
env - Sets the environment

SYNOPSIS
  env  [function|level]
  env  [-c[aller]|-s[ubroutine]|-u[p]|-d[own]|-integer|+integer]

DESCRIPTION
```

The env command changes the environment for subsequent debugger commands. The environment is the state of the machine at a specific point in the calling sequence (that is, the position in the call chain and the contents of variables). Setting the environment to a previous point in the calling sequence, such as the point where the current function was called, returns the machine to the state it was in at that time. However, only the information that can be retrieved from the stack or procedure save areas will have been restored. Scratch registers and other components, such as the values of externs, that are not saved and restored by function calls remain relative to the current execution point in the program and are not changed. Nonscratch registers are set to the values for the new environment.

The environment can be set to an absolute or relative position in the call chain. To set the environment to an absolute position, specify a function or level:

function
specifies the name of a function in the call chain. If the same function appears more than once in the call chain, the most deeply recursive one is selected.

level
moves the environment to the level specified. level must be an integer. Level 1 is defined to be the function in which you are currently stopped. The caller's level is 2, its caller is level 3, and so on. These level number designations change as the program steps into and returns from functions. You can display level numbers using the where command or Calls window.

The other forms to the env command set the environment relative to the current function:

-caller, -up, and -1
move the environment up one level to the caller of the current function

-subroutine, -down, and +1
move the environment down one level toward the bottom of the call chain

+integer
moves the environment down the call chain the number of levels you specify

-integer
moves the environment up the call chain the number of levels you specify.

If you attempt to move the environment up or down more levels than there are in the call chain, a warning is displayed and the environment is moved as far as possible.

Any command that causes the program to resume execution, such as the go or trace commands, automatically resets the environment to the

```
last point of execution.  Issuing the env command is the same as
double-clicking on a function call entry in the Calls window.

EXAMPLES
    env
  sets the user environment to the last point of execution.

    env main; d i
  sets the user environment to main and display i.

    env -subroutine
  sets the environment to the called function (down 1 level).

    env -caller
  sets the environment to the caller function (up 1 level).

    env +5
  moves the environment down 5 levels.

    env 7
  moves the environment to the level 7.

    env myclass::mymemberfunction
  sets the environment to the called member function
  myclass::mymemberfunction.


SEE ALSO
 WHEre
```

1.36 EXecute

```
execute - Executes a debugger command file

SYNOPSIS
    ex[ecute]  filename

DESCRIPTION
The execute command reads and executes CodeProbe commands from the
file specified by filename.  If the file cannot be found, CodeProbe
appends a .cpr extension to the filename and tries again.  Thus, you
can place a set of debugger commands in a file with a .cpr extension
and specify the root filename, minus the extension, in the execute
command.  CodeProbe searches for the specified file in your normal
Shell path as defined with the AmigaDOS path command:

For example, you may have a file, cmds.cpr, that contains the
following commands:

   break sort
   blist
   trace
   trace

The execute cmds.cpr command produces output similar to the
```

following:

```
   executing commands from cmds.cpr
   1  0xC32D3E sort:\sort.c\sort 22
   sort:\sort.c\init 5
   sort:\sort.c\init 9
```

The debugger commands themselves are not displayed unless echo mode is turned on.

If you are debugging a program frequently and want to set a number of breakpoints at the same places, you can place all of your breakpoint commands in a file and then use the execute command on this file. Using the execute command means that you do not have to re-enter the same commands each time you use the debugger.

If the execute command is used in a command list, it must be the last command in the list.

In the Dialog window, in line mode, and in execute files, lines containing only comments are ignored. In the Dialog window and in line mode, blank lines are treated as a proceed command.

EXAMPLES

execute setvars

executes the file named setvars or setvars.cpr.

execute test/cmds.cpr

executes the file named cmds.cpr located in the test directory.

1.37 EXPAND

expand - Expands and displays a command line

SYNOPSIS

expand [alias] command-line

DESCRIPTION

The expand command displays its arguments with all macros expanded. If you specify the alias option, this command also expands aliases. The command-line specifies any valid command-line entry that includes aliases or macros.

EXAMPLES

expand alias dp

expands the alias named dp to display the following:

```
        dump        $ hex4
```

expand ISEQUAL(1,5)

expands the macro ISEQUAL using the arguments 1 and 5. For example, if ISEQUAL has been defined as (a == b), then it would be expanded to (1 == 5).

```
SEE ALSO
 ALias ,  DEFine
```

1.38 FINish

```
finish - Terminates the kernel and the cross-debugger

SYNOPSIS
  fin[ish]

DESCRIPTION
In cross-debugging mode, the finish command terminates the session.
You should enter the finish command on the host machine.  The host
machine is the machine that displays the CodeProbe user interface.

The cross debugger (CPRX) on the host machine tells the kernel
(CPRK) on the target machine to terminate after cleaning up and
closing the communications link.  If you are not in cross-debugging
mode, the finish command does not work.

For a complete description of the cross debugger, see Chapter 8,
"Using the Cross Debugger."

EXAMPLE
    finish
  tells the kernel to terminate after disconnecting from the
  cross debugger.

SEE ALSO
 Quit
```

1.39 FRegister

```
fregister - Displays or modifies floating-point registers

SYNOPSIS
  fr[egister] [ register  [[=]  expression ]]

DESCRIPTION
On machines that have a math coprocessor chip, the fregister command
displays or modifies the contents of the floating-point registers.
If you do not specify any parameters, the fregister command displays
the current contents of all the machine's floating-point registers
in hexadecimal and as floating-point numbers. If you specify only a
register name, the contents of the register are displayed.  If you
also specify an expression, then that expression is saved in the
register.

You can also use the display and set commands to display and modify
the registers, and you can use the Register window to display the
```

```
registers.

EXAMPLES
    fr
  displays all floating-point registers and flags.

    fregister fp0 30.14
  sets floating-point register fp0 to 30.14.

    fr fp2 = sales
  sets floating-point register fp2 equal to the value of
  sales.

    fregister fp1
  displays the value of floating-point register fp1.

SEE ALSO
 Display ,  Register ,  RFlag ,  SEt
```

1.40 Go

```
go - Continues execution until a breakpoint is encountered or the
     program exits

SYNOPSIS
 g[o] [ location  [after(integer)] [when( expression )]]

DESCRIPTION
The go command begins execution of the program at the current
location, which is identified by the address stored in the program
counter (PC) register.  Execution continues until a breakpoint is
reached or until the program terminates, either normally or with an
error.

The location parameter specifies a location for a temporary
breakpoint.  If you specify a location, the temporary breakpoint
exists only for the duration of the go command.  The next time
program execution stops, the breakpoint is cleared, even if the
temporary breakpoint was not reached.

The go command supports the following options:

    after
  specifies the minimum number of times the specified location
  must be executed before the breakpoint is triggered.  The
  integer you specify with the after option is called the pass
  count.  Each time the breakpoint is hit, the pass count is
  tested.  If the pass count is greater than 1, the pass count
  is decremented by 1, and execution continues.  The
  breakpoint is triggered when the pass count equals 1.  For
  example, if you specify a pass count of 5, the breakpoint is
  triggered when the location is executed the 5th time. For go
  commands that do not contain an after clause, the pass count
  is set to 1.  The blist command shows the current pass count
  for each breakpoint.
```

```
    when
  specifies an expression that must evaluate to true (nonzero)
  before the breakpoint is triggered.

You can specify the after and when options in any order.  If you
specify both options, the pass count is tested and decremented only
if the when condition is true.

NOTE:  To avoid confusion, the go command does not use the default
       radix setting. You must specify 0x for an address; otherwise,
       you will be specifying a line number. For example, even if
       the radix is set to hex, the following command specifies a go
       command for line 20 and not address 0x20:

       g 20

EXAMPLES
    go
  continues program execution to the next breakpoint or the
  end of the program.

    go 14
  continues program execution to line 14 of the current
  module. Line numbers are relative to a module and not a
  function. This example assumes that code was generated at
  line 14. You cannot use the go command to go to a line with
  no code generated.

    go sort
  continues program execution to the first line of the sort
  function.

    go \myfile.c\sort 14
  continues program execution to line 14 of the sort function
  in module myfile.c. This example also assumes that code was
  generated at line 14.

    go mylib.library:myfunc
  continues program execution to the myfunc function in the
  shared C library mylib.library.

    go $ after(5)
  continues program execution until the fifth time the current
  line is executed.

    go 0x804A
  continues program execution until absolute address 0x804A is
  reached.

    go myclass::mymemberfunction
  continues program execution to the first line of the
  myclass::mymemberfunction function.

    go \myfile.cxx\myclass::mymemberfunction 14
  continues program execution to line 14 of the
  myclass::mymemberfunction function in module myfile.cxx.
```

```
  This example assumes that code was generated at line 14.

    go mycxxlib.library:myclass::mymemberfunction
  continues program execution to the myclass::mymemberfunction
  function in the shared C++ library mycxxlib.library.


