Using the am1 macro assembler for the PDP-1

This document describes the **am1** macro assembler and how to use it

What is am1?

(Yet) Another Macro assmbler for the PDP-1.

Why do we need another one? Simply because the avaialble ones seem to essentially duplcate the old **macro1** assembler with minor tweaks. It's time for a modern assembler that brings productive features and better error checking and better readability.

Am1 provides some significant new features while maintaining some compatibility with existing macro sources. Additionally, a separate conversion tool, **mactoam1**, is provided to process an existing macro source file and convert it to **am1** syntax.

It can, in fact, produce as its output valid source that can be assembled by **macro1**.

The features and differences

First, two different assembly modes are supported, generating of **macro1** source and direct generation of rim-lodable binary code. While most of the language constructs apply to both, one, *bank*, is not supported by **macro1**. Code will still be generated but should just be for reference as any code in extended memory will just overlay code in bank 0.

Features

- Uses **cpp** to bring the full power of **cpp** to the code
- Provides local variables and local scoping
- Provides a new, explicit, variable support with initializers
- Allows easy use of ascii characters and strings
- Supports octal, decimal, and hex numbers without special statements
- Adds operators for and, or, xor, complement, multiply, divide in expressions
- Can select between keeping -0 or automatically converting to +0 in expressions
- Allows space-separated symbols to be treated as *a | b | c* instead of *a + b + c*
- Adds parenthesized expressions
- Adds actual operator precedence
- Supports extended memory
- Can share location symbols across memory banks
- Can generate either **macro1** source or rim format binary code output
- The rim format loads a new loader to support extended memory use
- Does **not** allow redefining symbols (but cpp defines can)
- Does **not** treat tab as a statement delimiter
- 'Punches' the readable program header line in binary mode, like the original **macro** does

Differences in source code

These changes were done to allow the above changes and to remove some of the ambiguous use of characters in **macro1** that were poorly handled.

- Constants are now of the form *[xxx* and *[xxx]*, not *(xxx* and *(xxx)*
- Multiple statemens on a line are separated by *;* not tab
- Tabs are not statement delimiters and are treated as a space
- Cpp directives of the form *#xxx* are supported
- The operators *& | ~ ^ * /* as well as *(expr)* are added
- *%* is added to indicate a local variable, *%xxx*
- The numeric prefixes 0x, 0d, 0o are added to mean hex, decimal, octal regardless of the current radix
- *'x'* is allowed to mean the value of a single ascii character, the usual escapes are recognized
- *ascii "xxx"* is provided to create a table of ascii character values similar to *text*
- The directives *local* and *endloc* are provided to control local scoping
- The directive *bank* is provided to control placement in extended memory
- Location symbols in one bank can be referenced from another bank by using the *:banknum* suffix
- Explicit variable declarations with optional initializers is added, *var name=expr, name...*
- Tables with optional initializers are added, *table 10, 7+5*
- Macro1 style define-terminate blocks are **not** supported, use **cpp** #defines

A note on macro vs macro1

No particular effort is made to remain compatible with the original native **macro**, although if symbol lengths are limited to be unique in the first 3 characters and operators added in **macro1** aren't used, the generated code should be compatible.

Compatibility with **macro1** is provided as long as symbols are unique within the first 6 characters.

A **note** about **macro1** *define* and *terminate* constructs - as noted, these are not supported. But, see next.

But what about existing macro1 sources?

Never fear, help is here! - *Dudley Do-right*

The **mactoam1** utility, see its documentation, will convert existing source code to **am1** with good reliability, including converting **macro1** defines into the equivalent **cpp** defines.

If you have only a binary image, then use the **disassemble_tape** utility in its macro source mode to create a macro source file you can then process as above.

And of course, you can go back to macro sources from **am1** by using the macro source output flag.

What goes on inside?

Unlike other PDP-1 assemblers, this is built with modern(ish) tools. It uses lex and yacc (flex and bison) for the lexical analysis and parsing. The result of that is a parse tree representing the program.

Having a parse tree means it's easy to plug in different code generators. Just walk the tree and emit what you want.

Locals are implemented with a local symbol table that is created when a local scope begins. It's pushed on a stack of local tables if a nested local scope is encountered. Each ending of a scope updates all the scope's local symbols to reflect the value of the pc where they are allocated and pops back to the next higher context, if any.

