Amalasoft Printf

Introduction

The Amalasoft printf is part of the Amalasoft Eiffel Library (AEL). It is a collection of classes that provides a printf facility for the Eiffel language. It depends on the Eiffel base libraries (available in open source or commercial form from Eiffel Software) and is, like other Eiffel code, portable across platforms.

In use, the AEL printf routines are quite simple and are very reminiscent of the `printf` function in C and its close relatives.

Here is an example using AEL `printf` that prints a table-row-like line including a number and 3 strings (right, center and left justified).

```
  printf ("|%%3d|%%10s|%%=10s|%%-10s|%N", <<1, "right", "center", "left">>)
```

This call produces the following output

```
  |  1|     right|  center  |left      |
```

Unlike classic C `printf`, AEL printf routines are resilient. For example, a mismatched or empty argument list will not cause an illegal memory access, as C `printf` often can.

There is no option to omit the argument list, as there is in C. Doing so would be a syntax error, caught by the Eiffel compiler. If for some reason you feel the need to call one of the AEL printf routines without an argument list, you must provide an empty manifest `ARRAY` ("<<>>"), an empty `TUPLE` ("`[]`") or an explicit `Void`.

A mismatched *format-to-argument* pair will not cause an exception; not even the ever-popular segmentation fault that often happens with C `printf`. It will instead produce an output that is readily detectable and therefore debug-able. It will not be what you wanted, but it will be what you requested.

If there are too many arguments, AEL `printf` will consume the available arguments as needed, in sequence, and ignore the rest. An error condition will be recorded, but the output will not show it. Your application can check on the error after the fact, or can register an agent to be called whenever an error occurs within the AEL printf code.

If there are too few arguments, AEL `printf` will insert a literal "Void" (the string, not a Void reference) in place of the missing argument.

The Amalasoft printf cluster does not include an equivalent to scanf at this time.

Incorporating AEL Printf in Your Project

The AEL printf cluster uses the standard Eiffel base libraries. To incorporate AEL printf, simply add the AEL printf cluster or library to your configuration file.

Any classes that wish to use the printf routines must either inherit AEL_PRINTF or instantiate it (using default creation).