SEE ALSO
 Break ,  Proceed ,  PS ,  Trace ,  TS
```

1.41 HELP

```
help -- Displays help information

SYNOPSIS
  h[elp]  [command-or-topic]
  ?  [command-or-topic]

DESCRIPTION
The help command displays information about commands and operands.
To display help about a specific command or topic, enter the command
or topic as a parameter to the help command.

In window mode, if you do not specify a command or topic, CodeProbe
opens the Help window and displays a list of commands and topics.
You can click on a command or topic to display more detailed
information.  In line mode, CodeProbe displays this same list of
commands and topics, and must enter help followed by the command or
topic to see additional information.

Each help screen contains a See Also section listing related
commands and topics.  To display information about one of these
items, click on the command or topic.  For example, the See Also
section for the dump command lists the display and dzero commands.
You can display the help screen for either of these commands by
clicking on the name of the command.

The Help window is an AmigaDOS window invoked on the AmigaGuide
database sc:help/cpr.guide.

EXAMPLES
    help
  displays a list of commands and topics.

    h break
  displays information about the break command.

    ? break
  displays information about the break command.
```

1.42 HUnks

```
hunks - Lists the addresses and sizes of all hunks

SYNOPSIS
  hu[nks]

DESCRIPTION
The hunks command displays the list of loaded hunks, their
addresses, and their sizes for the current executable file.

The hunks command displays information in the following format:

  Hunk  Address       Size
  ----  -------     ------------
   0  00C32AE8    0xB54 (2900)
   1  00C28570    0x27C (636)

The size is displayed first in hexadecimal format followed by
decimal format in parentheses.

EXAMPLE
    hunks
  lists all hunks for the current executable file.
```

1.43 Jump

```
jump - Changes the current execution point

SYNOPSIS
  j[ump]  location

DESCRIPTION
The jump command updates the program counter to point to a new
location.  If you use the jump command, the code between the
previous execution point and the new location is not executed.  The
jump command also allows you to re-execute code that you want to
inspect more closely.

For the location parameter, you can specify any value that would
also be valid for the break or go commands.

NOTE:  The jump command does not restore any program variables or
       memory locations.  If you use jump to re-execute a statement
       that increments a variable, the variable is incremented again
       when you step over that statement.

CAUTION!!
  Certain registers may not be set up the way the code that is
  jumped to expects.

  Using the jump command is more likely to be safe if you always
  jump from one C source file line to another in the same function
  (not from assembly lines), and you have compiled with a debugging
  option that flushes non-register variables being held in registers
  to memory at C source line boundaries (that is, symbolflush or
```

```
  fullflush).  If the jump command is used on optimized code, the
  registers are probably not set correctly.


NOTE:  To avoid confusion, the jump command does not use the default
       radix setting. You must specify 0x for an address; otherwise,
       you will be specifying a line number. For example, even if
       the radix is set to hex, the following command specifies a
       jump command for line 20 and not address 0x20:

         j 20

EXAMPLES
    jump 12
  jumps to line 12 in the current function.

    jump 0x804E
  jumps to the address 0x804E.


SEE ALSO
 ENV ,  Go ,  RETurn
```

1.44 List

```
list - Lists the lines in a source file

SYNOPSIS
   l[ist]
   l[ist]  $
   l[ist]  +integer|- integer  [L length]
   l[ist]  line-range
   l[ist]  [executable-name:]\module  [line-range]
   l[ist]  [[executable-name:]\module\]function  [line-range]

DESCRIPTION

In line mode, the list command lists the lines in a source file.

If you do not specify any arguments, the list command displays lines
starting with the current list line, and the number of lines that
are displayed is controlled by the opt list command.  The current
list line is the source line at which the program is stopped.

If you specify list $, the list command displays a number of lines
above and below the current list line, and the number of lines above
and below is controlled by the opt context command.

See the description of the location parameter in Chapter 1, "Getting
Started," for a description of the executable-name, module, and
function parameters.

NOTE:  Specifying the module is not supported with C++ programs.

If you specify a line-range, the line command displays the lines in
```

```
that range.  Specify the range using one of the following forms:

  starting-line[[..]ending-line]
  starting-line L number-of-lines

If you specify a module or function, CodeProbe opens the source file
for that environment, if possible, and displays the desired line
range.  If you do not specify a line range or function, the listing
begins at line 1.  If you specify a function but do not specify a
line range, CodeProbe lists lines beginning with the function
declaration.

Each time you enter the list command, the current list line is set
to the line immediately following the last line listed.  the current
list line is also updated each time the program is allowed to
execute.

EXAMPLES
    list
  lists the next group of lines. The number of lines listed is
  set by the opt list command.

    list $
  lists lines around the line pointed to by the current
  program counter.

    list module
  lists first lines of the specified module.

    list \module 20 40
  lists lines 20 through 40 of the specified module.

    list \module 20 l 10
  lists 10 lines of the specified module starting at line 20.

    list +20
  starts listing lines 20 lines down from the current line.

    list -20
  starts listing lines 20 lines up from the current line.

    list -20 L 10
  lists 10 lines, starting 20 lines up from the current line.

    list foo
  starts listing at the declaration of the function named foo.

    list foo 10 20
  lists lines 10 through 20 of the function named foo.

SEE ALSO
 OPt ,  SOurce ,  Unassemble
```

1.45 listsym

```
listsym - Lists all currently defined symbols

SYNOPSIS

    listsym  [symbol-name]

DESCRIPTION

The listsym command displays the list of all symbols and as much
information as possible about the symbol, such as current value,
prototype, the address of the symbol (for externs), and mangled name.

EXAMPLES
    listsym
  displays all of the currently defined symbols.

    listsym i
  lists the value of variable i.

    listsym myclass::memfunc
  displays the prototype, address, and mangled name for memfunc in
  class myclass.


SEE ALSO
 display
```

1.46 LOg

```
log - Logs debugger commands to a file

SYNOPSIS
   lo[g]  [log-command]

DESCRIPTION
The log command saves, in a log file, a record of all activity in
the Dialog window including commands and results.  The name of the
log file is set with the log file command.  The default filename is
cpr.log.

The log command does not save menu selections.

If you do not specify a log-command, the log command displays the
current log state (either on or off) and the filename of the log
file.  The log-command parameter can be any of the following:

    append
  appends log information to the end of the log file

    file <filename>
  specifies the filename of the log file and opens the file

    off
  closes the log file
```

```
    on
  opens a new log file

    snap
  snapshots the contents of the Dialog window to the log file.

If you enter a command that turns on the logging of your session,
such as log on, log append, or log snap, but logging is already
turned on, the log command itself is added to the current log file.
If you change the log filename while logging is enabled, CodeProbe
closes the current log file and opens a new one.  The log file is
automatically closed when you exit the debugger.

EXAMPLES
    log snap
  saves the dialog for the current session into the log file
  named cpr.log.

    log file foo.log
  sets the log file to the filename foo.log.

    log on
  enables the log file.

    log off
  disables the log file.

    log append
  enables the log file and appends new log information to the
  end of the log file.
```

1.47 modules

```
modules - Lists all of the modules in an executable

SYNOPSIS

  m[odules]  [ location ]

DESCRIPTION

The modules command produces a list of all of the modules in an
executable. If you do not specify a location, the debugger lists all
of the modules for the current executable.  If you specify a
location, CodeProbe lists all of the modules for the specified
executable.

This command is not supported for C++ programs.

NOTE:  This command accepts a location containing just an executable
       name, even though that is not legal for locations in general.