All references within the scope are then emitted as pc-relative so no externally visible symbols are created.

Constants are handled specially to allow for constant reuse. When a constant is encountered, its defining expression is used to create a hash code that is saved as the key in the constants symbol table. Thus, the next time the same value is seen the existing entry will be used. When a *constants* directive is seen, all constants added since the last instance of the diretive are assigned locations starting from the current location, which will be updated to one past the last constant processed. However, the actual value is not computed until the end of the program so that forward references can be used.

If there are constants that have not been processed when the end of the program is reached, they will be automatically processed and placed just after the last location used. This might or might not be what is intended, so explicitly placing them is advised.

Each memory bank has its own context that holds the global symbols and constants, the current location, etc. The context is created the first time a bank is specified by the *bank* directive. The contexts are automatically switched when banks are switched.

A cross-bank reference looks up the given symbol in the global symbol table of the target bank and becomes the full 16-bit address if that symbol. If the symbol does not exist currently in the bank, it will be created. If it is never reolved in that bank, an error will be given at the end of the program.

Three code generators are implemented, one that emits correct macro1 code that can be assembled by it, and one that emits binary suitable for rim loading, and one for generating listings. The parser side knows nothing about the details of the code generators, it just manages the symbol tables and creates the parse tree.

Wrapping the parsing and code generation is control logic that manages the files and optionally passes the input through **cpp**.

The companion tool, **mactoam1** handles the syntactic differences between **macro1** and **am1** and deals with translating them, and also understands the original *define-terminate* syntax and converts that to **cpp** *#define* statements. It is implemented using flex.

Building am1

Be sure you have flex and bison installed, then:

Just type make.

Usage

am1 [-Wabmlnvz[xykp]] [-Dsymbol]... [-Ipath]... [-ipath] sourcefile

- W don't print warnings
- a space means add, default is or
- b generate binary tape image code, the default action

- m generate **macro1** code
- l generate a program listing
- n don't run **cpp** on the input
- v print the version number and exit
- z replace -0 with 0 for math operation results
- Dsymbol define a symbol for **cpp**, -Dsym or -D sym are both accepted
- Ipath add a search path to **cpp** for "files", -Ipath or -I path are both accepted
- ipath set the root directory for searches, -ipath or -i path are both accepted

These additional flags are generally for debugging:

- x send lex debugging output to stderr
- y send yacc debugging output to stderr
- k keep intermediate **cpp** file
- p print the internal parse tree in readable form

Both **macro1** and binary code can be generated at the same time.

If warnings are disabled, errors will still be printed.

Both **macro** and **macro1** treat *a space b* as *a + b*, which leads to some tortuous expressions in code when multiple operands are combined. By default, **am1** treats a space as an *or* operation, which makes much more sense. However, the original behavior can be used via the *-a* flag.

The 1's complement -0 value, all bits set, when produced by a math operation in an expression is normally preserved. The *-z* flag overrides this and converts -0 to +0, all bits cleared. Again, this only affects the results of the binary math operators and unary minus, other values are not altered.

If the 'system' include files are not installed in the default location, /opt/pidp1/Am1Includes, then the location should be specified either by using the *-i incroot* switch or by setting the environment variable *AM1INCDIR* to the full path to the location.

The -i flag has priority, followed by the environment variable, followed by the default.

At runtime, /usr/bin/cpp must exist if preprocessing is being done.

Listing file

The listing file is more complex than the **macro1** version in order to list code that is **#include**ed. Each file being processed will have an initial line identifying the file the listing following is from.

Multiple statements on a line will be listed on separate lines, but the line number will be the same for all.

Emptpy lines and comments do not show line numbers, only actual statements do so.

#define statements are not listed, but the expansion is.

While a listing can be produced for either macro or binary, the binary value for each location is only available if binary has been generated. For macro only output, the value field is meaningless. Of course, **macro1** will produce its own listing file, so creating one via **am1** is fairly useless other than seeing macro expansions.

General program structure

All programs start with a title line. This is mandatory. If you forget it, then whatever the first line is in the source will be the title, probably not what was expected.

After the title line, any number of program lines are given, which can include standard C preprocessor directives such as #include, #ifdef, etc.