The cluster includes several classes, but only the AEL_PRINTF class offers a client interface. Following is the contract view of the AEL_PRINTF class.

```
class interface
  AEL_PRINTF
create
  default_create


feature -- Convenience function


  print_line (some_text: detachable ANY)
      -- Print terse external representation of 'some_text'
      -- on standard output, followed by a newline character
      -- Was declared in AEL_PRINTF as synonym of printline.


  printline (some_text: detachable ANY)
      -- Print terse external representation of 'some_text'
      -- on standard output, followed by a newline character
      -- Was declared in AEL_PRINTF as synonym of print_line.


feature -- Debug assistance

  set_client_log_proc (v: PROCEDURE [ANY, TUPLE [STRING_8]])
      -- Set the procedure to call to log a message for the client
      -- For debugging support only


feature -- Error Status


  last_printf_error: detachable AEL_PF_FORMAT_ERROR
      -- Most recent error, from most recent operation
      -- Void if no errors occurred
```

```
last_printf_error_out: STRING_8
    -- Description from most recent error (if any)
  ensure
    exists: Result /= Void


Last_printf_errors: LINKED_LIST [AEL_PF_FORMAT_ERROR]


last_printf_successful: BOOLEAN
    -- Was the most recent printf operation a success?
  ensure
    no_false_positive: Result implies Last_printf_errors.is_empty
    no_misses: (not Result) implies not Last_printf_errors.is_empty
    positive_inverse: (not Last_printf_errors.is_empty) implies not Result
    negative_inverse: Last_printf_errors.is_empty implies Result
```

```
feature -- Formatting

  amemdump (p: POINTER; sz: INTEGER_32;
                opt: STRING_8; sa, cs: INTEGER_32): STRING_8
      -- Hex dump of memory locations starting at address 'p' and
      -- continuing for 'sz' bytes
      --
      -- 'opt' conveys formatting options
      -- 'opt' can contain 0 or more characters
      --
      -- If included, 'A' denotes "DO NOT show ASCII chars at line
      -- end"
      -- If included, 'a' denotes "show ASCII chars at line end"
      --   Default, included for completeness
      -- If included, 'd' denotes "show addresses as decimal"
      --   Default, included for completeness
      -- If included, 'w' denotes "wide format"
      --   64 byte sequences (default is 16)
      -- If included, 'x' denotes "show addresses as hexadecimal"
      --   Default is to show addressed in decimal
      --
      -- 'sa' is the starting address to associate with position 0
      --
      -- If 'cs' is positive, then add a blank line every
      -- 'cs' sequences (cs<=0 suppresses line insertion)
    require
      valid_address: p /= default_pointer
      valid_size: sz > 0
    ensure
      exists: Result /= Void


  aprintf (fmt: STRING_8; args: detachable ANY): STRING_8
      -- A new string object formatted according to
      -- the given format 'fmt' and arguments 'args'
      -- 'args' can be a TUPLE, a data structure conforming to
      -- FINITE, or, if no arguments are needed, simply Void
    require
      format_string_exists: fmt /= Void
    ensure
      exists: Result /= Void
```

```
axdump (buf, opt: STRING_8; ss, es, sa, cs: INTEGER_32): STRING_8
    -- Hex dump of buffer 'buf' with options 'opt'
    --
    -- 'opt' can contain 0 or more characters denoting output
    -- options
    --
    -- If included, 'A' denotes "DO NOT show ASCII chars at line
    -- end"
    -- If included, 'a' denotes "show ASCII chars at line end"
    --   Default, included for completeness
    -- If included, 'd' denotes "show addresses as decimal"
    --   Default, included for completeness
    -- If included, 'w' denotes "wide format"
    --   64 byte sequences (default is 16)
    -- If included, 'x' denotes "show addresses as hexadecimal"
    --   Default is to show addressed in decimal
    --
    -- 'ss' and 'es' are start and end sequence numbers to
    -- included in Result.  By default, output includes all
    -- sequences in 'buffer'
    -- If 'ss' = 0, then the assumed start is '1'
    -- If 'es' = 0 then end is end of the buffer
    -- If 'es' is greater than the total number of sequence,
    -- then end is the end of the bugger
    --
    -- 'sa' is the starting address to associate with position 0
    --
    -- If 'cs' is positive, then add a blank line every
    -- 'cs' sequences (cs<=0 suppresses line insertion)
  require
    valid_buffer: buf /= Void and then not buf.is_empty
  ensure
    exists: Result /= Void
```

```
fprintf (f: FILE; fmt: STRING_8; args: detachable ANY)
    -- Write to the end of given open FILE a string formatted
    -- according to the given format 'fmt' and arguments 'args'
    -- 'args' can be a TUPLE, a data structure conforming to
    -- FINITE, or, if no arguments are needed, simply Void
  require
    exists: f /= Void
    file_exists: f.exists
    file_is_open: f.is_open_write or f.is_open_append
    format_string_exists: fmt /= Void
```

```
lmemdump (p: POINTER; sz: INTEGER_32; opt: STRING_8;
          ss, es, sa: INTEGER_32): LINKED_LIST [ARRAY [STRING_8]]
    -- Hex dump of memory locations starting at address 'p' and
    -- continuing for 'sz' bytes, in the form of a list of rows
    -- or strings, each with 3 sections (position, values, ascii
    -- chars)
    --
    -- For use by list-oriented user interfaces
    --
    -- 'ss' and 'es' are start and end sequence numbers to
    -- included in Result.  By default, output includes all
    -- sequences in 'buf'
    -- If 'es' = 0 then end is last sequence in buffer
    --
    -- 'opt' conveys formatting options
    -- 'opt' can contain 0 or more characters
    --
    -- If included, 'A' denotes "DO NOT show ASCII chars at line
    -- end"
    -- If included, 'a' denotes "show ASCII chars at line end"
    --   Default, included for completeness
    -- If included, 'd' denotes "show addresses as decimal"
    --   Default, included for completeness
    -- If included, 'w' denotes "wide format"
    --   64 byte sequences (default is 16)
    -- If included, 'x' denotes "show addresses as hexadecimal"
    --   Default is to show addressed in decimal
    -- 'sa' is the starting address to associate with position 0
  require
    valid_address: p /= default_pointer
    valid_size: sz > 0
  ensure
    exists: Result /= Void
```

```
lxdump (buf, opt: STRING_8;
        ss, es, sa: INTEGER_32): LINKED_LIST [ARRAY [STRING_8]]
    -- Hex dump of buffer 'buf' in the form of a list of rows of
    -- strings, each with 3 sections
    -- For use by list-oriented user interfaces
    --
    -- 'ss' and 'es' are start and end sequence numbers to
    -- included in Result.  By default, output includes all
    -- sequences in 'buf'
    -- If 'es' = 0 then end is last sequence in buffer
    --
    -- 'opt' can contain 0 or more characters denoting output
    -- options
    --
    -- If included, 'd' denotes "show addresses as decimal"
    --   Default, included for completeness
    -- If included, 'w' denotes "wide format"
    --   64 byte sequences (default is 16)
    -- If included, 'x' denotes "show addresses as hexadecimal"
    --   Default is to show addressed in decimal
    -- 'sa' is the starting address to associate with position 0
  require
    valid_buffer: buf /= Void and then not buf.is_empty
  ensure
    exists: Result /= Void


printf (fmt: STRING_8; args: detachable ANY)
    -- Write to the standard output a string formatted
    -- according to the given format 'fmt' and arguments 'args'
    -- 'args' can be a TUPLE, a data structure conforming to
    -- FINITE, or, if no arguments are needed, simply Void
  require
    format_string_exists: fmt /= Void
  sprintf (buf, fmt: STRING_8; args: detachable ANY)
      -- Replace the given STRING 'buf''s contents
      -- with a string formatted according to
      -- the format 'fmt' and arguments 'args'
      -- 'args' can be a TUPLE, a data structure conforming to
      -- FINITE, or, if no arguments are needed, simply Void
    require
      buffer_exists: buf /= Void
      format_string_exists: fmt /= Void


feature -- Global status setting

  reset_default_printf_decimal_separator
      -- Reset the character used to denote the decimal point


  reset_default_printf_fill_char
      -- Reset the default fill character to blank


  reset_default_printf_list_delimiter
      -- Reset the default list delimiter string


  reset_default_printf_thousands_delimiter
      -- Reset the default thousands delimiter string


  set_default_printf_decimal_separator (v: CHARACTER_8)
      -- Change the character used to denote the decimal point
      -- to 'v' for ALL subsequent printf calls in this thread space


  set_default_printf_fill_char (v: CHARACTER_8)
      -- Change the fill character from blank to
      -- the given new value for ALL subsequent
      -- printf calls in this thread space


  set_default_printf_list_delimiter (v: STRING_8)
      -- Change the default list delimiter string from a single
      -- blank character to the given string for ALL subsequent
      -- printf calls in this thread space
  set_default_printf_thousands_delimiter (v: STRING_8)
      -- Change the default thousands delimiter string from an
      -- empty string to the given string for ALL subsequent
      -- printf calls (in this thread space)


  set_printf_client_error_agent (
              v: detachable PROCEDURE [ANY, TUPLE [AEL_PF_FORMAT_ERROR]])
      -- Set the procedure to call upon encountering a format error

end -- class AEL_PRINTF
```