EXAMPLES
    mod
```

```
  lists all modules in the current executable.

    mod img:
  lists all modules in the executable img.

    mod img:\mod.c\fnc
  lists all modules in img.  The mod.c and fnc are ignored.
```

1.48 OPt

```
OPt -- Show option values or change the value of a debugger option

SYNOPSIS

  op[t]
  op[t]  au[toswap]  [on|off]
  op[t]  ar[rdim]  integer
  op[t]  b[adchar]  integer
  op[t]  ca[se]  [on|off]
  op[t]  cat[ch] [on|off]
  op[t]  co[ntext] integer
  op[t]  dev[ices]  [on|off]
  op[t]  e[cho]  [on|off]
  op[t]  ib[ytes]  [on|off]
  op[t]  ig[norepath]  [on|off]
  op[t]  l[ist]  integer
  op[t]  rad[ix]  [d[ecimal]|h[ex]]
  op[t]  ran[gelen]  integer
  op[t]  re[slib]  [on|off]
  op[t]  se[arch]  [+|-]  directory[,directory]...
  op[t]  so[urce]  c|a[sm]|m[ixed]|n[ext]
  op[t]  st[rlen]  integer
  op[t]  tab n
  op[t] t[ask] [ task-name  |  task-address ]
  op[t]  u[nassemble]  integer

DESCRIPTION
The opt command is used to display and change the current setting of
the debugger's options.  The opt command by itself shows the current
settings of all options, plus the current task.  If you also specify
an option and a setting, this command changes the debugger's options
as you specify.  The following list describes the options and their
possible values:

    arrdim
  specifies the maximum number of array elements that are
  displayed at one time with the display command. The default
  value is 20.  If an array has fewer than the specified
  number of elements, all of the array's elements are
  displayed.  For larger arrays, the specified number of
  elements are displayed, followed by an ellipsis (...).

    autoswap
  if set to on, pushes the program's screen to the front each
  time control is given to your program.  When a breakpoint is
```

reached, the debugger screen is again pushed to the front. If you are single stepping through source code, the switching of screens will probably appear as a brief flash. Autoswap mode is particularly useful when debugging programs that require input from the keyboard. The program's screen is automatically pushed to the front whenever input is required.

If set to off, the program's screen is not pushed to the front. The default setting is off.

badchar
controls the number of non-printable characters that a dzero or display command will accept when displaying a string referenced by a character pointer. The default value is 3. A value of zero (0) indicates no limit.

case
controls case sensitivity for the search command. If you specify opt case on, the search command performs case-sensitive string searches. The default setting is off.

catch
if set to on, catches any new task generated by a task that is already under debugger control. The default setting is off. NOTE: Tasks started by a call to OpenDevice are controlled by the opt devices command.

context
controls the number of lines (context lines) displayed above and below the current line. The default number is 2. In windowing mode, the maximum number of lines is limited by the number of lines in the Source window. In source mixed mode, it is not always possible to keep the correct number of lines above or below the current line. In line mode, this option controls the number of lines displayed with the list $ command.

devices
if set to on, the debugger attaches to, or catches, any new device process or task generated by a task under debugger control. When a device is opened for the first time, a new task may be created. The default setting is off.

echo
if set to on, tells the debugger to echo all commands to the Dialog window before executing them. This option is especially useful if you want to use the execute command to execute a file of debugger commands and see the commands and their output as they are executed. The debugger also displays commands that are invoked by selecting menu options or by double-clicking with the mouse. The default is off.

ibytes
if set to on, tells the debugger to display the instruction bytes being disassembled. This affects the output of the unassemble command, and in windowing mode, it also affects

the Source window in mixed and asm source modes. If this option is set to on, the second field of the disassembly contains a hexadecimal dump of the instruction. For example, if ibytes is set to on, the unassemble command may produce: p 0025F950 48E70130 MOVEM.L D7/A2-A3,-(A7)

If ibytes is set of off, the same command would produce:

```
   0025F950     MOVEM.L D7/A2-A3,-(A7)
```

The default setting is off.

For more information, see the description of the unassemble command.

ignorepath
if set to on, tells the debugger to ignore the pathname for the source file provided by the compiler. In this case, CodeProbe looks in the current directory for the source file. By default, ignorepath is set to off, and CodeProbe uses the entire pathname when searching for the source file. You can use this option and the opt search command to override the source filename specified in the object file. For more information, see Chapter 2, "Customizing the Debugging Environment."

list
controls the default number of lines displayed by the list command in line mode. The default number is 6.

radix
sets the default input type for constants. You can specify hexadecimal or decimal. The default setting is decimal. If you specify hexadecimal, then you do not have to type the 0x before hexadecimal constants.

rangelen
controls the default size of ranges when no range size information is available. The default value is 64. This value is used when you specify only an address parameter for the display, dump, or watch commands.

reslib
steps into resident libraries while tracing or running a program with watch breaks. The default setting is on. Leave this option on unless you want to debug your own resident libraries. However, tracing into system libraries such as Exec or Intuition can cause problems. Stepping into system libraries means that you are in ROM. You will not have C source code available, and you cannot set breakpoints in ROM. The debugger will not be able to figure out where you are. Also, you should disable all watch breaks before stepping into a ROM-resident library routine.

search
defines a set of directories that you want the debugger to search for the source code. For a complete description of

the opt search command, see Chapter 2, "Customizing the Debugging Environment."

source
controls how your source code is displayed in the Source window in windowing mode. You can choose one of three possible settings:

C displays C or C++ source lines when you enter a trace or proceed command or when a breakpoint is triggered. In C mode, you cannot single step by assembly instruction. The default setting is C.

asm displays disassembled code. The trace and proceed commands step by assembly instruction.

mixed displays both assembly instructions and C or C++ source lines.

Repeatedly choosing the next value cycles through the three possible modes.

strlen
controls the maximum number of bytes that are displayed when a character string is displayed with the display and dzero commands. The default number is 128. If the character string contains unprintable characters, fewer characters may be displayed.

tab n
sets tab stops every n columns in the Source window. The number n must be greater than zero. The default setting is 8.

task
displays the current task or, if you specify a task-name or task-address, changes the current task to the task you specify. The task-name and task-address are the task name and address of the task control block, as displayed by the tasks command.

If you specify the name, the name should be unique. If the name is not unique, the debugger uses the first task with that name that appears in its task list. If the name contains a blank, enclose the name in double quotes. The task names are case sensitive.

If you specify the task address, enter the address in hexadecimal notation (with a 0x prefix).

When you change to a new task, CodeProbe displays a new set of registers. The highlighting of changed registers in the Register window may be incorrect since the debugger keeps track of changes only while stepping through a single task. The module displayed in the Source window may change. If the task was in a system or linked library, assembler lines are displayed.

```
  You may be able to display the calling sequence of the task
  by using the where command, but since assembler routines,
  including the Amiga system's resident libraries, do not
  follow C language calling conventions, the information
  displayed by the where command may not be accurate.

  You can modify any registers, condition control register
  (CCR) flags, or stack variables, just as in the breakpointed
  task.  You can also single step through the code.

 CAUTION!!
   Incorrect use of breakpoints may crash your machine.

    A breakpoint is implemented by placing an illegal instruction at
    the desired location.  All tasks under the debugger's control
    have a trap handler that reports back to the debugger when the
    illegal instruction is executed.  If you must place a breakpoint
    in shared code, such as a resident library under test, be sure
    that any task that might open the library is under debugger
    control.  Never place breakpoints in libraries that you do not
    control.


    unassemble
  controls the default number of instructions that are
  disassembled by the unassemble command in line mode when the
  source file is not available. When the source file is
  available, the unassemble command displays the disassembly
  for a single source line.  This option has no effect in C
  mode or windowing mode.  The default setting is 4.


EXAMPLES
    opt
  displays the settings of all options, plus the current task.
   You cannot change the current task with the opt command,
   but you can use the jump command to change the execution
   point.

    opt unassemble 10
  sets unassemble count to 10.

    opt search /src,/test
  sets search directories.

    opt search +test2
  appends a new search directory.

    opt task "Child"
  changes the current task to the task named Child.

    opt task 0x7D9F884
  changes the current task to the task with the address
  0x7D9F884.

    opt task "Child of multi"
```

```
  changes the current task to the task named "Child of multi".


SEE ALSO
 Display ,  DZero ,  jump ,  List ,  Unassemble
```

1.49 Proceed

```
proceed - Single-steps over function calls

SYNOPSIS
  p[roceed]  [integer]

DESCRIPTION
In C mode, the proceed command steps over the number of source
statements specified by the integer parameter.  In Mixed or Asm
modes, it steps over integer number of machine instructions.  If
CodeProbe encounters a function call, the function is executed as if
it were a single statement.  If you do not specify an integer, it
defaults to 1.

When you are debugging a C++ program in C source mode, the proceed
command steps over the translated C statements.  In other words, if
one C++ statement was translated into more than two C statements, it
will require two proceed commands to step over the C++ statement.

If you press Enter at the CodeProbe prompt without typing any
commands on the command line, CodeProbe executes a proceed 1
command.  You can also press the F6 key to enter the proceed
command.

EXAMPLES
    proceed
  steps 1 source statement in C mode or 1 machine instruction
  if in Mixed or Asm modes.

    proceed 5
  steps 5 source statements in C mode or 5 machine
  instructions if in Mixed or Asm modes.