The assembler has a current location counter. This corresponds to the current memory address being used. It is initially set to location 4 and is changed as statements are assembled.

Program lines consist of *statements*, each of which is an *expression* or a *directive*, pseudo-instruction in **macro1** parlance terminated by a new line or a semicolon, ;. Using a semicolon allows multiple statements on one line. A semicolon does not change the current line number but does change the current location counter. A new line changes both.

Comments are also allowed using the **C** notation:

// comment
or
statement.... // comment

Each statement generally increments the current location by one, but see more below. Comments and empty lines do not affect the current location.

Finally, the last line must be a *start xxx* statement to tell the loader where to start executing.

The original **macro1** was not particulary good at reporting errors, and a missing start would generally cause unintened behavior. This statement is now mandatory, enforced by the assembler.

Expressions

An *expression* is a sequence of *symbols*, *operators*, and *numbers*. Expressions are evaluated and the resulting value becomes the 18-bit result.

All the terms in an expression are evaluated at assembly time to give an integer value; there is no computation done during the execution of an instruction.

The PDP-1 used 1's complement math, which no modern computers use. But, **am1** runs on a modern computer and uses 2's complement math. It automatically converts the results of math operations (+, -, *, /, %, and unary minus) to the 1's complement representation during computations, the values in the binary output will be the correct 1's complement values.

Remember that one annoying thing about 1's complement is that there are two values for zero, 0 and -0. 0 is all bits off, -0 is all bits on, and it is maintained in the output. For example, the operation *-1 + 1* results in -0, not 0 as you would normally expect.

This only applies to binary mode; for macro mode, **macro1** deals with this itself.

Symbols

A *symbol* is a string of characters composed from an initial upper or lower case alphabetic character or underscore, followed optionally by any number of alphanumeric characters, digits, and underscore e.g. *scratch_Location_1*.

Note that while **am1** allows effectively unlimited symbol lengths (1023), **macro** only allows 3 and **macro1** only allows 6. More precisely, symbols must be unique in the first 3 characters for **macro**, 6 for **macro1**.

Unlike **macro1**, a digit cannot be used as the first character of a symbol.

Symbols are of four general and one special types.

- location symbol e.g. *abc34*, which represents a named location in memory
- local symbol, e.g. *%mylocal*, which represents a location specifically within a local block
- reserved symbol, e.g. *lac*, generally an opcode
- variable
- constant symbol, e.g [123]

Location symbols

An example of location symbols is:

```
    dac foo
    jmp bar
foo, 0
bar, .....
```

The first is a *reference*, its value is the memory location assigned to foo.
The second assigns foo a memory location, which is the current program location.
It is an error to reference a symbol that never has a location assigned to it, and it is an error to define a location for the same symbol more than once in the same memory bank.

Note that this is a departure from **macro1** which allow arbitrary redefinition of *any* symbol, a bad idea.

A location symbol is *defined* when its name is first used. It is *resolved* when it is used as a location. In the example, the first use of *foo* is a definition, the second is the resolution. The second form is also called a *location asssignment*.

Local symbols

Local symbols are a variation of location symbols. They are defined between *local* and *endloc* directives,

```
    local
    %target, iot 31
    .
    .
    .
    jmp %target
    endloc
```

Unlike **macro1** where all symbols are global and so all symbols must be unique, local symbols allow code to be written that won't clash with other code. The local symbol exists only within the local block and the generated code uses pc-relative addressing to access it.

Local blocks can be nested up to 1024 deep. A local symbol in an enclosing block can be referenced within a nested block, and regular location symbols can be used and declared within local blocks.

Variables

Variables are actually just a shorthand way of representing a common operation, which is naming a location in memory.

Variables must be explicitly declared via the *var* directive but then can be used just like any location symbol.

This:

```
     var a, b=34+c
     dac a
     jmp b
```

and this:

```
     dac a
     jmp b
     ...
  a, 0
  b, 34+c
```

are exactly equivalent. Also see the *variables* directive below.

Constants

Constants are another shorthand, but with special behavior.

Examples of constants are:

```
     lac [123]
     lio [a+456]
```

Functionally, the above is eqivalent to:

```
     lac const1
     lio const2
     .
     .
  const1, 123
  const2, a+456
```

But, constants are also tracked and kept in a *constants pool*. Sucessive uses of the same constant all share one memory location for their value, saving space.