Using AEL Printf Routines

The printf routines provide a means by which to format strings for output or other purposes in a manner reminiscent of the traditional printf functions in C and similar languages.

Format string construction (in order):

```
%
[<decoration_flag>]
[<agent_flag>]
[<alignment_flag>]
[<fill_specifier>]
[<field_width>]
<field_type>
```

Where:

The '%' character denotes a format specifier, as it does in C `printf`.

The '%' character is also Eiffel's escape character. As such, when creating a format string, be sure either to use a verbatim string, or to add another '%' character before each format specifier, lest Eiffel treat it as an escape character.

For example "`name=%s`", if not a verbatim string, will be interpreted by Eiffel as an attempt to use 's' as a character code, and because 's' is not an Eiffel character code, the compiler will flag it as a syntax error.

To compensate, simply double up the '%' characters. The successful form would then be "`name=%%s`".

When using verbatim strings, Eiffel does not interpret the '%' character, and so only a single '%' is needed in that case.

"{
name=%s
}"

`<decoration_flag> ::=  '#'`

Decoration consumes part of the field width

Decoration is applied as follows:

```
"0x" preceding hexadecimal values
"0" preceding octal values
"b" following binary values
```

Decimal values show delimiters at thousands (commas by default)

`<agent_flag> ::=  '~'`

Valid for List formats only. Cannot be combined with decoration flag

```
<alignment_flag> ::=   '-' |   '+'  |  '='
```

(left   right centered)

```
<fill_specifier> ::=  <character>
```

Fills remainder of field width with given character (default is blank)

```
<field_width> ::=  <simple_width> | <complex_width>
<field_type> ::=  <character>
```

Field type can be at least one of the following:

A denotes an Agent expression. Argument must be a function that accepts an argument of type ANY and returns a STRING.