SEE ALSO
 Go ,  PS ,  Trace ,  TS
```

1.50 PS

```
ps - Single-steps over function calls by source line

SYNOPSIS

   ps  [integer]

DESCRIPTION
```

```
The ps command steps over the number of C source statements
specified by the integer, even if mode is set to Mixed or Asm.  If
CodeProbe encounters a function call, the function is executed as if
it were a single statement.  If you do not specify an integer, it
defaults to 1.

When you are debugging a C++ program in C source mode, the ps
command steps over the translated C statements.  In other words, if
one C++ statement was translated into more than two C statements, it
will require two ps commands to step over the C++ statement.

EXAMPLE
    ps 2
  steps through two source lines regardless of the debugger
  mode.

SEE ALSO
 Go ,  Proceed ,  Trace ,  TS
```

1.51 Quit

```
quit - Terminates the debugger

SYNOPSIS

  q[uit]  [-abort]

DESCRIPTION
The quit command terminates the debugging session and returns to the
operating system prompt.

In cross-debugging mode, the quit command terminates only the cross
debugger (CPRX) on the host machine, leaving the kernel (CPRK)
waiting for a new debug session.

Normally, CodeProbe forces the program being debugged to call the
exit function before the debugger terminates the program.  The
debugger then exits.  If you invoke the debugger with the -startup
option, the debugger exits without forcing the program to call exit.
If you start CodeProbe with the -startup option, terminate CodeProbe
on the target machine with the finish command instead of entering
quit.

If you specify the -abort option, the debugger will not call exit
before terminating.  If you think that your destructors,
autotermination functions, or atexit functions may cause system
problems, then specify the -abort option.  For example, if you
determine during your debugging session that the memory heap managed
by malloc is corrupt, then you do not want to free that memory as
CodeProbe exits.  Normally, this memory is freed by an
autotermination function called from exit.

If you specify -abort, then cleanup for your program is not
performed.  Files opened with fopen or open are not closed, memory
allocated with malloc is not freed, and destructors are not
```

```
executed.

EXAMPLE
    quit
  terminates the debugging session.  The program calls exit
  (unless you started the debugger with the -startup option).

    quit -abort
  terminates the debugging session.  The program does not call
  exit.
```

1.52 Register

```
register - Displays or modifies registers

SYNOPSIS
  r[egister] [ register  [[=]  expression ]]

DESCRIPTION

If you do not specify any arguments, the register command displays
the current contents of all the machine registers except
floating-point registers.  You can display integer, status, and flag
registers, stack pointers, and instruction counters with the
register command.  If you specify a register, CodeProbe displays the
contents of that register.  If you also specify an expression,
CodeProbe stores the value of that expression in the register.

regs is a synonym for register.

You can also use the display and set commands to display and modify
the contents of registers.  (You can use the fregister command to
display floating-point registers.  You can use the rflag command to
modify flag registers.)

EXAMPLES
    register
  displays all registers and flags.

    register d5 = 30
  sets register D5 to 30.

    r d3 = index
  sets register D3 to the value of index.

    register a0 &x
  sets register A0 to the value pointed to by x.

    r a1
  displays the value stored in register A1.

SEE ALSO
 Display ,  FRegister ,  RFlag ,  SEt
```

1.53 REStart

```
restart - Restarts the program being debugged

SYNOPSIS
   res[tart]  [argument-list]

DESCRIPTION

EXAMPLES
  restart /* restarts the program using the same arguments
       with which it was initially invoked with */
  restart myfile.txt 5    /* restarts the program passing it two
            arguments: myfile.txt and 5 */
  restart "sample string "    /* restarts the program passing in a string
        as an argument */

SEE ALSO
 STArt
```

1.54 RETurn

```
return - Returns immediately from the current function

SYNOPSIS

    ret[urn]  [ expression ]

DESCRIPTION
The return command causes the current function to return to its
caller without executing the rest of the function.

NOTE: This command is not supported for C++ programs.

The value of the expression, if specified, is used as the return
value from the function. The value is converted to the appropriate
type if necessary.  However, it must be a scalar quantity (not a
structure or union).

If register is the machine register that holds the return value, the
return command is equivalent to the following:

   set register=expression;jump return;proceed

When you enter the return command, the return value, if any, is
returned to the calling function, and execution is suspended as
though a breakpoint were triggered in the calling function after the
call was made.  For example, you may have the following code:

   i = 5;
   j = func(i);
   if (j < 2)
       j++;
```

```
If a return 7 command is issued during the execution of func,
execution stops inside the line that did the call, just before the
assignment to j.  You can then step to the next statement.

An attempt to return a value from a function declared void is
treated as an error, and an error message is displayed.  Similarly,
an error message is displayed if you use a simple return command
(without an expression) from within a function declared as having a
return value.  In all other cases, the return value is cast (if
necessary) to the correct type and returned to the calling function.

   CAUTION!!
     Using the return command while stopped in the prolog of a
     function causes unpredictable results.

     A function prolog is code generated by the compiler at the
     start of each function. This code sets up the call frame and
     any automatic variables.  If you have specified opt prolog off
     or if you step into a function in assembler mode, it is
     possible to stop inside the function prolog.


   CAUTION!!
     Do not use return in optimized code.

     The optimizer may have modified the data on the stack, and the
     debugger will not be able to find the return address.

EXAMPLES
    return stringptr
  returns the value of stringptr.

    ret 5
  returns an integer value of 5.

    return
  returns from a void function.
```

1.55 RFlag

```
rflag - Displays or modifies flags

SYNOPSIS

   rf[lag]  [flag-setting...]

DESCRIPTION
The rflag command can be used to display or modify the settings of
the machine flags.  If no parameters are specified, the current
settings of the flags are displayed.  With one or more arguments,
the indicated flags are set or cleared according to the values
given.

A flag setting can be one of the values in the following table. The
Set column contains the values used to set flags, while the Clear
```

```
column contains the values used to clear flags.

    Flag Name         Set      Clear
    ----------------------- --------- ---------
    Overflow (yes/no)              OV              NV
    Sign (negative/positive)       NG              PL
    Zero (yes/no)                  ZR              NZ
    Auxiliary Carry (yes/no)       AC              NA
    Carry (yes/no)                 CY              NC

EXAMPLES
    rflag
  displays all registers and flags.

    rflag cy
  sets the carry flag to 1 (yes).

SEE ALSO
 FRegister ,  Register
```

1.56 SEArch

```
search - Searches for a string in the current source file

SYNOPSIS
  sea[rch] [ string ]

DESCRIPTION
The search command searches the current source file for the
specified string.

In windowing mode, the search begins from the top line of the
current Source window.  If the debugger finds a match, CodeProbe
repositions the Source window so that the line containing the match
is the number of lines specified by the opt context command from the
top of the window. The cursor is positioned at the beginning of the
matching string.

In line mode, the search begins at the first line that would be
displayed if you entered a list command with no arguments.
CodeProbe displays the line containing the match.

If you do not specify any arguments, the debugger searches for the
string specified in the last search command, starting from the new
current line.  If it does not find a match, CodeProbe prints a
message.  If you enter search again, the search resumes beginning at
the top of the file.

You can use the opt case command to specify whether the search is
case sensitive.

EXAMPLES
    search foo
  finds the string "foo".
```

```
    search foo;
  finds the string "foo". The semicolon used in this command
  is a command delimiter and is not considered part of the
  search string.

    search foo\;
  finds the string "foo;". The escape character (\) tells the
  debugger that the semicolon is part of the search string.

    search
  repeats the last search.

SEE ALSO
 OPt ,  List
```

1.57 SEt

```
set - Modifies the values of variables or memory locations

SYNOPSIS

  se[t]  variable
  se[t]  register-name  [=]  expression
  se[t]  address  [=]  string

DESCRIPTION
The set command modifies the values of variables, registers, or
memory locations in the program being debugged.

NOTE:  This command is not supported for C++ objects.

If you specify a variable or register name and an expression, this
command stores the result of expression in the location you specify.
In this command, variable can be a reference to an array element or
a structure member or an indirect reference. The variable cannot
refer to an aggregate such as a structure, union, or array.
Specifying a register name and expression is equivalent to:

   register register-name=expression

If necessary, the value for variable will be converted to the proper
type in accordance with the normal C conversion rules; it is then
entered into memory at the address of the first variable operand.
This is similar to a C assignment statement for a scalar variable.

The register can be any valid register name, D0 through D7 and A0
through A7, and if a co-processor is present, FP0 through FP7. You
can use the $ prefix to distinguish the register from a variable of
the same name.