If a constant is the last thing on a line, the trailing] can be omitted.
Note that *last* means **last**, no comments, etc. allowed.
Best practice is to always close the constant.

Also see the *constants* directive, below.

Tables

Tables correspond to the **macro1** *dimension* pseudo-op. A table statement reserves a number of words, optionally initialized to a value. The table keyword is followed by an expression that must be able to be fully evaluated at that time.

Examples are:

```
table 10
table 5*7
```

An optional expression can follow which will be the value to fill the table with. If no value is given, the table will not be initialized:

```
table 10, 'z'
```

fills the table with the ascii value of the letter *z*.

The table keyword is a statement, and as such must be on a line by itself. This means that:

```
foo, table 5
```

is **not** legal.

Functionally, it is eqivalent to the same number of the initializer value, e.g.

```
table 5, 17
-or-
17
17
17
17
17
```

Or, if there is no initializer:

```
table 5
-or-
.+5/
```

Using an initializer means that a 'tape' word will be written for each element of the table, which for large tables can significantly increase the size and load time.

Operators

The following operators are defined and listed in order of increasing priority.
Note that some are left associative, but most are right associative. (look it up)

- | ^ bitwise or, bitwise xor
- & bitwise and
- + - addition subtraction
- * / % multiply divide modulo
- ~ bitwise complement
- unary minus, -n
- ( ) expression nesting, the expression within the parentheses is evaluated as a group

Internally, 2's complement arithmetic is used for the math operators, but the result is adjusted to be a 1's complement value. The 1's complement -0 value, 7777777, can be produced by math operations such as -1+1, but by default will be converted to +0. This can be overridden if -0 is to be kept, see *Usage*.

Bitwise operations can of course affect any bit and their result is not adjusted, they are not math.

Numbers

Numbers are just that, an optional leading - followed by sequence of digits. However, the interpretation of the digitis can be either as octal or decimal numbers, depending upon the current radix, see *octal* and *decimal* below.

Several special representations for numbers are also provided, they override the current radix:

- 0oNNN an octal number, N must be 0-7
- 0dNNN a decimal number, N must be 0-9
- 0xNNN a hexadecimal number, N must be 0-9a-f or A-F

There are two additional special number representations:

- char cl, cm, or cr
- 'c'

The first is the same as the **macro1** version, *c* must be a valid *unshifted* Flex/Concise character, and the lmr specifies which 6 bit field in the resulting 18 bit value the character's value is place.

The second results in the value of the ascii character, but the usual escapes are allowed:

- \t tab
- \e escape
- \b backspace
- \^c control character c
- \f formfeed
- \n newline
- \NNN the value represented by the 3 **octal** digits

- \\ backslash

Even though the above are interpreted in the base they specify, when emitted in the output, they will be given as the equivalent value in the current radix, octal or decimal. For example, if the current radix is octal, then 0x1f will be emitted as 37, the octal equivalent.

Directives

Directives, aka pseudo-instructions, tell the assembler to do special operations. Some generate code, some don't.

Directives are:

- location assignment, xxx/
- octal
- decimal
- flexo
- text
- ascii
- variables
- constants
- bank
- start
- two special directives, see below

Location assigment

Location assignments directly set the current location to the value given, which must be an expression that evaluates *at that time* to a value. This means that for any use of a symbolic location, that location must already be resolved. Using a forward reference actually makes no sense, and **macro1** doesn't allow it either.

Examples are:

```
100:    the location counter is set to 100
.+10:   the location counter is incremented by 10

a, foo
a+10:   the location couter is set to the location of foo plus 10, a must be defined *prior* to its use
```

Octal and decimal

These two set the current radix for numbers entered without an explict radix specification. They stay in effect until the next one is seen. Any 'bare' number will be interpreted in that radix and an error given if the digits 8 or 9 are used in octal mode.

Examples are:

```
decimal
law 9   results in the octal word 700020

octal
law 9   will give an error
```

Regardless of the radix, the proper binary representation results.

Flexo

This packs up to 3 flex/concise characters into an 18-bit word. The first character is in the high 6 bits, the next in the middle 6 bits, the last in the low 6 bits. If not enough characters are given to fill the word, the space character, octal 0, is used.