B denotes a BOOLEAN expression

This shows as "True" or "False"

b denotes a Binary INTEGER expression

This shows as ones and zeroes.
If no field width is specified, the field width will be the smallest whole integer size (8, 16, 32, 64) that can hold the value of the argument. Values from 0 through 255 have an implicit field width of 8, values between 256 and 65535 have 16, values between 65536 and 4294967295 have 32, and larger values have an implicit field width of 64.
When a field width *is* specified, the default padding character is blank.
A zero padding character can also be specified (as with other integral types) for positive values, but when the value begin rendered is negative, the pad character used is a ‘1’.

c denotes a single CHARACTER

d denotes a Decimal INTEGER expression

Type specifier can be preceded by a delimiter character with which to separate groups of 3 adjacent digits (thousands).
Alignment characters cannot be used as delimiters.

f denotes a REAL or DOUBLE expression

Field width for floating point values are given in the form:

```
<overall_width>"."<right_width>
```

Where overall_width is the minimum width of the entire representation, and <right_width> is the width for the fractional part (a.k.a. precision).
A precision of 0 results in a whole number representation, without decimal point (effectively rounding to integer)

L denotes a list-like expression (any FINITE container)

Type specifier can be preceded by a delimiter character with which to separate list items (default is blank).

Alignment characters cannot be used a delimiters.

In place of a delimiter, the agent flag ('~') can be used. In that case, the argument must be a TUPLE [CONTAINER,FUNCTION] instead of a container

	alone.
o	denotes an Octal INTEGER expression
P	denotes an Percent expression (float value multiplied by 100 and followed by a percent symbol
s	denotes character STRING
u	denotes a NATURAL (unsigned Decimal integer) expression
x	denotes a Hexadecimal INTEGER expression

In use, a class calls one of the printf routines with at least a format string and an argument list.

The argument list can be of any type, but for expected behavior, there are a few restrictions.

- When the format string has no format specifiers, then the argument list can be an empty manifest `ARRAY`, and empty `TUPLE`, or an explicit **`Void`**.

  Example: `printf (“This has no format specifiers%N”, `**`Void`**`)`

  Example: `printf (“This has no format specifiers%N”, <<>>)`

  Example: `printf (“This has no format specifiers%N”, [])`

- When the format string contains a single format specifier, then the argument list can be either a container with a single item of a type conforming to the single format specifier, or an object of a type conforming to the single format specifier.

  Example: `printf (“This has %%d format specifier%N”, 1)`

  Example: `printf (“This has %%d format specifier%N”, <<1>>)`

- When the format string contains multiple format specifiers, then the argument list must be either a `TUPLE` or a proper descendent of `FINITE` [`ANY`], in which each item, related by position to its corresponding format specifier, has a type conforming to its corresponding format specifier.

  Example: `printf (“This has %%s (%%d) format specifiers%N”, <<”multiple”, 2>>)`

Shared Printf Settings

The AEL printf cluster supports shared (i.e. *once-per-thread*) settings to control or modify certain behaviors.

> *These are once-per-thread, rather than once-per-process to be as flexible as possible.*

Error Reporting

AEL printf lets you define an agent to be called when an error is encountered in the printf routines. To define the agent, call the following function.

```
set_printf_client_error_agent (
    v: detachable PROCEDURE [ANY, TUPLE [AEL_PF_FORMAT_ERROR]])
          -- Set the procedure to call upon encountering a format error
```

Padding Characters and Delimiters

Default Fill Character

The default padding/fill character is a blank (ASCII 32). If so desired, the default padding character can be changed, for all calls to AEL printf routines, to a different character. To change the default padding/fill character, call the following function.

```
set_default_printf_fill_character (v: CHARACTER)
          -- Change the fill character from blank to
          -- the given new value for ALL subsequent
          -- printf calls in this thread space
```

To reset the default padding/fill character to blank, call:

```
reset_default_printf_fill_character
          -- Reset the default fill character to blank
```