You can also use the set command to store a string into memory.  The
null terminator is not copied unless it is explicitly present in the
string.  The equal sign is optional.  You can also use the built-in
strcpy and memcpy functions to store a string in memory.
```

```
You should the built-in strcpy and memcpy functions instead of using
set address=string.  However, if you specify an address and a
string, the set command stores the string at that address in memory.
set does not copy the null terminator unless it is explicitly
present in the string.

EXAMPLES
    set i = 5
  sets the variable i equal to the integer value 5.

    set a = 3.1415
  sets the variable a equal to the floating-point value
  3.1415.

    set $d0 = 300
  sets register D0 equal to 300. The $ is optional.

    set time->date = newtime
  sets the date member of the time structure equal to the
  value of the variable newtime.

    set stringptr "this string has no null byte"
  stores a string that is not null-terminated to the memory
  location pointed to by stringptr.

    set stringptr "this string has a null byte\0"
  stores a null-terminated string to the memory location
  pointed to by stringptr.

    set stringptr "string one\0string two\0"
  stores two null-terminated strings to the memory location
  pointed to by stringptr.

SEE ALSO
 FRegister ,  MEMCPY ,  STRCPY ,  Register
```

1.58 SLEep

```
sleep - Pauses for the time specified

SYNOPSIS
  sle[ep]  number

DESCRIPTION
The sleep command pauses for the number of seconds specified by the
 number  parameter.

EXAMPLE
    sleep 10
  pauses for 10 seconds.
```

1.59 show

```
show - Sets Source window to display specified source

SYNOPSIS

  sh[ow]  \module[\func]  [line]
  sh[ow]  func  [line]
  sh[ow]  line

DESCRIPTION
The show command tells the debugger to display the specified source
in the Source window.  The current environment does not change.
Specify the module, function, or line in the same format as that
displayed by the env command.

To return the Source window to the current environment, enter the
env command without any parameters.

EXAMPLE
    show \sort.c\printArr 42
  moves the Source window to line 42 of the printArr function
  in the module sort.c.
```

1.60 SOurce

```
source - Displays the source file

SYNOPSIS

   so[urce]  [filename]

DESCRIPTION
The source command overrides the C source file associated with the
current module.  The specified value for filename replaces the
filename associated with the module for the rest of the current
debugging session.  CodeProbe displays the new file immediately and
echos its name in the Dialog window.

If you do not specify a filename, the name of the current source
file is echoed in the Dialog window but the current source file is
not changed. The file is reread from disk if it has changed.

If the filename to which you want to change is the same as the
default filename but the path is different, you can use the \nl opt
search command instead of the source command to change the locations
where CodeProbe looks for source files.  The opt search command
affects all source files, not just the current one.

EXAMPLES
    source
  displays the current filename.

    source test/test1.c
  changes the filename for the current module to test1.c in
  the test directory.
```

```
SEE ALSO
 OPt
```

1.61 STArt

```
start - Restarts the program being debugged

SYNOPSIS

   sta[rt]  [argument-list]

DESCRIPTION
The start command reloads and executes your program up to the entry
point of main, just as when the debugger was first invoked.  If you
do not specify any arguments, start restarts the program with the
same arguments that you specified when the debugger was first
invoked.  (If you started CodeProbe with the -startup option,
execution stops before the startup code is executed and not at the
first line of the main function.) Because the program is reloaded,
all static data is reinitialized.  All breakpoints are retained, and
watch breaks on static and external variables are retained and
disabled if:

    >  You linked with a startup module other than cres.o or
       catchres.o.

    >  The breakpoint or watch break is set on an item in the near
       data section.

If you started CodeProbe with the -command option, do not use the
start command.  Do not use the start command as one of a sequence of
commands separated by semicolons.

If you specify arguments to the program, separate each argument with
white space, just as the arguments would appear on the command line
when invoking the program.  The program uses these arguments as if
they were specified on the command line.

   CAUTION!!
     The start command may cause your machine to crash if used
     improperly.

     When you enter the start command, your program is immediately
     reloaded and re-executed with no cleanup other than calling the
     exit function to close files.  The machine may be left in a
     state that can cause problems later.  To avoid this problem,
     allow your program to run to completion, if possible, and do
     any necessary cleanup before entering the start command.


The start and restart commands are the same if the same arguments
are provided as the argument-list.  The significant difference
between the two commands is their behavior when they are specified
with no arguments.  The start command without an argument-list
```

```
restarts the program without any arguments; however, the restart
command without an argument-list restarts the program with the
original arguments used when the debugger was invoked.

EXAMPLES
    start
  restarts the program being debugged without passing it any
  arguments.

    start myfile.txt 5
  restarts the program being debugged passing in the arguments
  myfile.text and 5.

SEE ALSO
 REStart
```

1.62 SYMBol

```
symbol - Finds the symbol nearest to the specified address

SYNOPSIS
  symb[ol]  address

DESCRIPTION
The symbol command searches the symbol information and displays the
name of the symbol whose location is closest to the address
specified by the address parameter.

EXAMPLE
    symbol 0x7D9F884
  displays the symbol whose address is nearest to 0x7D9F884.

SEE ALSO
 SYMload
```

1.63 SYMload

```
symload - Loads debugging information for an executable file

SYNOPSIS
  sy[mload]
  sy[mload]  "executable-filename"
  sy[mload]  pc  address  ["executable-filename"]
  sy[mload]  proc  "proc-name"|proc-address  ["executable-filename"]
  sy[mload]  seg  address  "executable-filename"

DESCRIPTION

The symload command associates an executable file and any debugging
information found in that file with a load module.  The symload
command is useful when you are debugging several interacting
programs that are invoked from separate executable files.  This
```

command does not not load any code into memory or spawn any new processes or tasks.

If you do not specify any options and you are running SegTracker, symload asks SegTracker for the seglist and filename based on the current PC. symload then loads the debugging information from the file if it is avaliable.

If you specify an executable-file without any other options and you are debugging a process, the symload command gets the seglist from the current process CLI if the process was started from the Shell or from the current Process structure if the process was started from the Workbench. symload then loads the debugging information from the filename in the seglist. If the PC is not in the seglist, symload prints a message saying that the debugging information was loaded properly, but the program is probably in an operating system call. (This option is useful if you want to catch rpograms that are in a Wait system call.)

The symload command supports the following options:

pc
asks SegTracker for the seglist and filename based on the address that you specify. If you also specify an executable-filename, then symload loads the debug information associated with that file instead of looking for the file associated with the process.

proc
uses the process name or address to find the CLI structure (if the program was invoked from the Shell) or the current Process structure (if the program was started from the Workbench). symload gets the seglist and command name from the CLI or Process structure. If you specify a process name, enclose the name in double quotes (" "). If the process name is not unique, symload uses the first process with that name in its list. If you also specify an executable-filename, then symload loads the debug information associated with that file instead of looking for the file associated with the process.

seg
treats the hexadecimal address you specify as a pointer to a BCPL segment list and maps the list to the executable file you specify.

EXAMPLES

```
symload "myapp"
```
reads the current segment list and symbols from the file named myapp.

```
sym proc "myapp"
```
reads the segment list and executable file for process myapp.

```
symload proc 0x7D9F884 "myapp"
```

```
  reads the segment list and executable file for the process
  located at 0x7D9F884.

    symload seg 0x7D9F884 "myapp"
  uses the segment list at address 0x7D9F884.

SEE ALSO
 HUnks
```

1.64 TAsks

```
tasks - Displays a list of system tasks

SYNOPSIS
  ta[sks] [a[ll]]

DESCRIPTION
The tasks command displays information about tasks.  By default,
tasks displays the tasks currently under debugger control.  If you
specify all, this commands displays the list of all system tasks.

The opt task command displays information in the following format:

   Address  Type  Pri  State   SigWait   StackPtr  Debug  Name
   00C513B8  13    0  Waiting  80000000  00C551CC   act   multi

The fields contain the following information:

     Address   contains the address of the task control block.

     Type      indicates the type of node that begins the task
         control block.  The number 13 is used for processes,
         and 1 is used for simple tasks. For a complete list
         of the possible task types, see the header file
         include:exec/nodes.h.

     Pri       contains the priority of the task.

     State     indicates the process state: Waiting or Ready.

     SigWait   displays the SigWait field of the task control block.
         This field indicates which signal bits the task may
         be waiting on.

     StackPtr  is the current position of the stack pointer.  This
         value will be different from the value displayed in
         the SP register.  The value displayed by tasks
         includes any information placed on the stack by the
         trap handler or exception handler associated with
         this task.