NOTE: Remember that upper and lower case characters require a shift character, so something like

```
flexo Ok?
```

will not do what you expect. That sequence actually requires 6 characters, uppershift, O, lowershift, k, uppershift, ?. Unlike **macro1**, this is detected and a warning given.

Text

The **text** directive inserts a block of flex/concise characters packed 3 to a word. The location counter is incremented by the number of characters divided by 3, plus 1 if the number of characters wasn't a multiple of 3.

The string of characters is surrounded by a marker character, which can be any valid ascii character. The enclosed ascii characters are converted to flex/concise characters automatically.

Examples:

```
    text "this is text"
    text !This is text!
```

The first results in 4 words, 3 characters per word, packed first in the high 6 bits, second in the middle 6 bits, third in the low 6 bits. The location counter is advanced by 4.

The second results in 5 words because of the need to insert an upper-shift character and a lower-shift character. The location counter is advanced by 5.

Ascii

The **ascii** directive inserts a block of ascii characters packed 2 to a word and always terminated by a binary 0 byte, as **C** strings are. The first character is in the high 9 bits of the word, the second in the low 9 bits. In each case, the high bit is not used and will always be 0.

The location counter is advanced by the number of characters divided by 2. If there are an even number of characters, one additional word is written and the location advanced by one more.

The string of characters is always surrounded by double-quotes, ", and the same escape sequences supported by the **C** number form are supported.

Examples:

```
    ascii "this is text"
    ascii "this is text!"
```

The first results in 7 words, 6 for the 12 characters and 1 for the terminating binary 0.

The second also results in 7 words, 6 plus the high 9 bits in the 7th word for the characters with the remainder of the 7th word binary 0.

In both cases the location counter is advanced by 7.

Variables and constants

These cause any declared variables or constants to be emitted at the current location. Subsequent declarations will be held until the next variables or constants constants.

If no directive is given, any variables and constants will be written at the end of the program, the location where the start directive is given. This might not place these where you want, so using the directives is advised.

The location counter is updated appropriately after either directive.

Bank and cross-bank references

The bank directive causes all following code to be placed in the bank given, which must be between 0-31 decimal, 0-37 octal.

Each bank has its own global symbols and constants. These are preserved for each bank, so if a subsequent bank command switches back to an earlier bank, all of its symbols and constants will still be valid. The current location in that bank is also saved, and will be the same as it was when a bank switch was done.

The first time a bank is switched to, **the current location will be set to 0**.

Remember to use the *constants* and *variables* directives in *each* bank where constants or variables are used.

Global location symbols in one bank can be referenced from another bank by using the *bank reference* modifier on a location symbol, *sym:bankno*.

When it is used, the value is the 16-bit address of that symbol. If the location symbol is not defined in the target bank, it will be created. However, it must be resolved at some point in that bank or an error will be generated.

For example:

```
100/
eem

bank 1
200/
lac a
a, 0

bank 0
lac a:1
```

In bank 0, a:1 will have the value 10201. In bank 1, a will have the value 201, its in-bank address.

Additionally, as a convenience, the form *integer:bankno* can be used. This is just a shorthand for *(bankno << 12)* + *integer*, e.g. 100:3.

Code using banks and cross-bank references must understand the standard PDP-1 extended memory mode access rules.

The **macro** and **macro1** compilers do not support extended memory, so code using banks will generate code that isn't actually usable, but it will be annotated to show where banks were switched.

Some useful macros for dealing with cross-bank references are provided in the <memory.ah> include file.

Start

The start directive must be the last statement in a program, and this is enforced. It tells the loader where to begin running the program. The start address can be a numeric address or a location symbol. The start address can also be in any bank and can be a shared location symbol. Examples are:

```
start 100
start begin
start 20100 // start at location 100 in bank 2
start begin:3 // start at the location of *begin* in bank 3
```

Any defined location can be used.

Special directives

There are two special directives that are not of general use.

One is *%forcelocal*, which is a terrible hack to allow **mactoam1** to convert macro-style defines to **cpp** style. It causes any location symbol, even if not marked as local, seen in a local scope to be assumed local. If that same symbol is then used outside local scopes, it is made non-local.

The second is the **cpp**-inserted line and file marker, used to keep the current file name and line number correct. It will always be on a line by itself and is of the form:

```
# number "filename"
```

It is recommended that neither be used directly.