Default Decimal Separator

The default decimal separator (aka radix point), as used in floating point formats, is a period (ASCII 46).). If so desired, the default decimal separator can be changed, for all calls to AEL printf routines, to a different character. To change the default decimal separator, call the following function.

```
set_default_printf_decimal_separator (v: CHARACTER)
          -- Change the character used to denote the decimal point
          -- to 'v' for ALL subsequent printf calls in this thread space
```

To reset the default decimal separator to a period, call:

```
reset_default_printf_decimal_separator
          -- Reset the character used to denote the decimal point
```

Default List Item Delimiter

The default list delimiter, as used between items in list/container formats, is a single blank (ASCII 32).). If so desired, the default list item delimiter can be changed, for all calls to AEL printf routines, to a different *string*. To change the default list delimiter, call the following function.

```
set_default_printf_list_delimiter (v: STRING)
          -- Change the default list delimiter string from a single
          -- blank character to the given string for ALL subsequent
          -- printf calls in this thread space
```

To reset the default list delimiter to a single space, call:

```
reset_default_printf_list_delimiter
          -- Reset the default list delimiter string
```

Default Thousands Delimiter

The default delimiter, as used between groups of three adjacent digits (thousands) in decimal integer formats, is a comma (ASCII 44). Decimal integers are rendered without thousands separators unless the decoration flag is set.

If so desired, the default thousands delimiter can be changed, for all calls to AEL printf routines, to a different character. To change the default thousands delimiter, call the following function.

```
set_default_printf_thousands_delimiter (v: CHARACTER)
          -- Change the default thousands delimiter string from an
          -- empty string to the given string for ALL subsequent
          -- printf calls (in this thread space)
```

To reset the default thousands delimiter to a single space, call:

```
reset_default_printf_thousands_delimiter
          -- Reset the default thousands delimiter string
```

Printf Examples

Strings

```
printf (“I’m a little %%s, short and stout%N”, “teapot”)
```

```
I’m a little teapot, short and stout
```

```
printf (“Here is my %%s.  Here is my %%s.%%N”, <<”handle”, “spout”>>)
```

```
Here is my handle.  Here is my spout.
```

```
printf (“The string is right aligned in 16 spaces ->%%16s<-%N”, “string”)
```

```
The string is right aligned in 16 spaces ->          string<-%
```

```
printf (“The string is right aligned in 16 spaces ->%%+16s<-%N”, “string”)
```

```
The string is right aligned in 16 spaces ->          string<-%
```

```
printf (“The string is left aligned in 16 spaces ->%%-16s<-%N”, “string”)
```

```
The string is left aligned in 16 spaces ->string          <-%
```

```
printf (“The string is centered in 16 spaces ->%%=16s<-%N”, “string”)
```

```
The string is centered in 16 spaces ->     string     <-%
```

Decimal Integers

```
printf (“You are %%d in %%d%N”, << 1, 1000000 >>)
```

```
You are 1 in 1000000
```

```
printf (“You are %%d in %%#d%N”, << 1, 1000000 >>)
```

```
You are 1 in 1,000,000
```

```
printf (“You are %%d in %%,d%N”, << 1, 1000000 >>)
```

```
You are 1 in 1,000,000
```

```
printf (“You are %%d in %%_d%N”, << 1, 1000000 >>)
```

```
You are 1 in 1_000_000
```

```
printf (“You are %%d in ->%%#17d<- (right)%N”, << 1, 1000000 >>)
```

```
You are 1 in ->        1,000,000<-
```

```
printf (“You are %%d in ->%%#-17d<- (left)%N”, << 1, 1000000 >>)
```

```
You are 1 in ->1,000,000        <-
```

```
printf (“You are %%d in ->%%#=17d<- (center)%N”, << 1, 1000000 >>)
```

```
You are 1 in ->    1,000,000    <-
```

```
printf ("You are %%d in ->%%=17d<- (center)%N", << 1, 1000000 >>)
```

```
You are 1 in ->      1000000      <-
```