     Debug     indicates the task's status with the debugger:  act
         denotes an active task, and inact denotes an inactive
         task.  This field is blank for tasks that are not
         under the control of the debugger.
```

```
     Name      is the name of the task as specified in the task's
         node structure.

For more information on tasks or on any of these fields (except
Debug), refer to the Amiga ROM Kernel Reference Manual:  Libraries
and Devices.

EXAMPLES
    tasks
  displays a listing of all tasks under debugger control.

    ta all
  displays a listing of all tasks in the system.

SEE ALSO
 Activate ,  CAtch ,  DEActivate ,  DETach ,  OPt
```

1.65 Trace

```
trace - Single-steps into function calls

SYNOPSIS

    t[race]  [integer]

DESCRIPTION

In C mode, the trace command steps the number of source statements
specified by the integer parameter. In Mixed or Asm modes, it steps
the number of machine instructions specified by the integer
parameter. If CodeProbe encounters a function call, stepping
continues into that function. If you do not specify an integer,
trace steps one line.

You can also enter the trace command by pressing the F7 key.

When you are debugging a C++ program in C source mode, the trace
command steps over the translated C statements.  In other words, if
one C++ statement was translated into more than two C statements, it
will require two trace commands to step over the C++ statement.

EXAMPLES
    trace
  steps one source statement if in C mode, or one machine
  instruction if in Mixed or Asm mode.

    t 5
  steps five source statements or machine instructions.

SEE ALSO
 Go ,  Proceed ,  PS ,  TS
```

1.66 TS

```
ts - Single-steps by source line into function calls

SYNOPSIS

   ts  [integer]

DESCRIPTION
The ts command is similar to the trace command, except that ts steps
only by C source line, even if the source mode is set to Asm or
Mixed.  The integer parameter specifies the number of C source lines
to step. If you do not specify an integer, ts steps one line at a
time.

When you are debugging a C++ program in C source mode, the ts
command steps over the translated C statements.  In other words, if
one C++ statement was translated into more than two C statements, it
will require two ts commands to step over the C++ statement.


EXAMPLES
    ts
  steps one source line regardless of the mode the debugger is in.

    ts 2
  steps two source lines.

SEE ALSO
 Proceed ,  PS ,  Trace
```

1.67 UNAlias

```
unalias - Deletes an alias

SYNOPSIS

   una[lias]  name
   una[lias]  *

DESCRIPTION
The unalias command removes entries from the debugger's list of aliases.
To remove a specific alias, specify the alias name.  To remove all
aliases, specify an asterisk (*).

EXAMPLES
    unalias foo
  deletes the alias for foo.

    una *
  deletes all aliases.


SEE ALSO
```

ALias , DEFine , UNDefine

1.68 Unassemble

```
unassemble - Displays memory as assembler instructions

SYNOPSIS
  u[nassemble] [ start-location  [ end-location ]]

DESCRIPTION

The unassemble command displays the range of memory from
start-location to end-location in assembler format.

If you do not specify a location, unassemble disassembles the memory
at the address in the current program counter.  If you specify only
a start-address, unassemble displays the memory at that address.

If you enter additional unassemble commands before giving the
program control with a trace, proceed, or go command, unassemble
continues disassembling from the location at which the previous
unassemble command stopped.

If the debugging information contains source line numbers for the
memory being displayed, unassemble displays the C source line and
the assembler instructions generated for that line.  If the
debugging information does not contain source line numbers for the
memory being displayed, unassemble displays the number of
instructions as specified by the opt unassemble command.  By
default, unassemble displays 4 instructions.

The output of the unassemble command is equivalent in format to
output displayed in the Source window if you are in asm mode.  The
output consists of three or four fields depending on the setting of
the instruction bytes option, which is controlled by the opt ibytes
command.  The following is an example of unassemble output with
instruction bytes on and source unavailable:

   0x25F950 48E70130        MOVEM.L  D7/A2-A3,-(A7)
   0x25F954 BFEC0004        CMPA.L   0004(A4),A7
   0x25F958 65001BD6        BCS  00261530

The first field contains the hexadecimal address of the instruction
being disassembled.  The second field is a hexadecimal dump of the
actual bytes composing the instruction.  The third field contains
the M680x0 mnemonic for the given opcode.  The fourth field contains
the operands to the instruction.  If the ibytes option is turned
off, the second field is not displayed.

EXAMPLES
    unassemble \mod1\func1 10
  unassembles line 10 of func1 in mod1.

    unassemble 13 14
  unassembles lines 13 to 14.
```

```
    unassemble func1
  unassembles the first line of func1.

    unassemble func1 func2
  unassembles from &func1 to &func2.

    unassemble
  continues unassembling at the last location.

SEE ALSO
 OPt ,  List
```

1.69 UNDefine

```
undefine - Deletes a macro definition

SYNOPSIS

  [#]und[efine]  name
  [#]und[efine]  *

DESCRIPTION
The undefine command removes entries from the debugger's list of
macro definitions. To remove a specific macro, specify the macro
name. To remove all defines, specify an asterisk (*).

The # sign is optional and allows the debugger to read C header
files (.h files) using the execute command.

EXAMPLES
    #undef foo
  deletes the macro named foo.

    undefine *
  deletes all macros.

SEE ALSO
 ALias ,  DEFine ,  EXecute ,  UNAlias
```

1.70 Watch

```
watch - Sets a watch on a variable or memory

SYNOPSIS
  w[atch]  expression  |  range  [s[tatic] | d[ynamic]]

DESCRIPTION
The watch command sets a watch for the expression or memory range
specified.  A watch allows you to monitor a variable, an address, or
a range of memory.  Whenever control returns to the user, for
example by stepping or stopping at a breakpoint, the new value of
```

```
the watched object is displayed in the Watch window.  The Watch
window can be opened with the View menu, the window command, or by
pressing the F1 key.

In line mode, the object is not automatically displayed.  You must
use the wlist command to display it.

NOTE: This command is not supported with C++ objects.

The watch command does not stop program execution when the value
being watched changes; it just displays the changed value.  Use the
wbreak command if you want program execution to stop when the
watched value is changed.

By default, watches are dynamic.  The expression being watched is
re-evaluated whenever control returns to the debugger.  For example,
watch tmp[i] causes the watch to move every time i changes.

The static option evaluates the expression only once, when the watch
is set. The static option treats the result as an address, and
displays the contents of t

EXAMPLES
    watch text->len
  sets a watch for the symbolic scalar text->len.

    w \mod\f1\a[0] L 20
  sets a watch for a length of 20 bytes starting at the
  symbolic range specified by \mod\f1\a[0].

    w &a[0] .. &a[5]
  sets a watch for the range from &a[0]to &a[5].

    watch tmp[i] static
  if i = 3, watches tmp[3] regardless of changes to i.

SEE ALSO
 WBreak ,  WClear ,  WDisable ,  WEnable ,  WList
```

1.71 WBreak

```
wbreak -- Sets a watch break

SYNOPSIS
  wb[reak]  expression  |  range   [s[tatic] | d[ynamic]]

DESCRIPTION
The wbreak command sets a watch break for the expression or memory
range specified.  Whenever any byte in a specified range is
modified, control is returned to you.

NOTE: This command is not supported with C++ objects.

By default, watch breaks are static. The expression is evaluated
once when the watch break is set. The debugger treats the result as
```

```
an address, and displays the contents of the memory at that address.

The dynamic option re-evaluates the expression or range whenever
control returns to the debugger.  For example, wbreak tmp[i] causes
the watch break to break every time i changes.

EXAMPLES
    wbreak text->len
  sets a watch break for the symbolic scalar text->len.

    wb \mod\f1\a[0] L 20
  sets a watch break for a length of 20 bytes starting at the
  symbolic range specified by \mod\f1\a\lbk 0].

    wbreak &a[0]..&a[5]
  sets a watch break for the range from &a[0]to &a[5].

    wb p->q dynamic
  sets a watch break for different locations depending on the
  value of p.

SEE ALSO
 Watch ,  WClear ,  WDisable ,  WEnable ,  WList
```

1.72 WClear

```
WClear -- Clears one or more watches

SYNOPSIS

   wc[lear]  integer [integer ...]
   wc[lear]  *|l[ast]
   wc[lear]  integer..integer

DESCRIPTION
wclear command clears (deletes) one or more watches or watch breaks.
When a watch or watch break is cleared, it ceases to exist and can
be reinstated only by entering the watch or wbreak command again.
To disable watches or watch breaks temporarily, use the wdisable
command.

The integer parameter specifies the watch or watch break number as
displayed by the wlist command.  You can specify as many watch or
watch break numbers as needed.  An asterisk (*) clears all watches
and watch breaks, and last clears only the most recently set watch
or watch break.  You can specify a range of watches or watch breaks
with integer..integer.

EXAMPLES
    wclear 2 5 6
  clears watches or watch breaks numbered 2, 5, and 6.

    wclear last
  clears the last watch or watch break set.
```

```
    wc *
  clears all watches or watch breaks.

    wc4..7
  clears watches 4, 5, 6, and 7.

SEE ALSO
 Watch ,  WBreak ,  WDisable ,  WList
```

1.73 WDisable

```
wdisable - Disables (turns off) one or more watches

SYNOPSIS

  wd[isable]  integer  [integer...]
  wd[isable]  *|l[ast]
  wd[isable]  integer..integer

DESCRIPTION
The wdisable command disables (turns off) one or more watches or
watch breaks.  When a watch or watch break is disabled, it is not
recognized by CodeProbe, but it remains on the list of current
watches or watch breaks.

The integer parameter specifies the watch or watch break number as
displayed by the wlist command.  You can specify as many watch or
watch break numbers as needed.  An asterisk (*) disables all watches
and watch breaks and last disables only the most recently set watch
or watch break.  You can specify a range of watches or watch breaks
with integer..integer.

To re-enable a disabled watch or watch break, use the wenable
command.

EXAMPLES
    wdisable 2 5 6
  disables watches or watch breaks numbered 2, 5, and 6.

    wdisable last
  disables the last watch or watch break set.

    wdisable *
  disables all watches or watch breaks.

    wdisable 4..7
  disables watches 4, 5, 6, and 7.

SEE ALSO
 Watch ,  WBreak ,  WClear ,  WEnable ,  WList
```

1.74 WEnable

```
wenable - Enables (turns on) one or more watches

SYNOPSIS

  we[nable]  integer...
  we[nable]  [*|l[ast]
  we[nable]  integer..integer

DESCRIPTION
The wenable command enables (turns on) one or more watches or watch
breaks that have been disabled by the wdisable command.

The integer parameter specifies the watch or watch break number as
displayed by the wlist command.  You can specify as many watch or
watch break numbers as needed.  An asterisk (*) enables all watches
and watch breaks and last enables only the most recently set watch
or watch break.  You can specify a range of watches or watch breaks
with integer..integer.

EXAMPLES
    we 2 5 6
  enables watches or watch breaks numbered 2, 5, and 6.

    we last
  enables the last watch or watch break set.

    wenable *
  enables all watches or watch breaks.

    we 4..7
  enables watches 4, 5, 6, and 7.

SEE ALSO
 Watch ,  WBreak ,  WClear ,  WDisable ,  WList
```

1.75 WHAtis

```
whatis - Determines the type of an object

SYNOPSIS
  wha[tis]  expression
  wha[tis] ( type )

DESCRIPTION
The whatis command displays the type, location, and storage class of
an object or gives additional information about a data type.  If you
specify an expression, whatis displays the C data type of the
expression.  If the result of the expression is a simple variable or
a member of an array, whatis displays the address of the object and
identifies it as either static, extern, or automatic.  If the result
of the expression is a function, whatis displays the address of the
function, its return type, and its location.

If you specify a type, whatis displays the full definition for the
```

```
type.  For struct, union, and enum types, the debugger displays all
members of the aggregate. For a type that has been defined by a
typedef statement, the debuggger displays the base type of the
typedef.  You can also specify basic C data types such as int or
long as the type.

This command is not supported for C++ names.  Use the listsym
command to display the mangled C name, and specify the C name as the
parameter to whatis.

EXAMPLES

    whatis i
  determines the type of a variable named i.

    wha main
  determines the type, address, and defining module for the
  main function.

    wha 3.5
  determines the type of a constant.

    whatis Red
  determines the type of an enumeration constant.

    wha *p->c[3]
  determines the type of the expression *p->c[3].

    whatis (int) (1*3.4)
  determines the type of an expression with a type cast.

    whatis (node)
  determines the type of a typedef.

    whatis (struct X)
  displays the members of struct X.
```

1.76 WHEre

```
where - Shows the calling sequence

SYNOPSIS

  whe[re]  [a[rgs]]  [integer]

DESCRIPTION
The where command displays a list of the function calls in the call
sequence.  Calls are listed in reverse order beginning with the most
deeply nested function.  For example, if main calls firstfunc at
line 92, firstfunc calls nextone, and nextone calls innermost at
line 68, the output from the where command would look like this:

   1  In routine fpreg.o:\fpreg.c\innermost 48
   2* Called from fpreg.o:\fpreg.c\nextone 68 (+0xE)
   3  Called from fpreg.o:\fpreg.c\firstfunc 82
```

```
   4  Called from fpreg.o:\fpreg.c\main 92
```

The numbers on the left indicate a level number that can be passed to the env command. Level 1 is the run environment or the function in which you are currently stopped. The caller's level is 2, its caller is 3, and so on. These level number designations change over time as the program steps into and returns from functions. The asterisk indicates the current user environment as set by the env command.

The args option displays the arguments to each function like the args command.

The integer parameter specifies the number of calls to be displayed. By default, only the 20 most recent calls are printed.

You can also display the calling sequence by opening the Calls window.

The Calls window also displays this information.

EXAMPLES

```
    whe
```

displays a list of the last 20 function calls.

```
    where 5
```

displays a list of the last 5 function calls.

```
    where a
```

displays the arguments to each function.

SEE ALSO

ARgs , ENV

1.77 WIndow

window - Opens or closes a window

SYNOPSIS

```
  wi[ndow] <window-name> [on | off]
```

DESCRIPTION

The window command opens or closes the specified window. You can specify any of the following window-names:

```
    c[alls]        mo[dules]      w[atch]
    he[lp]         ms[g]        s[ource]
    me[mory]         r[egister]
```

If you do not specify on or off, the window command toggles the specified window. (If the window is currently opened, the window command closes it, and if it is currently closed the window command opens it.)

If you specify on and the name of a window that is already open, the

```
specified window pops to the foreground.

The window command is ignored in line mode.

EXAMPLES
    wi register
  toggles the Register window.

    window help on
  opens the Help window or pops it to the top.
```

1.78 WList

```
wlist - Lists all watches

SYNOPSIS
  wl[ist]

DESCRIPTION
The wlist command displays a list of all watches and watch breaks.
wlist displays the list of all watch and watch breaks, as shown in
the following example:

   1   <dynamic> i : 42 (0x2A)
   2!  <static register> d : 21
   3 * <dynamic> q : 14 (0x0E)

The numbers 1, 2, 3, and 4 are watch or watch break numbers.  You
can use these numbers to identify a specific watch or watch break in
the wclear, wdisable, and wenable commands.  If the number is
followed by an asterisk (as with number 3), the watch or watch break
is disabled. An exclamation point (!) indicates a watch break.
Following the number is a description of the watch or watch point.
In the above example, watch number 1 is a watch on the variable i,
and its current value is 42 (0x2A in hexadecimal).

EXAMPLES
    wlist
  lists all watches.


SEE ALSO
 Watch ,  WBreak ,  WClear ,  WDisable ,  WEnable
```

1.79 WMSG

```
wmsg - Writes a message to the Message window

SYNOPSIS
  wmsg  <integer>  <message-text>

DESCRIPTION
```

```
The wmsg command writes the text specified by the message-text in
the Message window.  The integer parameter specifies the line number
in the window for the text.  Line 0 is the top line.  This command
is used primarily with AREXX scripts.

EXAMPLES
    wmsg 0 --- This is the Message window. ---
  displays the string --- This is the Message window. --- on
  the top line of the Message window.

    wmsg 12 This goes on line 12.
  displays the string This goes on line 12. on line 12 of the
  Message window.
```

1.80 MEMCMP

```
memcmp - Compares two memory blocks

SYNOPSIS
   i = memcmp(a, b, n);
   int i;      /* comparison results    */
   void *a, *b;      /* blocks being compared */
   int n;      /* block size in bytes   */

DESCRIPTION
The memcmp function compares two memory blocks and returns a value whose
sign indicates the collating sequence of the blocks, as follows:

  Return       Meaning
       --------    -----------------------
       Negative    First block below the second
   Zero    Blocks are equal
       Positive    First block above the second

EXAMPLES
    display memcmp(ptr1, ptr2, n)
  displays the result of comparing ptr1 and ptr2 for n bytes.

    b 27 when(memcmp(ptr1, ptr2, 10) < 0)
  breaks at line 27 when the 10 bytes pointed to by ptr1
  evaluate to a value less than the value of the 10 bytes
  pointed to by ptr2.