Floating Point Numbers

```
printf ("A coin lands on heads %%2.0f%% of the time%N", << 50 >>)
```

```
A coin lands on heads 50% of the time
```

```
printf ("A coin lands on heads %%2.0f%% of the time (P=%%1.2f)%N", <<50, 0.5>>)
```

```
A coin lands on heads 50% of the time
```

```
printf ("The sqrt of 2 is %%1.4f to 4 places%N", {MATH_CONST}.sqrt2)
```

```
The sqrt of 2 is 1.4142 to 4 places
```

```
printf ("The sqrt of 2 is %%1.8f to 8 places%N", {MATH_CONST}.sqrt2)
```

```
The sqrt of 2 is 1.41421356 to 8 places
```

```
printf ("The sqrt of 2 is ->%%10.4f<- to 4 places, in 10 columns (right)%N",
          {MATH_CONST}.sqrt2)
```

```
The sqrt of 2 is ->    1.4142<- to 4 places, and in 10 columns
```

```
printf ("The sqrt of 2 is ->%%-10.4f<- to 4 places, in 10 columns (left)%N",
          {MATH_CONST}.sqrt2)
```

```
The sqrt of 2 is ->1.4142    <- to 4 places, and in 10 columns
```

```
printf ("The sqrt of 2 is ->%%=10.4f<- to 4 places, in 10 columns (center)%N",
          {MATH_CONST}.sqrt2)
```

```
The sqrt of 2 is ->  1.4142  <- to 4 places, and in 10 columns
```

Containers

```
printf ("The value in this list are: %%L%N",<< <<1, 79, 2, 1492>> >>)
```

```
The value in this list are: 1 79 2 1492
```

```
printf ("The value in this list are: %%,L%N",<< <<1, 79, 2, 1492>> >>)
```

```
The value in this list are: 1,79,2,1492
```

```
printf ("The value in this list are: %%:L%N",<< <<1, 79, 2, 1492>> >>)
```

```
The value in this list are: 1:79:2:1492
```

Binary Integers

```
printf ("The binary form of %%d (%%#x) is %%#b%N",<< 17, 17, 17 >>)
```

```
The binary form of 127 (0x7f) is 00010001b
```

```
printf ("The binary form of %%d (%%#x) is %%b%N",<< 17, 17, 17 >>)
```

```
The binary form of 127 (0x7f) is 00010001
```

```
printf ("The binary form of %%d (%%#x) is ->%%12b<-%N",<< 17, 17, 17 >>)
```

```
The binary form of 127 (0x7f) is ->    00010001<-
```

```
printf ("The binary form of %%d (%%#x) is ->%%012b<-%N",<< 17, 17, 17 >>)
```

```
The binary form of 127 (0x7f) is ->000000010001<-
```

```
printf ("The binary form of %%d (%%04x) is %%b%N",<< 127, 127, 127 >>)
```

```
The binary form of 127 (007f) is 01111111
```

```
printf ("The binary form of %%d (%%04x) is %%b%N",<< 65535, 65535, 65535>>)
```

```
The binary form of 65535 (ffff) is 1111111111111111
```

```
printf ("The binary form of %%d (%%#04x) is %%b%N",<< 65536, 65536, 65536>>)
```

```
The binary form of 65536 (0x1000) is 00000000000000010000000000000000
```

```
printf ("The binary form of %%d (%%#04x) is %%20b%N",<< 65536, 65536, 65536>>)
```

```
The binary form of 65536 (0x1000) is 00010000000000000000
```

Verbatim Strings for Formats

When using a verbatim string as a format string, there should be no escape characters preceding the format specifiers. In other words, the format specifiers require only single percent symbols when in verbatim strings.

```
printf ("{
This is the first line, and has a quoted string ->"%s"<- here.
This is the second line, and has an integer ->%d<- here.
This is the third line and has no specifiers.
This is line %d, and we're at 100%.
}"
          ,<< "string", 42, 4 >>)
```

```
This is the first line, and has a quoted string ->"string"<- here.
This is the second line, and has an integer ->42<- here.
This is the third line and has no specifiers.
This is line 4, and we're at 100%.
```

Hex Dump Routines

The AEL_PRINTF class includes routines for generating hexadecimal dumps (not exactly classic printf, but formatted strings certainly). The contract forms of these routines appear earlier in the document.

There are four variants of these routine (two source variants and two output variants).

	From STRING	From POINTER
To STRING	axdump	amemdump
To LIST	lxdump	lmemdump

Table 1 - Hex Dump Routines

The hex dump routines each accept an options string argument. The options string can contain 0 or more case-sensitive characters.

Table 2 lists the possible values for characters in the options string arguments to the axdump and amemdump routines.