SEE ALSO
 MEMCPY ,  MEMMOVE ,  MEMSET
```

1.81 MEMCPY

```
memcpy - Copies a memory block (non-overlapping)

SYNOPSIS
   to = memcpy(to, from, n);
```

```
   void *to;  /* destination pointer */
   void *from;  /* source pointer      */
   int  n;  /* block size in bytes */

DESCRIPTION
The memcpy function copies data from one memory block to another.
The memcpy function cannot be used to copy overlapping memory
blocks.  You should use the memmove function when copying
overlapping blocks.

  CAUTION!!
    This function produces unpredictable results if there is not
    enough memory to hold the data at the destination memory
    location.

    On some systems, you can crash your machine.

EXAMPLES
    call memcpy(ptr1, &j, 10)
  copies 10 bytes from ptr2 to ptr1.

    call memcpy(0x804a, myptr, 50)
  copies 50 bytes from myptr to location 0x804a.

    b 27 {memcpy(a0, a1, 15)}
  sets a breakpoint at line 27 with an action to copy 15 bytes
  from the memory pointed to by register A1 to the memory
  pointed to by register A0.

SEE ALSO
 MEMCMP ,  MEMMOVE ,  MEMSET
```

1.82 MEMMOVE

```
memmove - Copies a memory block (possibly overlapping)

SYNOPSIS
   to = memmove(to, from, n);
   void *to;  /* destination pointer */
   void *from;  /* source pointer      */
   int  n;  /* block size in bytes */

DESCRIPTION
The memmove function copies data from one memory block to another.
Overlapping blocks are handled correctly.

CAUTION!!
   This function produces unpredictable results if there is not
   enough memory to hold the data at the destination memory
   location.

   On some systems, you can crash your machine.

EXAMPLES
    call memmove(ptr1, &j, 10)
```

```
  copies 10 bytes from ptr2 to ptr1.

    call memmove(0x804a, myptr, 50)
  copies 50 bytes from myptr to location 0x804a.

    b 27 {memmove(a0, a1, 15)}
  sets a breakpoint at line 27 with an action to copy 15 bytes
  from the memory pointed to by register A1 to the memory
  pointed to by register A0.

SEE ALSO
 MEMCPY ,  MEMCMP ,  MEMSET
```

1.83 MEMSET

```
memset - Sets memory to a specified value

SYNOPSIS
   to = memset(to, c, n);
   void *to   /* base of memory to be initialized  */
   int  c;    /* initialization value      */
   int  n;    /* number of bytes to be initialized */

DESCRIPTION
The memset function sets the specified number of bytes of memory to
the specified value.

  CAUTION!!
   This function produces unpredictable results if there is not
   enough memory to hold the data at the destination memory
   location.

   On some systems, you can crash your machine.

EXAMPLES
    call memset(ptr, 0, 100)
  sets 100 bytes to 0 starting at the address pointed to by
  ptr.

    call memset(a0, 'X', 50)
  sets 50 bytes to the ASCII character X starting at the
  address pointed to by register A0.

SEE ALSO
 MEMCPY ,  MEMCMP ,  MEMMOVE
```

1.84 STRCAT

```
strcat - Concatenates two strings

SYNOPSIS
   to = strcat(to, from);
```

```
   char *to;   /* destination pointer */
   char *from;   /* source pointer  */

DESCRIPTION
The strcat function copies data from one string to the end of
another string until a null character is found.  Do not use strcat
to copy overlapping strings.

CAUTION!!
  This function produces unpredictable results if there is not
  enough memory to hold the data at the destination memory location.

  On some systems, you can crash your machine.

EXAMPLES
    call strcat(myptr, "foo")
  concatenates the string "foo" after the string pointed to by
  the variable myptr.

SEE ALSO
 STRLEN ,  STRCMP ,  STRCPY
```

1.85 STRCMP

```
strcmp-Compares two strings

SYNOPSIS
   i = strcmp(a, b);
   int  i;   /* comparison result    */
   char *a, *b;  /* strings being compared */

DESCRIPTION
The strcmp function compares two strings and returns a value whose
signs indicate the collating sequence of the blocks as follows:

  Return       Meaning
       --------    ------------------------
       Negative    First string below the second
   Zero    Strings are equal
       Positive    First string above the second


EXAMPLES
    display strcmp("abc", "def")
  tests the strings "abc" and "def" and displays the return
  value. The value indicates whether the strings are equal or
  which string is higher.

    break myfunc when(strcmp(arg, "foobar") == 0)
  breaks at the function myfunc when the variable arg points
  to the string "foobar".

SEE ALSO
 STRLEN ,  STRCPY ,  STRCAT
```

1.86 STRCPY

```
strcpy - Copies a string

SYNOPSIS
   to = strcpy(to, from);
   char *to;     /* destination pointer */
   char *from;     /* source pointer    */

DESCRIPTION
The strcpy function copies data from one string to another until a
null character is found.  Do not use the strcpy function to copy
overlapping strings.

CAUTION!!
   This function produces unpredictable results if there is not
   enough memory to hold the data at the destination memory
   location.

   On some systems, you can crash your machine.

EXAMPLE
    call strcpy(a, b)
  copies the string pointed to by b to the memory location
  pointed to by a.


SEE ALSO
 STRLEN ,  STRCMP ,  STRCAT
```

1.87 STRLEN

```
strlen - Returns the length of a string

SYNOPSIS
   len = strlen(s);
   int  len;  /* length of string s      */
   char *s; /* string to scan for length */

DESCRIPTION
The strlen function returns the length of a string, as determined by
the index of the first null character found.

EXAMPLE
    display strlen("hello")
  displays the length of the string "hello", which is 5.

    display strlen(0x804a)
  displays the length of the string starting at address
  0x804a.

SEE ALSO
 STRCMP ,  STRCPY ,  STRCAT
```

1.88 HELP

```
    Getting Around
    Organization
```

1.89 getting around

```
You have reached this Help window by either clicking on the Help
button or by hitting the Help key within the SAS/C Help utility.
Unlike other help topics present in the SAS/C Help utility, the Help
help topic opens its own window.  You must close this window by
clicking on the close gadget or hitting escape before returning to
the SAS/C help utility.  You cannot hit the Retrace button to return.

To quit the SAS/C Help utility, select Quit from the Project menu or
click on the close gadget.  You may also hit escape.

Most help screens will display one or more buttons as part of the text.
Clicking on these buttons will provide further information on the topic
listed on the button.  You can also reach these help topics through the
main Contents screen or one of its sub-screens.

In addition, double-clicking in the help window will bring up a help
screen for the word under the mouse cursor, if such a help screen
exists.

While in the SAS/C Help utility, you may retrace your steps through the
help screens you have selected by clicking on the Retrace button.

The Browse buttons will move you forward and backwards between help
screens.  The help screens are usually arranged alphabetically by
command or topic.
```

1.90 ORGANIZATION

```
The CPR Amigaguide Help Utility is organized into three sections.  The
sections are:

      - Commands and Builtin Functions
      - Parameter Types
     - Common Problems

The first screen (the screen displayed when you select the "Contents"
button) displays an alphabetized list of all commands and builtin
functions that you can use in CPR.  Selecting one of these buttons will
display a window with sections titled "SYNOPSIS", "DESCRIPTION",
"EXAMPLES", and "SEE ALSO".

The "SYNOPSIS" section describes the syntax of the command, including
any parameters that it may take.  A parameter is the part of the command
that is specific to each situation in which you use it.  For example,
the command "display my_variable" displays the contents of a variable
```

called my_variable. The name of the variable can vary, so this is represented in the "SYNOPSIS" section by placing the word "variable" on a button in its proper place in the syntax. Clicking on this button will pop up a description of what can replace "variable" in the syntax for the command.

The "DESCRIPTION" section describes what the command does. If present, the "EXAMPLES" section gives some common examples that will help explain the use of the command. The "SEE ALSO" section lists other related commands that either have a similar function or that can be used in conjuction with the command. The commands listed in the "SEE ALSO" section are equipped with links to allow you to quickly reference them.

The first screen also displays two buttons that will lead you to the Parameter Types and Common Problems windows. In the Parameter Types window, buttons are displayed for all of the different types of parameters that the CodeProbe commands and builtin functions may take. Selecting one of these buttons will pop up the same description for each that is displayed from the parameter buttons in the "SYNOPSIS" section of each command.

The Common Problems section described some problems that you might run into while running CPR for which there is a simple solution. If you have questions that aren’t answered in this question, please contact Technical Support. Please read the section titled "Contacting Technical Support" in the User’s Guide volume 1 before doing so.