A	Do NOT show ASCII characters at line end
a	Show ASCII characters at line end (default)
d	Show addresses as decimal (default)
w	Use wide output format (64-bytes per line; default is 16 bytes per line)
x	Show addressed in hexadecimal format (default is decimal)

Table 2 - Options String Values for axdump and amemdump

The possible values for characters in the options string arguments to the lxdump and lmemdump routines are a subset of those for axdump and amemdump, and appear in Table 3.

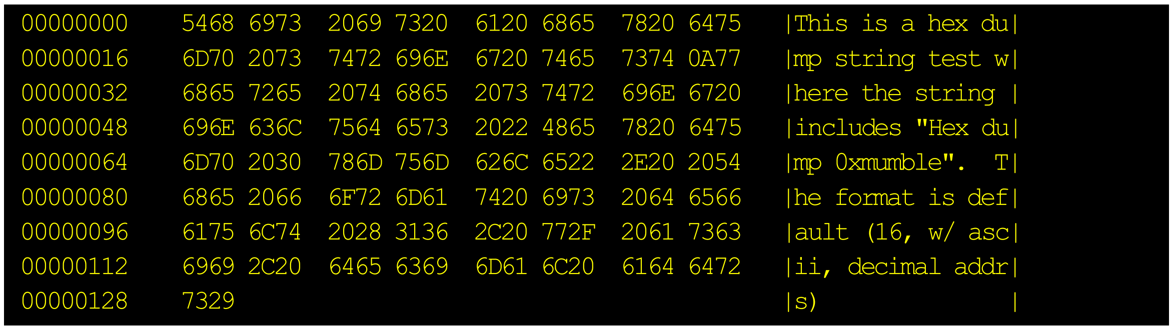
d	Show addresses as decimal (default)
w	Use wide output format (64-bytes per line; default is 16 bytes per line)
x	Show addressed in hexadecimal format (default is decimal)

Table 3 - Options String Values for lxdump and lmemdump

Following is the output resulting from calling axdump with default values and source string of:

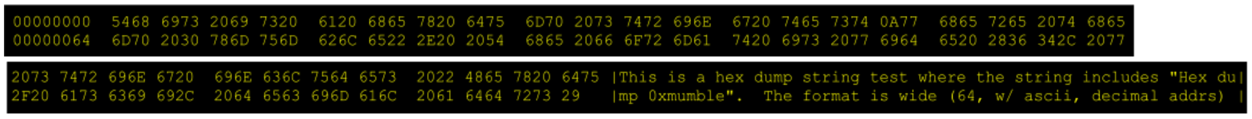
```
This is a hex dump string test
where the string includes "Hex dump 0xmumble".  The format is default (16, w/
ascii, decimal addrs)”
```

Output:



```
00000000    5468 6973  2069 7320  6120 6865  7820 6475   |This is a hex du|
00000016    6D70 2073  7472 696E  6720 7465  7374 0A77   |mp string test w|
00000032    6865 7265  2074 6865  2073 7472  696E 6720   |here the string |
00000048    696E 636C  7564 6573  2022 4865  7820 6475   |includes "Hex du|
00000064    6D70 2030  786D 756D  626C 6522  2E20 2054   |mp 0xmumble".  T|
00000080    6865 2066  6F72 6D61  7420 6973  2064 6566   |he format is def|
00000096    6175 6C74  2028 3136  2C20 772F  2061 7363   |ault (16, w/ asc|
00000112    6969 2C20  6465 6369  6D61 6C20  6164 6472   |ii, decimal addr|
00000128    7329                                         |s)              |
```

Below is an example of the output from axdump when the wide format (‘w’) option is requested. Note that the limitations of page width in the document require that the output be shown in two parts. This is not the way the actual output is handled (it is contiguous)



```
00000000  5468 6973 2069 7320  6120 6865 7820 6475  6D70 2073 7472 696E  6720 7465 7374 0A77  6865 7265 2074 6865
00000064  6D70 2030 786D 756D  626C 6522 2E20 2054  6865 2066 6F72 6D61  7420 6973 2077 6964  6520 2836 342C 2077
```

```
2073 7472 696E 6720  696E 636C 7564 6573  2022 4865 7820 6475 |This is a hex dump string test where the string includes "Hex du|
2F20 6173 6369 692C  2064 6563 696D 616C  2061 6464 7273 29   |mp 0xmumble".  The format is wide (64, w/ ascii, decimal addrs) |
```

The list-generating forms of the routines produce lists of strings, each representing a row of hex dump output. The list forms are provide to let clients present the information in different forms as desired.